

AN OPTIMAL SIMULATION OF COUNTER MACHINES*

PAUL M. B. VITÁNYI†

Abstract. Each multicounter machine can be simulated by an oblivious one-head tape unit in real-time, using logarithmic space. The solution uses redundant symmetric number representation and implicit recursion. It represents a new positional representation for (vectors of) the integers.

Key words. counter machines, multicounter machines, real-time simulation by one-tape Turing machines, redundant symmetric number representation, implicit recursion, counting, coding

1. Introduction. The idea of counting, that is, adding or subtracting a unit from any given number, to obtain another one, is the substrate of arithmetic if not of all of mathematics. Thus, it is frequently necessary in computing to maintain many counts simultaneously, while the only information we want to extract at any time is the set of currently zero counts. The process of storing several integer counts, each count independently being incremented or decremented by a unit in each step, governed by the current input and the set of zero counts, is abstracted and formalized in the notion of a multicounter machine. Such machines have numerous connections with both theoretical issues and more or less practical applications. It is of considerable interest, for many questions, to implement multicounter machines as efficiently as possible. We shall show that counting is basically simple, in the computational complexity sense of the word, by demonstrating that each multicounter machine can be simulated in real-time by an oblivious one-head tape unit using minimal storage space. Since the presented implementation is optimal in all commonly considered resources at once, the two decade old quest for better simulations of multicounter machines by Turing machines is finalized in one stroke.

Doing arithmetic presupposes number representations. Different representations are better suited to different arithmetical operations. All of arithmetic can be performed by multicounter machines. Because we shall simulate a multicounter machine by a one-head tape unit, we need to straightforwardly represent a vector of integers as a linear string. No known representation for single integers allows the counter steps to be performed by an oblivious one-head tape unit without unbounded time loss in between simulated steps. Neither does any known representation, for pairs of integers, allow the counter steps to be performed by a one-head tape unit, oblivious or not, without unbounded time loss in between simulated steps. To achieve our objective, we in effect have to develop a new representation, with the required properties, for vectors of integers.

Multicounter machines and Turing machines. For the present purpose, machines are viewed as *transducers*, that is, as abstract storage devices connected to input and output terminals. Thus we consider a machine as hidden in a black box with input and output terminals. Consequently, the presented simulation results concern the input-output behavior of black boxes and are independent of input-output conventions, or whether we want to recognize or to compute. The abstract storage structure embodied by a *k-counter machine* (*k*-CM) consists of a finite control connected to an input and

* Received by the editors February 8, 1983. A preliminary exposition of this work appeared in the Proceedings of the 14th ACM Symposium on Theory of Computing (STOC) held in San Francisco, California, May 5-7, 1982. The present work is registered at the Mathematical Centre as IW 216/82.

† Centre for Mathematics and Computer Science, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands.

an output terminal, and k counters each capable of containing any integer. The states of the finite control are partitioned into *polling* and *autonomous* states. (Here we can assume without loss of generality that all states are polling states.) At the start of a computation the finite control of the k -CM is in a designated *initial state* and all counters are set to zero. A *step* in a k -CM computation is uniquely determined by the state of the finite control, by the symbol scanned at the input terminal if the state is a polling state and the set of counters which contain zero. The action at that step consists of independently altering the contents of each counter by -1 , 0 , or $+1$, changing the state of the finite control and producing an, possibly empty, output string. Thus the machine effects a transduction from input strings to output strings. If you will, the input and output may be thought of as written on input and output tapes, on which the resident access pointers (heads) are steered by the finite control. The steering commands issued can be viewed as part of the output. Above we closely followed the formulation in [2] where also a more precise definition can be found. For the more standard concept of *multitape Turing machines* consult [2], [6]. Note that, for us, a one-tape Turing machine consists of a finite control connected to an input and output terminal, and a single head storage tape. A one-head tape unit is a one-tape Turing machine.

Simulation. A machine A *simulates* a machine B in time $T(n)$ if, for all $n > 0$, the input/output behavior of B during the first n steps, the atomic inputs and outputs ordered according to their occurrences in time, is exactly mimicked by A within the first $T(n)$ steps. That is, if for every input sequence i_1, i_2, \dots read from the input terminal: (i) the output sequences written to the output terminals by A and B are the same, and (ii) if $t_1 \leq t_2 \leq \dots \leq t_k \leq t_{k+1} \dots$ are the steps at which B reads or writes a symbol, then there are corresponding steps $t'_1 \leq t'_2 \leq \dots \leq t'_k \leq t'_{k+1} \dots$ at which A reads or writes the same symbols, and $t'_i \leq T(t_i)$ for all $i \geq 1$. For a *linear time* simulation it is required that $T(n) \in O(n)$; for a simulation with *constant delay* that $t'_{n+1} - t'_n \leq c(t_{n+1} - t_n)$ for some fixed constant c and all n ; for a *real-time* simulation that $T(n) = n$. It is well known that a constant delay simulation can always be sped up to a real-time one, but not a linear time simulation in general. We use *simulation* in the above strong sense of *on-line simulation* [6] throughout.

Obliviousness. A Turing machine is *oblivious* if the movements of the storage tape heads are fixed functions of time independent of the particular inputs to the machine. Many problems seem inherently oblivious: the usual algorithms for computing the four main arithmetic operations, a table look-up by sequential search, can easily be programmed obliviously without sacrificing worst case time efficiency. Other tasks like binary search or sorting are, it appears, nonoblivious in nature. For many purposes, there are excellent reasons to restrict attention to oblivious computations [6], [7]. Here we show yet another, more heuristic, motive for doing so. Viz., restriction of the considered model of computation to its oblivious version may shift the emphasis in the problem to be solved, from one difficulty to a completely different one, thus directing us to a solution. Whereas the difficulty in real-time simulating k -counter machines by k' -tape Turing machines, $k' < k$, stems from the fact that $k' < k$, the same problem with the simulating machine restricted to its oblivious version knows as difficulty but the obliviousness of the simulating device alone.

For suppose we can simulate some abstract storage device S in time $T(n)$ by an oblivious Turing machine M . Then we can also simulate a collection of k copies of S , say S_1, S_2, \dots, S_k , interacting through a common finite control, by dividing all storage tapes of M into k tracks, each of which is a duplicate of the corresponding former tape, and by an appropriate modification of M 's finite control. The same head move-

ments of the resulting machine M' can now do the same job, on each of the k collections of tracks, as they formerly did on the collection of tapes of the original machine M . So the resources used by M' are, apart from sizes of finite control and alphabets, the same as those used by M . In particular this holds for time and storage complexity. Therefore the following two statements are equivalent:

- (i) We can simulate an abstract storage device S by an oblivious Turing machine M in time $T(n)$ and storage $S(n)$.
- (ii) For each $k > 0$ we can simulate a collection of k copies of S , interacting through a common finite control, by an oblivious Turing machine M' in time $T(n)$ and storage $S(n)$, where M' has the same tape/head constellation as M .

We are in particular interested in the following specialization of the above maxim.

Define the *quintessential counter* S as a 1-CM with input commands “add δ ”, $\delta \in \{-1, 0, 1\}$. At each step S reads an input command from the input terminal, modifies the stored count in the obvious way, and outputs either “count equal zero” or “count unequal zero” in concordance with the current state of affairs.

PROPOSITION 1. *If we can real-time simulate the quintessential counter S by an oblivious one-head tape unit then we can real-time simulate each multicounter machine by an oblivious one-head tape unit (which for each multicounter machine makes the same head movements as a function of time alone).*

Background. Counter machines are relatively old devices in computer science. Unrestricted 2-counter machines were shown to be as powerful as Turing machines in [5]. Subsequently the efficiency of implementations on Turing machines was investigated. On linear arrays, as formalized by Turing machines, the use of a tally representation for each count either requires a separate access pointer (storage tape head) per count or unbounded update time in between simulated steps. Curiously, even with the use of a separate pointer for each count, binary representations also require unbounded update time, although minimal storage space. This sorry state of affairs was improved in [1], [2] which both presented linear array simulations using minimal space, while [1] eliminated the unbounded update time at the cost of retaining all access pointers and [2] eliminated all access pointers but one at the cost of retaining unbounded update time. Thus, [2] exhibited the classic linear time/logarithmic space simulation of multicounter machines by one-tape Turing machines. Efforts to reduce this simulation to a real-time one using a fixed number of storage heads failed, but did produce some weaker problems. For the Origin Crossing Problem, where the task is to recognize the set of sequences of unit basis vectors in k -space, $k \geq 1$, which leave from and end in the origin, an ingenious solution by a real-time one-tape Turing machine was constructed in [1]. The result implies that each k -counter machine can be real-time simulated by a k -tape Turing machine in logarithmic space, $k \geq 1$. Next in difficulty comes the Axis Crossing Problem, where the task is to recognize the set of sequences of unit basis vectors in k -space, which leave from the origin and end in one of the $(k-1)$ -dimensional hyperplanes with one zero coordinate, $k > 1$. For no $k > 1$, a real-time solution on but a $(k-1)$ -tape Turing machine was found, for the k -dimensional Axis Crossing Problem, after its proposal in [2].

In [8] we made the linear time/logarithmic space one-tape solution of [2] oblivious, retaining the same resource bounds. This is a matter of some significance, since by its nature an oblivious Turing machine is usually far slower than a nonoblivious one. For example, each oblivious multitape Turing machine needs $n \log n$ steps to simulate n steps of a single pushdown store, although an oblivious 2-tape Turing machine can achieve this bound [6]. For oblivious one-tape Turing machines the lower bound on this simulation time increases, perhaps, up to n^2 . Due to the compact way the counts

can be stored, the situation for counter machines was somewhat better. In [7, Cor. 2] it was shown how to simulate each multitape Turing machine, using at most $S(n)$ storage in n steps, by an oblivious 2-tape Turing machine in $n \log S(n)$ steps and $S(n)$ storage. So the previously best simulation of multicounter machines, by combining [2] and [7], yielded a $n \log \log n$ time and $\log n$ storage simulation by an oblivious 2-tape Turing machine. Since the thrust of [8] was to achieve fast low-cost combinational logic networks implementing multicounter machines, as an expedient intermediate next result a real-time simulation by, basically, a linear iterative array with a restricted amount of oblivious local rewriting was proposed. Although not very elegant, this intermediate model served its purpose in yielding an optimal implementation of multicounter machines on combinational logic networks and, perhaps more important, the ideas embodied in the method suggest the approach to the final simulator presented here.

Outline of the paper. The objective is to construct an oblivious one-head tape unit capable of simulating any multicounter machine in real-time. In § 2 a stylized version of such a simulator is exhibited and shown to work. This version, one of many which are possible on the basic underlying principles, is chosen because it is at once amenable to short rigorous proofs of validity and achieves, it seems, the utmost frugality of machinery. To a large extent this gain is obtained at a cost of loss of intuition as to how and why it does what it is supposed to do. To counterbalance this expository defect, we insert some informal comments. The reader may also follow the genesis of the result by consulting [8] and the earlier version in the STOC Proceedings. In § 3 we enlarge on the optimality of the result, its connection with number representations, and on additional fruit borne.

2. The simulation. After some vain attempts to real-time simulate multicounter machines by Turing machines with a fixed number of tapes, one gets the feeling that, anyway, a real-time simulation by an *oblivious* one-head tape unit is out of the question. In the event, intuition is wrong; but let us informally consider the matter in some more detail. It quickly becomes apparent that updating a count, in real-time on an oblivious machine, requires a redundancy in notation which seems to make a simultaneous real-time check for zero impossible. To achieve the latter, we allow only encodings of integers such that an integer is zero iff the scanned position of the encoding (the “first” position, so to speak) shows this uniquely. Since the head motion is supposed to be oblivious we must, roughly speaking, update each “initial” $\Omega(\log i)$ length segment (situated around the head) of the encoded integer within each interval of i steps, for all $i \geq 1$. While moving the head to update longer segments of code in an oblivious manner, we may have actually stored small counts which may reach zero during this motion. So the machine has to simultaneously shift and update smaller segments of code, while updating larger segments of code, and so on recursively down to the smallest segments. Such considerations force compact encodings, and, apart from giving us some feel for what behavior is necessarily involved in a simulation as desired, they show that the integer representation used must be positional in nature.

Outline of the simulation. The simulation splits naturally into two parts. First we introduce a redundant binary representation for the integers, and formulate certain minimal requirements for real-time maintaining the representation of the stored integer under the counter operations. These requirements consist in a fixed strategy, of accessing constant length segments of this representation, for all input streams. Second, we construct an oblivious one-head tape unit capable of implementing these requirements in real-time.

The current count of the quintessential counter, as figuring in Proposition 1, is stored on the single tape in a (garbled form of) redundant binary representation, with marked most significant nonzero digit and leading distinguished blank symbols. As a consequence of preserving some invariants, the stored count equals zero iff the “first” position of the representation is a blank. Since this first position shall reside in the finite control of the simulator, that situation is instantly recognized.

Hence the problem is solved, if we can real-time update the representation of the current count while preserving the invariants. In the chosen representation it suffices to update each segment of the $2i$ th through $(2i+3)$ th position of the representation at least once within each interval of 3^i consecutive steps, for all $i \geq 0$, while also processing the current input commands, by an update of the first two positions, in each step. Intuitively speaking, the timing allows us to propagate carries and borrows (negative carries) fast enough. Although there is a considerable freedom about how to implement the required datamovement on an oblivious one-head tape unit, we choose for frugality in attendant machinery and minimal bit compression (that is, a small storage tape alphabet). Therefore, we divide the representation into blocks of two digits each, and store the first three blocks in the finite control. Each digit of the representation residing on the tape is tagged with an opening or a closing bracket, viz. the first digit of a block with an opening bracket and the second one with a closing bracket. To access each segment of the $2i$ th through $(2i+3)$ th digits of the representation at least once in every interval of 3^i steps, we develop a method of recursively transporting the digits of block j , from one side of the combination of the first i blocks to the other side, back and forth, for all i, j , $1 \leq i < j$. This transport, which entails moving the total combination of the first i blocks, in its turn supplies the necessary motion for the combination of the first $i+1$ blocks, while it also allows the single head to access blocks $i+2$ and $i+3$ within the timing constraints. The single head, without being able to determine the positional index of the scanned digits (since there will be all in all but four tags, viz. two types of opening brackets and two types of closing brackets), preserves a topology which allows it to single out and update due segments. The net effect will be that, for all i simultaneously, the combination of the first i blocks acts like a very fat head, moving slower the greater i is, but fast enough to do the same job to blocks $i+j$ as the head itself does to blocks j , for all $i, j \geq 1$.

On notation. To be able to express and prove the subsequent constructions, it is convenient to introduce some notation first. The objects operated upon are *linear arrays* or *strings* of symbols from a finite alphabet. Arrays can be finite or one-way infinite. In a one-way infinite array $A[0:\infty]$, $A[0]$ is the first element and $A[i]$ is the $(i+1)$ th element, $i \geq 0$. $A[i:j]$ denotes the $(j-i+1)$ -length subarray consisting of the $(i+1)$ th through $(j+1)$ th elements, $0 \leq i \leq j$. The concatenation $A[i:j]A[j+1:k]$ equals $A[i:k]$, $0 \leq i \leq j < k$, and we identify $A[i:i]$ with $A[i]$, $i \geq 0$. Finite arrays are treated similarly. Arrays are operated upon by functions from arrays to arrays. Since these functions shall be partial we introduce the *undefined* array \emptyset . By definition, for any array A , $\emptyset A = A\emptyset = \emptyset$. The undefined array should be distinguished from the *empty* array ε for which by definition, for any array A , $\varepsilon A = A\varepsilon = A$. Mappings from arrays to arrays are defined in terms of length preserving functions from finite arrays to finite arrays. If a function P maps an array S to an array S' , with S, S' finite and of equal length, then we write $P: S \mapsto S'$. By definition $P: \emptyset \mapsto \emptyset$ for all functions P . Functions induce relations amongst one-way infinite arrays in essentially two ways. In the first type of relation $\overset{Q}{\mapsto}$ the argument of Q determines integers i, j , $i \leq j$, and for all arrays $A[0:\infty]$, $A'[0:\infty]$ if $P: A[i:j] \mapsto A'[i:j]$, for a function P associated with Q ,

$A'[0:i-1] = A[0:i-1]$ and $A'[j+1:\infty] = A[j+1:\infty]$, then $A \stackrel{O}{\Rightarrow} A'$. In this case, clearly $\stackrel{O}{\Rightarrow}$ is a *function* from one-way infinite arrays to one-way infinite arrays. In the second type of relation $\stackrel{P}{\Rightarrow}$, P is a function, and $A \stackrel{P}{\Rightarrow} A'$ if $A = S_1 S S_2$, $A' = S_1 S' S_2$ and $P: S \rightarrow S'$. It will be shown that all such relations of the second type we consider are also *functions* when restricted to a set of *well formed* arrays. In both cases, if for some $\stackrel{O}{\Rightarrow}$ and some array A , there is no $A' \neq \emptyset$ such that $A \stackrel{O}{\Rightarrow} A'$, then $A \stackrel{O}{\Rightarrow} \emptyset$. Considering the relation $\stackrel{O}{\Rightarrow}$ amongst arrays as *rewriting*, the rewriting shall thus be proved to be always *monogenic*, that is, if $A \stackrel{O}{\Rightarrow} A'$ and $A \stackrel{O}{\Rightarrow} A''$ then $A'' = A'$. We *compose* functions P_1, P_2, \dots, P_n to a function P , or *decompose* or *expand* a function P into a sequence of constituent functions P_1, P_2, \dots, P_n as follows. If for some P, P_1, \dots, P_n and all arrays A there exist arrays A_1, A_2, \dots, A_n such that $A \stackrel{P}{\Rightarrow} A_n$ and $A \stackrel{P_1}{\Rightarrow} A_1 \stackrel{P_2}{\Rightarrow} A_2 \cdots \stackrel{P_n}{\Rightarrow} A_n$ then $P = P_1; P_2; \dots; P_n$. The function composition operator “;” denotes sequential rewriting from left to right. Whenever necessary, we denote the value of an array A at time t , $t \geq 0$, by A^t and A^0 is the *initial* array. We dispense with the superscript if t is understood or when we view A as a variable.

Main objective. We concentrate on real-time simulating the quintessential counter of Proposition 1 by an oblivious one-head tape unit.

2.1. An integer representation. Consider a positional base 2 notation for representing the integers, which may be called *redundant symmetric binary*, using the digits $-2, -1, 0, 1, 2$. So the integer c represented by $c_0 c_1 c_2 \cdots c_m$, $c_i \in \{-2, -1, 0, 1, 2\}$, equals $\sum_{i=0}^m c_i 2^i$. Such a representation is *binary* because of the weight of digits in distinct positions, *symmetric* because of the used digits, and *redundant* since each integer has infinitely many representations, even without leading nonsignificant zeros. To represent the stored integer count on a, potentially infinite, linear tape we essentially use a restricted version of this representation, with a marked most significant nonzero digit and distinguished leading nonsignificant zeros. Let $\Delta = \{-2, -1, 0, 1, 2\}$ and $\bar{\Delta} = \{-\bar{2}, -\bar{1}, \bar{0}, \bar{1}, \bar{2}\}$. The barred digits have the same value as their nonbarred counterparts, $\bar{\Delta} - \{\bar{0}\}$ is reserved for the most significant nonzero digit, and “ $\bar{0}$ ”, called *blank*, is reserved for the nonsignificant zeros. Let $\Sigma = \Delta \cup \bar{\Delta}$ and let *code*: $\mathbb{Z} \rightarrow 2^{\Sigma^\infty}$ be a function of the integers into the power set of Σ^∞ , where Σ^∞ is the set of one-way infinite strings over Σ . The function *code* satisfies restrictions (A)–(D) below, for all $c_0 c_1 \cdots c_i \cdots \in \text{code}(c)$, $c \in \mathbb{Z}$.

Separation of a finite significant initial segment and nonsignificant zeros:

$$(A) \quad \exists i \geq 0 [c_i = \bar{0}] \quad \& \quad \forall i > 0 [(c_i = \bar{0} \Rightarrow (c_{i-1} \in \bar{\Delta} \ \& \ c_{i+1} = \bar{0})) \\ \& \ (c_i \in \Sigma - \{\bar{0}\} \Rightarrow c_{i-1} \in \Delta)].$$

Correct representation:

$$(B) \quad \sum_{i=0}^{\infty} c_i 2^i = c.$$

To identify representations of 0 by just a small initial segment:

$$(C) \quad \forall i \geq 0 [(c_{i+1} > 0 \Rightarrow c_i \geq 0) \ \& \ (c_{i+1} < 0 \Rightarrow c_i \leq 0)].$$

Under (A)–(C), $(-2)^i 0 \bar{1} \bar{0}^\infty$ represents the integer 2 for all $i \geq 0$. To prevent racing of the most significant nonzero digit to the first position, in just a few steps of the desired single head real-time simulator:

$$(D) \quad \forall i \geq 0 [i \text{ is odd} \Rightarrow |c_i| < 2].$$

Now if $c_0c_1 \cdots c_{m-1}c_m \cdots \in \text{code}(c)$, with $c_i \in \Delta$ for $0 \leq i \leq m-1$, and $c_i \in \bar{\Delta}$ for $m \leq i$, then for $m=0$ we have $0 \leq |c| \leq 2$, for $m=1$ we have $2 \leq |c| \leq 4$ and in general for $m \geq 2$:

$$2^m - r \leq |c| \leq 2^{m+1} + r'$$

with

$$r = 2 \sum_{i=0, i \text{ even}}^{m-2} 2^i + \sum_{i=1, i \text{ odd}}^{m-2} 2^i, \quad r' = 2 \sum_{i=0, i \text{ even}}^{m-1} 2^i + \sum_{i=1, i \text{ odd}}^{m-1} 2^i$$

which yields

$$m-3 < \log_2 |c| < m+2.$$

Thus the length of the initial significant segment of the representation of $c \in \mathbb{Z}$ follows by and large the length of the usual binary representation of $|c|$. We are particularly interested in representations for zero. Note that the following proposition holds for code functions satisfying only (A)–(C).

PROPOSITION 2. *Let $c_0c_1 \cdots c_m c_{m+1} \cdots \in \text{code}(c)$ with $c \in \mathbb{Z}$. Then $c=0$ iff $c_i = \bar{0}$ for all $i \geq 0$ iff $c_0 = \bar{0}$.*

Proof. By (A) $c_0 = \bar{0}$ iff $c_i = \bar{0}$ for all $i \geq 0$. So we only have to prove $c=0$ iff $c_0 = \bar{0}$. Assume $c_i = \bar{0}$ for all $i \geq 0$. By (B) $c=0$. Assume $c_i \neq \bar{0}$ for some $i \geq 0$. Then by (A) there exists a least $m \geq i$ such that $c_j = \bar{0}$ for all $j > m$, and $|c_m| \neq 0$. For $m=0$, $|c| \geq 1$, and for $m=1$ we have $|c| \geq 2$. For $m \geq 2$:

$$\begin{aligned} |c| &= \left| \sum_{i=0}^m c_i 2^i \right| && \text{(by (B))} \\ &\geq \left| 2^m - \left| \sum_{i=0}^{m-1} c_i 2^i \right| \right| && \text{(triangle inequality)} \\ &\geq 2^m - 2 \sum_{i=0}^{m-2} 2^i && \text{(by (C))} \\ &= 2. \end{aligned}$$

□

2.2. Maintenance of the count. Let S be the simulated quintessential counter and let $\delta_1, \delta_2, \dots, \delta_t, \dots, \delta_i \in \{-1, 0, 1\}$, be any fixed sequence of unit additions/subtractions. So at time $t \geq 0$, S contains the integer $\sum_{i=1}^t \delta_i$. We maintain the count in an array $C[0:\infty]$ such that the value of the array at time $t \geq 0$ is $C^t[0:\infty] \in \text{code}(\sum_{i=1}^t \delta_i)$, for any such input stream. The *initial* array $C^0[0:\infty]$ at time $t=0$ is defined by $C^0[i] = \bar{0}$ for all $i \geq 0$, and therefore $C^0[0:\infty] \in \text{code}(0)$. In the t th simulated step, $t \geq 1$, the current value C^{t-1} of the array is mapped to the next value C^t by a function $\text{COUNT}(t, \delta_t)$. The mapping COUNT is defined in terms of a composition of mappings, with the aid of an auxiliary function $I: \mathbb{N} \rightarrow 2^{\mathbb{N}}$, called the *parameter selection* function, which has as values sets of bounded cardinality (cardinality four suffices).

DEFINITION. For $t \geq 1$, let $I(t) = \{i_t, i_{t-1}, \dots, i_1\}$ with $i_t > i_{t-1} > \dots > i_1$, and let $\delta \in \{-1, 0, 1\}$. $\text{COUNT}(t, \delta)$ is defined as a composition of mappings:

$$\text{COUNT}(t, \delta) \stackrel{\text{def}}{=} \text{UPDATE}(i_t); \text{UPDATE}(i_{t-1}); \dots; \text{UPDATE}(i_1); \text{INPUT}(\delta).$$

Hence

$$C \xrightarrow{\text{COUNT}(t, \delta)} C', \text{ with } C' \neq \emptyset,$$

if there exist $C_b, C_{l-1}, \dots, C_1 \neq \emptyset$ such that

$$C \xrightarrow{\text{UPDATE}(i_l)} C_l \xrightarrow{\text{UPDATE}(i_{l-1})} C_{l-1} \cdots \xrightarrow{\text{UPDATE}(i_1)} C_1 \xrightarrow{\text{INPUT}(\delta)} C'.$$

In all other cases $C \xrightarrow{\text{COUNT}(t,\delta)} \emptyset$.

DEFINITION. Let $i \in \mathbb{N}$ and let $C[0:\infty]$ be a one-way infinite array, $C \neq \emptyset$.

$$C[0:\infty] \xrightarrow{\text{UPDATE}(i)} C'[0:\infty],$$

$C'[0:\infty] \neq \emptyset$, if $\text{UPDATE}: C[2i:2i+3] \mapsto C'[2i:2i+3] \neq \emptyset$ and $C'[0:2i-1] = C[0:2i-1]$ and $C'[2i+4:\infty] = C[2i+4:\infty]$, with the function $\text{UPDATE}: \Sigma^4 \rightarrow \Sigma^4$ defined below. For convenience we first define $\text{UPDATE}: \Delta^4 \rightarrow \Delta^4$ and then extend the mapping to Σ^4 .

$$\begin{array}{ll} \text{UPDATE:} & 2 \ 0 \ x \ y \mapsto 0 \ 1 \ \ x y \text{ for } xy \in \{00, 01, 0-1, 10, 11, 20, 21\} \\ & 2 \ 0 \ x \ y \mapsto 0-1 \ x+1 \ y \text{ for } xy \in \{-10, -20, -1-1, -2-1\} \\ & 2 \ 1 \ x \ y \mapsto 0 \ 0 \ x+1 \ y \text{ for } xy \in \{00, 01, 10, 11\} \\ & 2 \ 1 \ 0-1 \mapsto 0 \ 0 \ -1 \ 0 \\ & 2 \ 1 \ 2 \ y \mapsto \emptyset \quad \text{for } y \in \{0, 1\} \\ & -2 \ 0 \ x \ y \mapsto 0-1 \ \ x y \text{ for } xy \in \{00, 0-1, 01, -10, -1-1, -20, -2-1\} \\ & -2 \ 0 \ x \ y \mapsto 0 \ 1 \ x-1 \ y \text{ for } xy \in \{10, 20, 11, 21\} \\ & -2-1 \ x \ y \mapsto 0 \ 0 \ x-1 \ y \text{ for } xy \in \{00, 0-1, -10, -1-1\} \\ & -2-1 \ 0 \ 1 \mapsto 0 \ 0 \ 1 \ 0 \\ & -2-1-2 \ y \mapsto \emptyset \quad \text{for } y \in \{0, -1\} \\ & v \ w \ x \ y \mapsto v \ w \ \ x y \text{ for } v \notin \{-2, 2\} \\ & v \ w \ x \ y \mapsto \emptyset \quad \text{for } vwxy \text{ not in the above list.} \end{array}$$

Extension of UPDATE to mappings from Σ^4 into Σ^4 : if $vwxy \notin \Delta^4$ then

$$\text{UPDATE: } vwxy \mapsto v' w' x' y'$$

for all $vwxy, v' w' x' y' \in \Delta^*(\bar{\Delta} - \{\bar{0}\})\{\bar{0}\}^* \cup \{\bar{0}\bar{0}\bar{0}\bar{0}\}$ such that the unbarred version of the mapping is in the previous list, and $\text{UPDATE: } vwxy \mapsto \emptyset$ in all other cases. (Recall that if V is a finite alphabet, then V^* is the set of all finite strings over V including the empty string ε .)

DEFINITION. Let $\delta \in \{-1, 0, 1\}$ and let $C[0:\infty]$ be a one-way infinite array, $C \neq \emptyset$.

$$C[0:\infty] \xrightarrow{\text{INPUT}(\delta)} C'[0:\infty],$$

$C'[0:\infty] \neq \emptyset$, if $\text{INPUT}_\delta: C[0:1] \mapsto C'[0:1] \neq \emptyset$ and $C'[2:\infty] = C[2:\infty]$ with $\text{INPUT}_\delta: \Sigma^2 \rightarrow \Sigma^2$ defined below. For convenience we first define $\text{INPUT}_\delta: \Delta^2 \rightarrow \Delta^2$ and then extend the mapping to Σ^2 .

$$\begin{array}{ll} \text{INPUT}_{-1}: & x \ y \mapsto x-1 \ y \text{ for } xy \in \{00, 0-1, -10, -1-1, 10, 11\} \\ & 0 \ 1 \mapsto 1 \ 0 \\ & x \ y \mapsto \emptyset \text{ for } xy \in \{-20, -2-1, 20, 21\} \\ \text{INPUT}_0: & x \ y \mapsto x \ y \text{ for } xy \in \{00, 0-1, -10, -1-1, 10, 01, 11\} \\ & x \ y \mapsto \emptyset \text{ for } xy \in \{-20, -2-1, 20, 21\} \\ \text{INPUT}_1: & x \ y \mapsto x+1 \ y \text{ for } xy \in \{00, 01, 10, 11, -10, -1-1\} \\ & 0 \ -1 \mapsto -1 \ 0 \\ & x \ y \mapsto \emptyset \text{ for } xy \in \{-20, -2-1, 20, 21\}. \end{array}$$

Extension of INPUT_δ to mappings from Σ^2 into Σ^2 : if $xy \notin \Delta^2$ then

$$\text{INPUT}_\delta: xy \mapsto x' y'$$

for all $xy, x' y' \in \Delta^*(\bar{\Delta} - \{\bar{0}\})\{\bar{0}\}^* \cup \{\bar{0}\bar{0}\}$ such that the unbarred version of the mapping is in the previous list and $\text{INPUT}_\delta: xy \mapsto \emptyset$ in all other cases.

If for some array $C[0:\infty]$ and $P = \text{UPDATE}(i)$, $i \geq 0$, we have $\text{UPDATE}: C[2i:2i+3] \mapsto \emptyset$ then $C \stackrel{P}{\mapsto} \emptyset$, by definition. If for some array $C[0:\infty]$ and $P = \text{INPUT}(\delta)$, $\delta \in \{-1, 0, 1\}$, we have $\text{INPUT}_\delta: C[0:1] \mapsto \emptyset$ then $C \stackrel{P}{\mapsto} \emptyset$, by definition. For all $P \in \{\text{INPUT}(\delta), \text{UPDATE}(i) \mid \delta \in \{-1, 0, 1\}, i \geq 0\}$ we have by definition $\emptyset \stackrel{P}{\mapsto} \emptyset$, and thus $\stackrel{P}{\mapsto}$ is a *mapping* from $\Sigma^\infty \cup \{\emptyset\}$ into $\Sigma^\infty \cup \{\emptyset\}$ and not just a relation. Basically, $\text{INPUT}(\delta)$ adds the current input to the currently represented integer and $\text{UPDATE}(i)$ propagates carries and borrows in a segment of the representation, both preserving representations from the code function.

For each input sequence $D = \delta_1, \delta_2, \dots, \delta_t, \dots$, with $\delta_t \in \{-1, 0, 1\}$ for $t \geq 1$, the sequence of mappings

$$\text{COUNT}(I, D) \stackrel{\text{def}}{=} \text{COUNT}(1, \delta_1); \text{COUNT}(2, \delta_2); \dots; \text{COUNT}(t, \delta_t); \dots$$

defines a sequence of (a priori possibly undefined) arrays $C^0, C^1, \dots, C^t, \dots$ such that C^0 is the all-blank initial array $C^0[0:\infty] = \bar{0}^\infty$, and for all $t \geq 1$:

$$C^{t-1} \xrightarrow{\text{COUNT}(t, \delta_t)} C^t.$$

Decomposing $\text{COUNT}(t, \delta_t)$ into its constituent functions for all $t \geq 1$, with $I(t) = \{i_{t,l(t)}, i_{t,l(t)-1}, \dots, i_{t,1}\}$ and $i_{t,l(t)} > i_{t,l(t)-1} > \dots > i_{t,1}$, we obtain for each input sequence $D = \delta_1, \delta_2, \dots, \delta_t, \dots$ the sequence of basic mappings

$$\begin{aligned} \text{COUNT}(I, D) = & \text{UPDATE}(i_{1,l(1)}); \text{UPDATE}(i_{1,l(1)-1}); \dots; \text{UPDATE}(i_{1,1}); \\ & \text{INPUT}(\delta_1); \text{UPDATE}(i_{2,l(2)}); \dots \end{aligned}$$

In this sequence, the subsequence of mappings

$$\begin{aligned} \text{COUNT}(t, \delta_t) = & \text{UPDATE}(i_{t,l(t)}); \text{UPDATE}(i_{t,l(t)-1}); \dots; \text{UPDATE}(i_{t,1}); \\ & \text{INPUT}(\delta_t) \end{aligned}$$

is said to constitute the *t*th step of the maintenance of array C . Starting from C^{t-1} the sequence of intermediate arrays defined by the *t*th step is

$$C^{t-1} (= C_{t-1,0}), C_{t,l(t)}, C_{t,l(t)-1}, \dots, C_{t,1}, C_{t,0} (= C^t)$$

defined by

$$C_{t-1,0} \xrightarrow{\text{UPDATE}(i_{t,l(t)})} C_{t,l(t)} \dots \xrightarrow{\text{INPUT}(\delta_t)} C_{t,0} = C^t.$$

Note that in the decomposition of $\text{COUNT}(I, D)$ in the basic mappings $\text{UPDATE}(\cdot)$ and $\text{INPUT}(\cdot)$ the parameter t does not occur explicitly; the sequence of basic mappings is defined totally by the sequence of successive values of I and the sequence of inputs. This is important in the next sections. In this section we show in Lemma 1 that, for any input sequence $D = \delta_1, \delta_2, \dots$,

$$C^0, C_{1,l(1)}, \dots, C_{1,1} \in \text{code}(0) \cup \{\emptyset\}$$

and for all $t \geq 1$

$$C_{t-1,0} (= C^{t-1}), C_{t,l(t)}, \dots, C_{t,1} \in \text{code}\left(\sum_{i=1}^{t-1} \delta_i\right) \cup \{\emptyset\}.$$

In Proposition 3 it is demonstrated that for certain choices of the parameter selection function I we have that $C_{t,j} \neq \emptyset$ for all $t \geq 1$ and all $j, l(t) \geq j \geq 0$, whence $C^t \in \text{code}\left(\sum_{i=1}^t \delta_i\right)$ for all $t \geq 0$.

LEMMA 1. Let array $C \in \text{code}(c)$ for some integer c . If, for some $i \geq 0$, $C \xrightarrow{\text{UPDATE}(i)} C'$, $C' \neq \emptyset$, then $C' \in \text{code}(c)$. If, for some $\delta \in \{-1, 0, 1\}$, $C \xrightarrow{\text{INPUT}(\delta)} C'$, $C' \neq \emptyset$, then $C' \in \text{code}(c + \delta)$.

Proof. Let $C \in \text{code}(c)$ for some integer c and $C \xrightarrow{\text{UPDATE}(i)} C'$, $C' \neq \emptyset$, for some $i \geq 0$. If $|C[2i]| \neq 2$ then $C' = C$ and there is nothing to prove. So let $|C[2i]| = 2$. Then, for $j < 2i$ and $j > 2i + 3$, $C'[j] = C[j]$. Since also $\sum_{j=0}^3 C'[2i+j]2^{2i+j} = \sum_{j=0}^3 C[2i+j]2^{2i+j}$, we have $\sum_{i=0}^{\infty} C'[i]2^i = \sum_{i=0}^{\infty} C[i]2^i = c$. It is easy to check from the definition of UPDATE (i), that if (A), (C) and (D) hold for C , $C \xrightarrow{\text{UPDATE}(i)} C'$ and $C' \neq \emptyset$, then (A), (C) and (D) also hold for C' . Hence $C' \in \text{code}(c)$. Let $C \in \text{code}(c)$ for some integer c and $C \xrightarrow{\text{INPUT}(\delta)} C'$, $C' \neq \emptyset$, for some $\delta \in \{-1, 0, 1\}$. Since $C' \neq \emptyset$ we have $|C[0]| < 2$. For all $j > 1$, $C'[j] = C[j]$. Because also $C'[0] + 2C'[1] = C[0] + 2C[1] + \delta$ we have $\sum_{i=0}^{\infty} C'[i]2^i = c + \delta$. It is easy to check from the definition of INPUT (δ) that if (A), (C) and (D) hold for C , $C \xrightarrow{\text{INPUT}(\delta)} C'$, and $C' \neq \emptyset$, then (A), (C) and (D) also hold for C' . Hence $C' \in \text{code}(c + \delta)$. \square

PROPOSITION 3. Let $T: \mathbb{N} \rightarrow \mathbb{N}$ be any function such that $T(i) \leq 3^i$ for all $i \geq 0$. Let the parameter selection function $I: \mathbb{N} \rightarrow 2^{\mathbb{N}}$, associated with the mapping COUNT, be such that for all indices $i \geq 0$ and steps $t \geq 1$ there exists a t' , $t \leq t' < t + T(i)$ and $i \in I(t')$. Then for each input sequence $\delta_1, \delta_2, \dots, \delta_t, \dots$, $\delta_i \in \{-1, 0, 1\}$, $t \geq 1$, there exists a sequence of one-way infinite arrays $C^0, C^1, \dots, C^t, \dots$, with C^0 the all blank initial array and C^{t-1} is mapped to C^t by COUNT (t, δ_i) for all $t \geq 1$, such that $C^t \in \text{code}(\sum_{i=1}^t \delta_i)$ for all $t \geq 0$.

Proof. Roughly speaking the proposition states that if, starting from the all blank initial array C^0 , UPDATE (i) is executed at least once in every interval of 3^i steps, for all $i \geq 0$, and INPUT (δ) is executed each step, with $\delta \in \{-1, 0, 1\}$ the currently polled input, then the array at time t represents the stored integer at time t according to the code function. By Lemma 1 and the definition of COUNT this is the case if, under the timing assumption on the parameter selection function I , each time UPDATE (i) and INPUT (δ) map an array satisfying (A), (C) and (D), the result is not the undefined array \emptyset . The only way UPDATE (i) can map an array $C[0:\infty]$, satisfying (A), (C) and (D) to \emptyset is for $C[2i:2i+2] \in \{212, 21\bar{2}, -2-1-2, -2-1-\bar{2}\}$. Similarly, the only way INPUT (δ) can map an array $C[0:\infty]$ satisfying (A), (C) and (D) to \emptyset is for $C[0] \in \{2, \bar{2}, -2, -\bar{2}\}$. Hence we have to prove that, under the assumptions on I , and starting from the all blank initial array C^0 , these undesirable subarrays do not occur at the crucial moments. Induction is on the number of steps t .

Base case: the first step. Since C^0 is all-blank, for all $i \geq 0$ we have $C^0[2i:2i+3] = \bar{0}\bar{0}\bar{0}\bar{0}$. Hence $C^0 \xrightarrow{\text{COUNT}(1, \delta_1)} C^1$ with $C^1[0] = \bar{\delta}_1$ and $C^1[i] = \bar{0}$ for all $i \geq 1$. That is, $C^1 \in \text{code}(\delta_1)$.

Induction: $t \geq 1$. Assume, by way of contradiction, that for the input sequence $\delta_1, \delta_2, \dots, \delta_t$ ($\delta_j \in \{-1, 0, 1\}$, $1 \leq j \leq t$) we have for all j , $1 \leq j \leq t$:

$$C^{j-1} \xrightarrow{\text{COUNT}(j, \delta_j)} C^j, C^j \neq \emptyset \quad (\text{induction assumption}),$$

and

$$C^t \xrightarrow{\text{COUNT}(t+1, \delta)} \emptyset \quad (\text{contradictory assumption}),$$

for some $\delta \in \{-1, 0, 1\}$. For all j , $1 \leq j \leq t$, by Lemma 1, $C^j \in \text{code}(\sum_{i=1}^j \delta_i)$. Let $I(t+1) = \{i_t, i_{t-1}, \dots, i_1\}$ and $i_t > i_{t-1} > \dots > i_1$. Decomposing COUNT ($t+1, \delta$) into its

constituent mappings we have

$$C' = C_{l+1} \xrightarrow{\text{UPDATE}(i_l)} C_l \xrightarrow{\text{UPDATE}(i_{l-1})} C_{l-1} \cdots \xrightarrow{\text{UPDATE}(i_1)} C_1 \xrightarrow{\text{INPUT}(\delta)} C_0 = \emptyset,$$

for some intermediate, possibly undefined, arrays C_b, C_{l-1}, \dots, C_0 . By the contradictory assumption there must be a first undefined array in this sequence, say $C_{j-1} = \emptyset$ and $C_j \neq \emptyset$ for some $j, 0 < j \leq l+1$. Note that, by Lemma 1, $C_j \in \text{code}(\sum_{i=1}^j \delta_i)$.

Case 1. $j > 1$. Setting i to i_{j-1} , to avoid subscripts,

$$C_j \xrightarrow{\text{UPDATE}(i)} \emptyset.$$

Since $C_j \in \text{code}(\sum_{i=1}^j \delta_i)$ and therefore satisfies (A), (C) and (D), this can happen only if $C_j[2i:2i+2] \in \{-2, -1, -2, -2, -1, -2, 212, 21\bar{2}\}$. Assume $C_j[2i:2i+2] = 212$, the other cases being symmetrical. Since the initial array C^0 contained only blanks, there must be a $t', 0 < t' \leq t$, with $t-t'$ minimal, such that

$$C^{t'-1} \xrightarrow{\text{COUNT}(t', \delta_{t'})} C^{t'},$$

$C^{t'}[2i+2] = 2$ and $C^{t'-1}[2i+2] \neq 2$. (A previous mapping $\text{UPDATE}(k)$, with $k > i$, in the $(t+1)$ th step could not have set $C[2i+2]$ to 2 from another value, so if $C_j[2i+2] = 2$ then $C_k[2i+2] = 2$ for all $k, l+1 \geq k \geq j$. Since $C_{l+1} = C'$ indeed $t' \leq t$.) From the definitions it follows that $C[2i+2]$ can be set to 2, from another value, only by the mapping $\text{UPDATE}(i)$. So $i \in I(t')$, and we denote by C' the array mapped upon by the occurrence of $\text{UPDATE}(i)$ in $\text{COUNT}(t', \delta_{t'}) = \text{UPDATE}(i'_{t'}); \text{UPDATE}(i'_{t'-1}); \dots; \text{UPDATE}(i'_1); \text{INPUT}(\delta_{t'})$. By the definition of $\text{UPDATE}(i)$ we must have $C^{t'}[2i:2i+2] = 002$. Since during the mappings, following $\text{UPDATE}(i)$ in $\text{COUNT}(t', \delta_{t'})$, subarray $C[2i+2:\infty]$ is not accessed, and we have by Lemma 1 that $C' \in \text{code}(\sum_{i=1}^{t'-1} \delta_i)$ and $C^{t'} \in \text{code}(\sum_{i=1}^{t'} \delta_i)$, it therefore follows that

(1)

$$\sum_{k=0}^{2i+1} C^{t'}[k]2^k = \sum_{k=0}^{2i+1} C^{t'-1}[k]2^k + \delta_{t'} = \sum_{k=0}^{2i-1} C^{t'-1}[k]2^k + \delta_{t'} \leq (4^{i+1} - 1)/3 \quad (\text{by (C) and (D)}).$$

Any first occurrence of an $\text{UPDATE}(i+1)$ in a $\text{COUNT}(t'', \delta_{t'')}$, $t' < t'' < t+1$, so in between the mappings by the two occurrences of $\text{UPDATE}(i)$ in steps t' and $t+1$, would have set $C[2i+2]$ to 0, resulting in $|C^{t''}[2i+2]| \leq 1$, contradicting the minimality of $t-t'$. Therefore, for all t'' , $t' < t'' < t+1$, $i+1 \notin I(t'')$. By the assumption on I in the proposition it follows that

$$(2) \quad t-t' < 3^{i+1}.$$

We are now ready to derive a contradiction. For the only mappings which can alter something in $C[2i+2:2i+3]$ are $\text{UPDATE}(i)$ and $\text{UPDATE}(i+1)$. However, in between the mappings according to the occurrence of $\text{UPDATE}(i)$ in step t' and that of $\text{UPDATE}(i)$ in step $t+1$, no occurrence of $\text{UPDATE}(i)$ has changed $C[2i+2:2i+3]$ (since this would contradict the minimality of $t-t'$), and $\text{UPDATE}(i+1)$ has not occurred at all (since $C_j[2i+2] \neq 0$ by assumption, $i+1$ is not in $I(t+1)$ too). So, by the definitions of COUNT and UPDATE we obtain:

$$(3) \quad \sum_{k=2i+2}^{\infty} C^{t'}[k]2^k = \sum_{k=2i+2}^{\infty} C_j[k]2^k.$$

Furthermore, by Lemma 1,

$$(4) \quad \sum_{k=0}^{\infty} C^t[k]2^k = \sum_{k=1}^t \delta_k = \sum_{k=0}^{\infty} C_j[k]2^k.$$

Thus:

$$(5) \quad \begin{aligned} \sum_{k=0}^{2i+1} C_j[k]2^k &= \sum_{k=0}^{\infty} C_j[k]2^k - \sum_{k=2i+2}^{\infty} C_j[k]2^k \\ &= \sum_{k=1}^t \delta_k - \sum_{k=2i+2}^{\infty} C_j[k]2^k && \text{(by (4))} \\ &= \sum_{k=1}^t \delta_k - \sum_{k=2i+2}^{\infty} C^t[k]2^k && \text{(by (3))} \\ &= \sum_{k=1}^t \delta_k - \sum_{k=1}^{t'} \delta_k + \sum_{k=0}^{2i+1} C^{t'}[k]2^k && \text{(by Lemma 1)} \\ &\leq t - t' + (4^{i+1} - 1)/3 && \text{(by (1))} \\ &< 3^{i+1} + (4^{i+1} - 1)/3. && \text{(by (2)).} \end{aligned}$$

But, by way of contradiction, it was assumed that $C_j[2i:2i+1] = 21$. Therefore,

$$(6) \quad \sum_{k=0}^{2i+1} C_j[k]2^k = 4^{i+1} + \sum_{k=0}^{2i-1} C_j[k]2^k \geq 4^{i+1} - (4^i - 4)/3 - 4^i/2,$$

for $i \geq 2$ (and ≥ 14 for $i=1$, ≥ 4 for $i=0$), by (C) and (D). Since for all $i \geq 0$ the contradictory assumption leads to the contradictory inequalities (5) and (6) we conclude that $j=1$ and case 2 holds.

Case 2. $j=1$ and

$$C_1 \xrightarrow{\text{INPUT}(\delta)} \emptyset.$$

However, under the assumptions in the Proposition, $0 \in I(t)$ for all $t \geq 1$, so $\text{COUNT}(t+1, \delta) = \dots$; $\text{UPDATE}(0)$; $\text{INPUT}(\delta)$. But if $C_1[0:\infty] \neq \emptyset$ is the value of $\text{UPDATE}(0)$ then $C_1[0] \notin \{-2, -\bar{2}, 2, \bar{2}\}$. Therefore, the contradictory assumption also fails in this case and

$$C_1 \xrightarrow{\text{INPUT}(\delta)} C_0 \neq \emptyset.$$

Since the contradictory assumption has now been proven false, by Lemma 1 $C_{i+1}, C_i, \dots, C_1 \in \text{code}(\sum_{i=1}^t \delta_i)$ and $C_0 \in \text{code}(\sum_{i=1}^t \delta_i + \delta)$. Setting $C^{t+1} = C_0$ completes the induction. \square

Proposition 3 shows us a way of real-time simulating the quintessential counter S figuring in Proposition 1. Let C^0 be the all-blank initial array, and let the parameter selection function I meet the timing conditions in Proposition 3. If we map in the t th step, for each $t \geq 1$, the current array value to the next one by $\text{COUNT}(t, \delta)$, where “add δ ”, $\delta \in \{-1, 0, 1\}$, is the input command polled from the input terminal in the t th step, then the array at each time $t \geq 0$ is a representation from code (stored integer at time t). Since the mapping $\text{COUNT}(t, \delta) = \dots$; $\text{INPUT}(\delta)$, and $\text{INPUT}(\delta)$ maps $C[0:1]$ to a next value, we can simultaneously output “count equals zero” if the next value of $C[0] = \bar{0}$, or “count unequal zero” if the next value of $C[0] \neq \bar{0}$, according to Proposition 2.

Note that the requirement of an initial zero count is not essential. We can as well prove Proposition 3 starting from C^0 equals a representation of an arbitrary integer c . For instance, a representation from code (c) , containing only equal signed digits of absolute value less than 2, for C^0 , lets Proposition 3 go through as well. Thus, the arrangement can real-time simulate initially nonzero counters.

2.3. An oblivious one-head tape unit. Proposition 3 puts a heavy burden on a one-head tape unit: $C[0:3]$ must always be under scan, $C[2:5]$ within each third step, and in general $C[2i:2i+3]$ at least once within *each* interval of 3^i steps, for all $i \geq 0$ *simultaneously*. This requires that, basically, at all times all $C[i]$ must be on the move, drifting inward or outward from the location occupied by the single head, so to speak. This data motion must be due to the swapping of array elements amongst the momentarily simultaneously scanned tape squares. To be able to scan $C[2i:2i+3]$ within certain time intervals, for all $i \geq 0$, it is necessary that at certain times arbitrarily many of such quadruples are split and the pieces geometrically far apart. The piece $C[2i:2i+1]$ must be joined to piece $C[2i-2:2i-1]$ at certain times and to $C[2i+2:2i+3]$ at other times, for all $i \geq 1$. Apart from performing the splitting, moving and glueing, the head must also recognize quadruples $C[2i:2i+3]$ to perform UPDATE, and also know the relative order amongst pairs of such foursomes. Hence we need to maintain some order and identification of the array elements. Yet we cannot identify the individual elements of C with respect to their position, since such an identification tag for $C[i]$ needs $\log i$ space and $\log i$ time to evaluate. All this points in the direction of a recursive process, but again we cannot maintain depth of recursion parameters.

The process exhibited below rests on the following intuition. The goal is roughly to access quadruples of consecutive elements of C , of index $\Theta(i)$, at least once in each interval of $2^{\Theta(i)}$ steps, for all $i \geq 0$. We call the individual array elements *cells* and consider them as packets of information to be swapped amongst simultaneously scanned squares. Assume we are able to move a *block* of cells, called A_1 , by, according to some regime, moving the head, centered on the cells constituting A_1 , from the left end of A_1 , where it scans some squares left adjacent to A_1 , to the right end of A_1 , where it scans some squares right adjacent to A_1 , and back again to the left end of A_1 . Let A_1 be contained in a block of cells called A_2 . Then A_1 moves by transporting cells of $A_2 - A_1$ through A_1 to the other side of A_1 , while simultaneously shifting the cells of A_1 . Thus, we will shift the total block A_1 from the left end of A_2 to the right end, and back again to the left end. During such a full sweep of A_1 over A_2 , we will shift block A_2 within a larger block A_3 by a single square. So the relation between A_2 and A_3 is analogous to that between A_1 and A_2 . See Fig. 1.

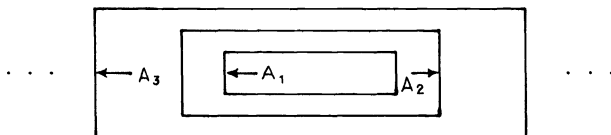


FIG. 1. The blocks are individually "moving" in the indicated directions.

In general, we envision an infinite series of nested blocks, $A_1, A_2, \dots, A_i, A_{i+1}, \dots$, with A_i properly embedded in A_{i+1} , $i \geq 1$, such that a full sweep of block A_i over block A_{i+1} shifts block A_{i+1} one square in the currently desired direction. In the above arrangement, the head is *always* centered on block A_1 , and therefore, since it is allowed to scan but a fixed number of squares, when it is centered

at the end of block A_1 it scans but a fixed amount of squares outside. Since the ends of the individual blocks govern the action the single head ought to take, and also cells of $A_{i+1} - A_i$ have to be transported through A_i for arbitrary i , we cannot have the physically present ends of all of blocks A_2, A_3, \dots, A_i in between the head centered on A_1 and a cell, to be transported, in $A_{i+1} - A_i$. So we want the blocks to move, in a sense, completely out of each other. That is, an arrangement as below in Fig. 2, where we denote the cells in $A_{i+1} - A_i$ as B_{i+1} , for all $i \geq 1$, and A_1 by B_1 . ($x \xRightarrow{*} y$ denotes that y occurs after x .)

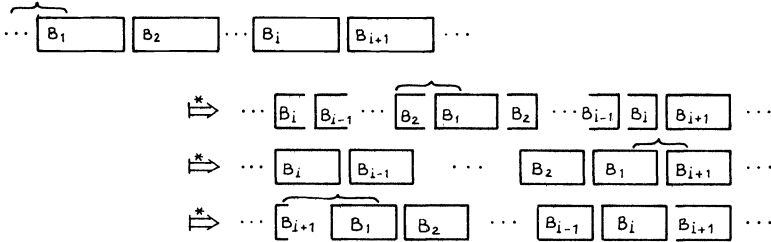


FIG. 2

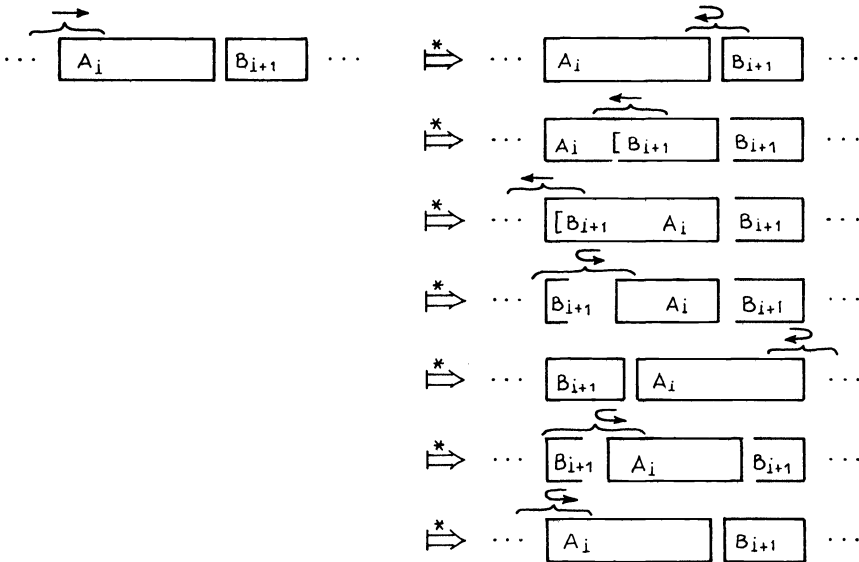


FIG. 3. The action of block $A_{i+1} = A_i \cup B_{i+1}$ with respect to blocks $B_j, j > i+1$ is not depicted.

In this manner we telescope the blocks, as it were, inwards and then outwards in the other direction, subsequently reversing the process. To achieve this behavior, we transport, for all $i \geq 1$, elements of block B_{i+1} through $A_i = \cup_{j=1}^i B_j$ while simultaneously shifting the cells of A_i to accommodate the transport. The motion of the head through A_i is governed by recursively moving B_{j+1} through A_j , for all $j, j \leq i$. Schematically, level i of the process is depicted in Fig. 3. When the head was at the ends of block $A_{i+1} = A_i \cup B_{i+1}$, it now could have picked up or deposited a cell outside, that is, of a block $B_j, j > i+1$. Assume that all blocks $B_i, i \geq 1$, have the same number of cells, say x . By a *full sweep* of the head over block A_i we shall mean the action the head has to perform, starting from one end of A_i , to pick up a cell of $B_j, j > i$, deposit

it on the other side of A_i , and finish at the same end of A_i from which it started. So basically a full sweep of the head is a traversal of block A_i from one end to the other end and back again. Let a full sweep of the head over A_i take at most $S(i)$ steps, $i \geq 1$. Then to transport all of B_{i+1} from one end of A_i to the other end, and back again, takes at most $cxS(i)$ steps for some constant c . Since this constitutes a full sweep of the head over block A_{i+1} , we have $S(i+1) \leq cxS(i)$, for all $i \geq 1$, and obviously $S(1) \leq cx$. So $S(i) \in 2^{O(i)}$.

In the formal construction below we set the block size to 2, and represent the loosely described block B_i by “[i] $_i$ ”, in the understanding that the two cells concerned are tagged with “[” and “]”. The subscripts on the tags are just there to aid the reader, but do not occur in the actual simulation. An element of block B_j in transport through block A_i , $j > i$, is identified by a curly bracket of the appropriate type. Thus each individual cell has permanently assigned to it a tag, consisting of either an opening or closing bracket, which may at different times be square or curly. Fig. 4 sketches a descriptive situation:

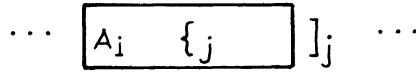


FIG. 4

After these preliminaries we formally define a one-head tape unit M . It is convenient to view the *instantaneous descriptions* (i.d.'s) (momentary snapshots of M 's tape contents and the head position) as one-way infinite linear arrays T , with “<” or “>” denoting the center of the head position. We tag the cells, containing elements of the array C of the previous section, with “[”, “]”, “{” or “}”. Below we display only these tags, since for the moment we are not interested in the cell contents. The identity of the underlying squares is not important, but the identity of the tagged cells is fixed, wherever they end up. For convenience of the reader we index the tagged cells (or rather the tags). The eventually defined machine, however, has no indexes associated with the cells, only one out of the four mentioned tags. The *initial i.d.* is, now focussing on the tags only,

$$T^0 = >[_1]_1[_2]_2 \cdots [_i]_i[_{i+1}]_{i+1} \cdots$$

We describe transformations of the array T in the form of six parametrized recursive functions, and four nonparametrized functions, each of two types. Each such function X will, for a unique subarray of T , rewrite this subarray by reordering its elements, specified by $X(\cdot)$: $\alpha \diamond \beta \mapsto \alpha' \diamond' \beta'$ with $\alpha, \alpha', \beta, \beta'$ being strings of (for clarity indexed) tags and $\diamond, \diamond' \in \{>, <\}$. A definite requirement for the process is that, at some time, it has to scan “[$_{i+2}$] $_{i+2}$ [$_{i+3}$] $_{i+3}$ ” for the first time. So we define, for all $i \geq 0$:

$$A(>, i): >[_1]_1[_2]_2 \cdots [_i]_i[_{i+1}]_{i+1} \mapsto [_{i+1}]_{i+1}[_i]_i[_{i-1}]_{i-1} \cdots [_1]_1 >]_{i+1}.$$

For symmetry we also define:

$$A(<, i): [_{i+1}]_{i+1}[_i]_i[_{i-1}]_{i-1} \cdots [_1]_1 < \mapsto [_{i+1}]_{i+1} <[_1]_1[_2]_2 \cdots [_i]_i]_{i+1}.$$

To abbreviate notation we shall henceforth denote shortly, for all $i \geq 0$,

$$\diamond[_i]_i \stackrel{\text{def}}{=} \diamond[_1]_1[_2]_2 \cdots [_i]_i$$

$$[_i]_i \diamond \stackrel{\text{def}}{=} [_i]_i[_{i-1}]_{i-1} \cdots [_1]_1 \diamond,$$

with

$$\diamond \in \{<, >\}, \text{ and } []_l = \varepsilon \text{ for } l = 0.$$

So for all $i \geq 0$:

$$A(>, i): >[]_l []_{i+1} []_{i+1} \mapsto []_{i+1} []_l > []_{i+1},$$

$$A(<, i): []_{i+1} []_{i+1} []_l < \mapsto []_{i+1} < []_l []_{i+1}.$$

By execution of $A(>, i)$ on the appropriate unique substring of T^0 we have therefore, using the same rewriting denotation as in the previous section:

$$T^0 \xrightarrow{A(>, i)} T^{t_i}$$

with

$$T^0 = > []_1 []_2 []_2 \cdots []_i []_{i+1} []_{i+1} []_{i+2} []_{i+2} []_{i+3} []_{i+3} \cdots$$

and

$$T^{t_i} = []_{i+1} []_i []_{i-1} []_{i-1} \cdots []_1 []_1 > []_{i+1} []_{i+2} []_{i+2} []_{i+3} []_{i+3} \cdots$$

where t_i is the number of steps it takes to execute the mapping $A(>, i)$, to be specified later, $i \geq 0$.

With the head scanning at least five squares right of the center position, indicated by “>”, the subarray “[]_{i+2} []_{i+2} []_{i+3} []_{i+3}” is scanned at time t_i , for all $i \geq 0$, while at time $t = 0$ the subarray “[]₁ []₂ []₂” is scanned.

DEFINITION. To achieve the required interchange of tagged cells, define the functions below. Recall that “ $\{ \}_j$ ” denotes the same cell as “ $\{ \}_j$ ”, only the attached tags have changed. Similarly for “ $\{ \}_j$ ” and “[]_j”. For all $i \geq 0$ and $j > i + 1$:

$$A(>, i): > []_l []_{i+1} x \mapsto []_{i+1} []_l > x \quad (x \neq [])$$

$$A(<, i): x []_{i+1} []_l < \mapsto x < []_l []_{i+1} \quad (x \neq [])$$

$$B(>, i): []_{i+1} []_l > []_j x \mapsto []_j < []_l []_{i+1} x \quad (x \neq [])$$

$$B(<, i): x []_j < []_l []_{i+1} \mapsto x []_{i+1} []_l > []_j \quad (x \neq [])$$

$$C(>, i): []_{i+1} []_l > []_j []_{j+1} \mapsto []_j \{ \}_{j+1} < []_l []_{i+1}$$

$$C(<, i): []_{j+1} []_j < []_l []_{i+1} \mapsto []_{i+1} []_l > []_{j+1} \{ \}_j$$

$$D(>, i): > []_l x \mapsto []_l > x \quad (x \neq [])$$

$$D(<, i): x []_l < \mapsto x < []_l \quad (x \neq [])$$

$$E(>, i): []_{i+1} []_l > []_{i+1} x \mapsto []_{i+1} []_{i+1} []_l > x \quad (x \neq [])$$

$$E(<, i): x []_{i+1} < []_l []_{i+1} \mapsto x < []_l []_{i+1} []_{i+1} \quad (x \neq [])$$

$$F(>, i): []_{i+1} []_l > []_{i+1} []_{i+2} x \mapsto []_{i+2} []_{i+1} []_{i+1} []_l > x \quad (x \neq [])$$

$$F(<, i): x []_{i+2} []_{i+1} < []_l []_{i+1} \mapsto x < []_l []_{i+1} []_{i+1} []_{i+2} \quad (x \neq [])$$

$$G(>): > \{ \}_j []_{i+1} \mapsto > []_i []_{i+1} \{ \}_j$$

$$G(<): []_{i+1} []_i \{ \}_j < \mapsto []_j []_{i+1} []_i <$$

$$H(>): > []_{j+1} \{ \}_j []_{i+1} \mapsto > []_i []_{i+1} []_{j+1} \{ \}_j$$

$$H(<): []_{i+1} []_i \{ \}_j []_{j+1} < \mapsto []_j \{ \}_{j+1} []_{i+1} []_i <$$

$$J(>): > \{ \}_{i+1} []_{i+1} \mapsto < []_i []_{i+1} []_{i+1}$$

$$J(<): []_{i+1} []_i []_{i+1} < \mapsto []_{i+1} []_{i+1} []_i >$$

$$K(>): >_{i+2} \{_{i+1}]_i]_{i+1} \mapsto <]_i [_{i+1}]_{i+1}]_{i+2}$$

$$K(<): [_{i+1} []_{i+1} \{_{i+2} < \mapsto [_{i+2} []_{i+1}]_{i+1} []_i >$$

The *parametrized* functions A through F set the basic pattern to transport tagged cells from one side of $[]_I$ to the other side. (The index j is always greater than $i+1$.) The *nonparametrized* functions G and H serve to move (linked pairs of) curly brackets through “[]” interfaces. If curly brackets are adjacent to a (linked) pair like “[]” then they have not yet reached their destination. If curly brackets are adjacent to a pair of square brackets of equal type, then they have reached their destination, and are fitted in place and changed back to square brackets, by the functions J and K . In the G and H functions, the index j is again greater than $i+1$. However, to make the point once more, the indexes are only put there to aid the reader. The intention of the described rewritings is that the arrays concerned consist of nonsubscripted brackets, each bracket viewed as tagging a particular cell. The rewriting reorders these tagged cells in the array, and possibly changes brackets from square ones to curly ones of the same type, or vice versa, as indicated in the indexed version above. Note that $A(>, i): Y \mapsto Y'$ and $A(<, i): Z \mapsto Z'$ are related by the fact that Z is the mirror image of Y and Z' is the mirror image of Y' . With *mirror image* we do not mean only the reverse, but the reverse with every constituent symbol changed to its mirror image, so “>” to “<”, “[” to “]”, “{” to “}”, “{” to “}” and “}” to “{”. Similarly for the other functions.

LEMMA 2. For all $i > 0$, the functions are related as follows:

$$a) A(>, i) = A(>, i-1); F(>, i-1)$$

$$A(<, i) = A(<, i-1); F(<, i-1)$$

$$b) B(>, i) = B(>, i-1); G(<); F(<, i-1)$$

$$B(<, i) = B(<, i-1); G(>); F(>, i-1)$$

$$c) C(>, i) = C(>, i-1); H(<); F(<, i-1)$$

$$C(<, i) = C(<, i-1); H(>); F(>, i-1)$$

$$d) D(>, i) = A(>, i-1); E(>, i-1)$$

$$D(<, i) = A(<, i-1); E(<, i-1)$$

$$e) E(>, i) = B(>, i-1); J(<); D(>, i-1); E(>, i-1)$$

$$E(<, i) = B(<, i-1); J(>); D(<, i-1); E(<, i-1)$$

$$f) F(>, i) = C(>, i-1); K(<); D(>, i-1); E(>, i-1)$$

$$F(<, i) = C(<, i-1); K(>); D(<, i-1); E(<, i-1)$$

that is, six parametrized functions recursively calling each other. (Since $D(>, 0)$ and $D(<, 0)$ are “no operation”’s which do not change anything we leave them out, cf. below.)

Proof. For a) through f) we prove one equality each; the other one is symmetric.

For all $i > 0$, with $[]_{I-1} = \varepsilon$ for $i = 1$ by definition:

a) For $x \neq []$:

$$> []_I []_{i+1} x = > []_{I-1} []_i []_{i+1} x$$

$$\xrightarrow{A(>, i-1)} []_i []_{I-1} >]_i []_{i+1} x$$

$$\xrightarrow{F(>, i-1)} []_{i+1} []_i []_{I-1} > x = []_{i+1} []_I > x;$$

b) For $x \neq]$:

$$\begin{aligned}
 x[j < []_I]_{i+1} &= x[j < [_{I-1}]_{I-1} [i]]_{i+1} \\
 &\xrightarrow{B(<,i-1)} x[i [_{I-1}]_{I-1} > \{j\}]_{i+1} \\
 &\xrightarrow{G(>)} x[i [_{I-1}]_{I-1} >]_{i+1} \{j\} \\
 &\xrightarrow{F(>,i-1)} x[i+1 [i] [_{I-1}]_{I-1} > \{j\} \\
 &= x[i+1 []_I > \{j\};
 \end{aligned}$$

c)

$$\begin{aligned}
]_{j+1} [j < []_I]_{i+1} &=]_{j+1} [j < [_{I-1}]_{I-1} [i]]_{i+1} \\
 &\xrightarrow{C(<,i-1)} [i [_{I-1}]_{I-1} > \}_{j+1} \{j\}]_{i+1} \\
 &\xrightarrow{H(>)} [i [_{I-1}]_{I-1} >]_{i+1} \}_{j+1} \{j\} \\
 &\xrightarrow{F(>,i-1)} [i+1 [i] [_{I-1}]_{I-1} > \}_{j+1} \{j\} \\
 &= [i+1 []_I > \}_{j+1} \{j\};
 \end{aligned}$$

d) For $x \neq [$:

$$\begin{aligned}
 > []_I x &= > [_{I-1}]_{I-1} [i] x \\
 &\xrightarrow{A(>,i-1)} [i [_{I-1}]_{I-1} >] x \\
 &\xrightarrow{E(>,i-1)} [i] [i [_{I-1}]_{I-1} > x \\
 &= []_I > x;
 \end{aligned}$$

e) For $x \neq]$:

$$\begin{aligned}
 x[i+1 < []_I]_{i+1} &= x[i+1 < [_{I-1}]_{I-1} [i]]_{i+1} \\
 &\xrightarrow{B(<,i-1)} x[i [_{I-1}]_{I-1} > \{i+1\}]_{i+1} \\
 &\xrightarrow{J(>)} x[i [_{I-1}]_{I-1} <]_{i+1} [i+1]_{i+1} \\
 &\xrightarrow{D(<,i-1)} x[i < [_{I-1}]_{I-1}]_{i+1} [i+1]_{i+1} \\
 &\xrightarrow{E(<,i-1)} x < [_{I-1}]_{I-1} [i] [i+1]_{i+1} \\
 &= x < []_I [i+1]_{i+1};
 \end{aligned}$$

f) For $x \neq]$:

$$\begin{aligned}
 x]_{i+2} [i+1 < []_I]_{i+1} &= x]_{i+2} [i+1 < [_{I-1}]_{I-1} [i]]_{i+1} \\
 &\xrightarrow{C(<,i-1)} x]_{i+2} [i [_{I-1}]_{I-1} > \}_{i+2} \{i+1\}]_{i+1}
 \end{aligned}$$

$$\begin{aligned}
& \xrightarrow{K(>)} x[i \llbracket_{I-1} \rrbracket_{I-1} \langle i \llbracket_{i+1} \rrbracket_{i+1} \rrbracket_{i+2} \\
& \xrightarrow{D(<, i-1)} x[i \langle \llbracket_{I-1} \rrbracket_{I-1} i \llbracket_{i+1} \rrbracket_{i+1} \rrbracket_{i+2} \\
& \xrightarrow{E(<, i-1)} x \langle \llbracket_{I-1} \rrbracket_{I-1} \llbracket_i \rrbracket_i \llbracket_{i+1} \rrbracket_{i+1} \rrbracket_{i+2} \\
& = x \langle \llbracket_I \rrbracket_I \llbracket_{i+1} \rrbracket_{i+1} \rrbracket_{i+2}. \quad \square
\end{aligned}$$

The mappings $D(>, 0): > x \mapsto > x (x \neq \square)$ and $D(<, 0): x \langle \mapsto x \langle (x \neq \square)$ are “no operation” or “skip” instructions. Deleting them henceforth in the expansion rules of Lemma 2e) and 2f), for $i = 1$, those become:

$$\text{ad Lemma 2a) } E(>, 1) = B(>, 0); J(<); E(>, 0)$$

$$E(<, 1) = B(<, 0); J(>); E(<, 0)$$

$$\text{ad Lemma 2f) } F(>, 1) = C(>, 0); K(<); E(>, 0)$$

$$F(<, 1) = C(<, 0); K(>); E(<, 0).$$

A *level i expansion* of a function $X(>, j)$ or $X(<, j)$, $j \geq i \geq 0$, results from expanding that parametrized function with parameter j into a sequence of parameter i functions and nonparametrized functions, according to Lemma 2 (with the “no operation”’s $D(>, 0)$ and $D(<, 0)$ left out in case $i = 0$). So if $Y_1^{(i)}; Y_2^{(i)}; \dots; Y_n^{(i)}$ is a level i expansion of $X(\cdot)$ then $X(\cdot) = Y_1^{(i)}; Y_2^{(i)}; \dots; Y_n^{(i)}$ with $Y_l^{(i)} \in \{A(>, i), A(<, i), B(>, i), B(<, i), \dots, F(<, i), G(>), G(<), \dots, K(<)\} - \{D(>, 0), D(<, 0)\}$, $1 \leq l \leq n$. We extend the concept in the obvious way to sequences of functions $X_1(\diamond_1, j_1); X_2(\diamond_2, j_2); \dots; X_m(\diamond_m, j_m)$, $j_1, j_2, \dots, j_m \geq i$ and $\diamond_1, \diamond_2, \dots, \diamond_m \in \{\langle, \rangle\}$. For example, the level 0 expansion of $A(>, 3)$ is found by way of the level 2 and level 1 expansions:

$$\begin{aligned}
A(>, 3) &= A(>, 2); F(>, 2) \\
&= A(>, 1); F(>, 1); C(>, 1); K(<); D(>, 1); E(>, 1) \\
&= A(>, 0); F(>, 0); C(>, 0); K(<); E(>, 0); C(>, 0); H(<); \\
&F(<, 0); K(<); A(>, 0); E(>, 0); B(>, 0); J(<); E(>, 0).
\end{aligned}$$

The atomic mappings of the level 0 expansions of the parametrized functions are called the *local rewriting rules*, and govern the switching of the tagged cells, in the squares scanned, by the basic steps of the oblivious one-head tape unit M . Note that a fat head covering four squares left and four squares right of the displayed center “ $>$ ” or “ $<$ ” suffices to execute these atomic mappings. Below we use superscripts to distinguish the identity of the various tagged cells before and after rewriting.

Local rewriting rules:

$$G(>): > \{^1\}^2 \{^3 \mapsto > \}^2 \{^3 \{^1$$

$$G(<): \{^1 \{^2 \}^3 \{^3 \langle \mapsto \}^3 \{^1 \{^2 \langle$$

$$H(>): > \{^1 \{^2 \}^3 \{^4 \mapsto > \}^3 \{^4 \}^1 \{^2$$

$$H(<): \{^1 \{^2 \}^3 \{^4 \langle \mapsto \}^3 \{^4 \}^1 \{^2 \langle$$

$$J(>): > \{^1 \{^2 \}^3 \mapsto \langle \}^2 \{^1 \}^3$$

$$J(<): \{^1 \{^2 \}^3 \langle \mapsto \{^1 \}^3 \{^2 \rangle$$

$$K(>): >\}^1\{^2\}^3]^4 \mapsto <]^3\{^2\}^4]^1$$

$$K(<): [^1\{^2\}^3\{^4< \mapsto [^4\{^1\}^3\{^2>$$

$$A(>, 0): >[\mapsto [>$$

$$A(<, 0):]< \mapsto <]$$

$$B(>, 0):]^1>]^2x \mapsto \}^2<]^1x \quad (x \neq \square)$$

$$B(<, 0): x[^1<[^2 \mapsto x[^2>\{^1 \quad (x \neq])$$

$$C(>, 0):]^1>]^2\{^3 \mapsto \}^2\{^3<]^1$$

$$C(<, 0):]^1\{^2<[^3 \mapsto [^3>\}^1\{^2$$

$$E(>, 0): [^1>]^2x \mapsto [^1]^2>x \quad (x \neq \square)$$

$$E(<, 0): x[^1<]^2 \mapsto x<[^1]^2 \quad (x \neq])$$

$$F(>, 0): [^1>]^2\{^3 \mapsto [^3\{^1\}^2>$$

$$F(<, 0):]^1\{^2<]^3 \mapsto <[^2\}^3]^1$$

The only use of the context symbols x in the definitions of $A(>, i)$, $A(<, i)$, $F(>, i)$ and $F(<, i)$ was to force a unique expansion into functions with parameter $j, j < i$, according to Lemma 2. Since $A(<, 0)$, $A(>, 0)$, $F(<, 0)$ and $F(>, 0)$ are atomic indivisible actions, because the local rewriting rules shall not be decomposed any further, we do not need these context symbols at the lowest level.

In the sequel it is useful to talk about *well formed* arrays, that is, the set of arrays from which the consecutive i.d.'s of M are taken.

(i) T^0 is a well formed array.

(ii) If T is a well formed array and $X(\cdot)$ is any local rewriting rule, with the dot standing for any appropriate argument, such that $T \xrightarrow{X(\cdot)} T'$, $T' \neq \emptyset$, then T' is a well formed array.

(iii) No array is well formed except by (i) and (ii).

Since no mapping either deletes or multiplies a headmarker, i.e., “<” or “>”, all well formed arrays contain a single headmarker. By the mutual exclusion of the subarrays they rewrite, if a well formed array T is rewritten to $T' \neq \emptyset$ by a local rewriting rule, then T is rewritten to \emptyset by all other local rewriting rules. We now show that a well formed array T is always rewritten by some local rewriting rule to a another well formed array, which rewriting rule and array are therefore unique.

Earlier, we observed that, for all $i \geq 0$,

$$T^0 \xrightarrow{A(>, i)} T^i.$$

If $Y_1^{(0)}$; $Y_2^{(0)}$; $Y_3^{(0)}$; \dots ; $Y_n^{(0)} = A(>, i)$ is the level 0 expansion of $A(>, i)$ then, by Lemma 2, there exist well formed arrays $T_0^{(0)}$, $T_1^{(0)}$, $T_2^{(0)}$, \dots , $T_n^{(0)}$, $T_0^{(0)} = T^0$ and $T_n^{(0)} = T^i$, such that

$$T_{j-1}^{(0)} \xrightarrow{Y_j^{(0)}} T_j^{(0)}$$

for all j , $1 \leq j \leq n$. By the uniqueness of application of local rewriting rules it follows that

$$T_{j-1}^{(0)} \xrightarrow{X} \emptyset$$

for all j , $1 \leq j \leq n$, $X \neq Y_j^{(0)}$ and X is a local rewriting rule. Hence each well formed array in the sequence $T_0^{(0)}, T_1^{(0)}, \dots, T_{n-1}^{(0)}$ has exactly one local rewriting rule which is applicable to it, and the application of this local rewriting rule yields exactly one next well formed array.

By Lemma 2a) we have $A(>, i) = A(>, i-1); F(>, i-1)$ for all $i > 0$, which leads to

$$A(>, i) = A(>, 0); F(>, 0); F(>, 1); \dots; F(>, i-1)$$

with

$$T^0 \xrightarrow{A(>, 0)} T^{t_0}$$

and

$$T^{t_j} \xrightarrow{F(>, j)} T^{t_{j+1}}$$

for all j , $0 \leq j < i$. Define $A(>, \infty)$ by

$$A(>, \infty) = \lim_{i \rightarrow \infty} A(i) = A(>, 0); F(>, 0); F(>, 1); \dots; F(>, i); \dots$$

and the level 0 expansion of $A(>, \infty)$ as the infinite, or unbounded, sequence of local rewriting rules resulting from the level 0 expansions of the constituent functions $F(>, i)$, $i > 0$, above. So

$$\begin{aligned} A(>, \infty) &= Y_1^{(0)}; Y_2^{(0)}; \dots; Y_i^{(0)}; \dots \\ &= A(>, 0); F(>, 0); C(>, 0); K(<); E(>, 0); \dots \end{aligned}$$

and there exists an infinite sequence of well formed arrays $T_0^{(0)}, T_1^{(0)}, \dots, T_i^{(0)}, \dots, T_0^{(0)} = T^0$, such that for all $j \geq 1$

$$T_{j-1}^{(0)} \xrightarrow{Y_j^{(0)}} T_j^{(0)}$$

and for no local rewriting rule $X \neq Y_j^{(0)}$ and $T \neq \emptyset$

$$T_{j-1}^{(0)} \xrightarrow{X} T,$$

i.e., $Y_j^{(0)}$ is the only local rewriting rule applicable to $T_{j-1}^{(0)}$. Consequently, a machine which wants to execute the sequence of local rewritings of the level 0 expansion of $A(>, \infty)$, starting with i.d. T^0 , needs only to select the single local rewriting rule $Y_j^{(0)}$, applicable to the current $T_{j-1}^{(0)}$, by considering the length 9 subarray of $T_{j-1}^{(0)}$ with the current headmarker in the center, to obtain the next $T_j^{(0)}$, $j \geq 1$. From the expansions in Lemma 2 we see that a nonparametrized function of G, H, J, K is always followed by a parametrized function from A, B, C, E, F in the level 0 expansion of $A(>, \infty)$. In a single step of M we shall first execute a local rewriting according to G, H, J , or K , if possible, and then execute a local rewriting according to A, B, C, E or F , which by the above is always possible, starting with initial i.d. T^0 . So the oblivious one-head tape unit M at each step shall examine the squares around the headmarker, and switches tagged cells and head position amongst the scanned squares according to the only local rewriting rules applicable. Fig. 5 shows an initial segment of the sequence of well formed arrays $T_0^{(0)}, T_1^{(0)}, \dots, T_i^{(0)}, \dots$ produced by the successive execution of the local rewriting rules in the level 0 expansion of $A(>, \infty)$ using the simple procedure SWITCH below.

Step 2. Examine the length 9 subarray, centered on the headmarker, of array T from step 1, and switch tagged cells and headmarker position according to the single local rewriting rule from the A, B, C, E, F rules which is applicable. The resultant well formed array is the next i.d.

LEMMA 3. *Starting from the initial i.d. T^0 , a one-head tape unit M , executing SWITCH at each single step, executes exactly the local rewriting rule sequence of the level 0 expansion of $A(>, \infty)$. For each $t > 0$, in the first t steps M executes this sequence up to and including the t th occurrence of a local rewriting rule of the A, B, C, E or F type.*

The goal of introducing the present bracket manipulator was to scan the subarray “[i] [$i+1$] [$i+1$]” at least once in each interval of $2^{\Theta(i)}$ steps, $i \geq 0$. We can express precisely what the t th i.d. T^t is. T^0 is the initial array at time $t=0$, and T^t results from an execution of SWITCH on T^{t-1} , for all $t > 0$. According to Lemma 3, t equals the number of occurrences of A, B, C, E, F -type local rewriting rules executed. We need to recognize “[i] [$i+1$] [$i+1$]” as being the correct sequence of cells, which, since the cells are tagged with nonindexed brackets in the manipulator proper, cannot go by way of identifying the individual cells. For this purpose, the next lemma establishes a *topology* for the well formed arrays. Before proceeding, we review a few facts about well formed arrays which are pertinent to the proof of that lemma. By definition, and the discussion preceding Lemma 3, the set of well formed arrays equals the set $\{T_j^{(0)} | j \geq 0\}$ defined by the level 0 expansion of $A(>, \infty)$.

$$A(>, \infty) = Y_1^{(0)}; Y_2^{(0)}; \dots; Y_i^{(0)}; \dots$$

and for all $i \geq 1$

$$T_{i-1}^{(0)} \xrightarrow{Y_i^{(0)}} T_i^{(0)} \quad \text{with } T_0^{(0)} = T^0.$$

By the definition of the initial array T^0 , and those of the various procedures, each well formed array contains exactly one symbol from $\{<, >\}$ and, for each $i \geq 1$, exactly one symbol from $\{[i, \{i}\}$ and exactly one symbol from $\{[i, \}i\}$. Recall that the indices are not really there but serve to identify the individual cells for the reader by distinguishing between the individual attached tags.

If a well formed array T contains a pair of adjacent brackets “[j] $_k$ ” then $j = k$; if it contains “[j] $_k$ ” then $k = j+1$ in case the headmarker is to the left and $j = k+1$ in case the headmarker is to the right. More precisely:

LEMMA 4. *Let T be a well formed array, and let $\diamond \in \{<, >\}$ and $\alpha, \beta, \gamma \in \{[i, \{i}\}$. Then:*

- (i) $T = \alpha \diamond \beta [j]_k \gamma \Rightarrow k = j$;
 $T = \alpha [k]_j \beta \diamond \gamma \Rightarrow k = j$;
- (ii) $T = \alpha \diamond \beta]_j [k \gamma \Rightarrow k = j+1$;
 $T = \alpha]_k [j \beta \diamond \gamma \Rightarrow k = j+1$.

Proof. We basically prove the lemma by induction on the sequence of well formed arrays $T_j^{(0)}$, as defined by the level 0 expansion of $A(>, \infty)$, $j \geq 0$. To do so, we consider the initial segments $T_j^{(0)}[0:2(i+1)]$, $j \geq 0$ and $i \geq 0$, in isolation and show by the claim below that they internally satisfy the lemma. Viz., in executing the level 0 expansion of $A(>, i)$ to obtain T^i from T^0 the elements of the sequence of subarrays $T_0^{(0)}[0:2(i+1)]$, $T_1^{(0)}[0:2(i+1)]$, \dots , $T_{a(i)}^{(0)}[0:2(i+1)]$, with $T_0^{(0)} = T^0$ and $T_{a(i)}^{(0)} = T^i$, will be shown to internally satisfy the lemma. Since during the execution of $A(>, i)$ the final segment

$T[2(i+1):\infty]$ is not changed at all, and T^0 satisfies the lemma, the elements of the sequence of subarrays $T_0^{(0)}[2(i+1):\infty]$, $T_1^{(0)}[2(i+1):\infty]$, \dots , $T_{a(i)}^{(0)}[2(i+1):\infty]$ do internally satisfy the lemma. Because we have an overlap of one symbol between $T_j^{(0)}[0:2(i+1)]$ and $T_j^{(0)}[2(i+1):\infty]$ for all j , $0 \leq j \leq a(i)$ with $T_0^{(0)} = T^0$ and $T_{a(i)}^{(0)} = T^i$, we can conclude that each well formed array $T_j^{(0)}$, $0 \leq j \leq a(i)$, satisfies the lemma. Taking the limit for $i \rightarrow \infty$, that is, considering $A(>, \infty)$, it follows that the lemma holds for all well formed arrays.

CLAIM. *Let, for all $i \geq 0$, $X \in \{A, B, C, D, E, F\}$, $\diamond \in \{<, >\}$ and T, T' be well formed arrays such that $X(\diamond, i): T[p:q] \mapsto T'[p:q]$, for some $p, q \geq 0$ and $T \xrightarrow{X(\diamond, i)} T'$, and let $Y_{l+1}^{(0)}$; $Y_{l+2}^{(0)}$; \dots ; $Y_{l+x(i)}^{(0)}$ be the level 0 expansion of $X(\diamond, i)$ with $T_{l+j-1}^{(0)} \xrightarrow{Y_{l+j}^{(0)}} T_{l+j}^{(0)}$ for all j , $1 \leq j \leq x(i)$ with $T_l^{(0)} = T$ and $T_{l+x(i)}^{(0)} = T'$. Then, for all j , $l \leq j \leq l+x(i)$, $T_j^{(0)}[p:q]$ internally satisfies the lemma.*

Proof of claim. **Base case $i=0$.** Since for $i=0$ the procedures are essentially but the local rewriting rules, we only have to verify that in the definitions of the various functions the subarrays left and right of the arrow internally satisfy the lemma. Note that $[]_l \stackrel{\text{def}}{=} \varepsilon$ for $i=0$.

Induction. Assume, by way of contradiction, that for some $X(\diamond, i)$, with $X \in \{A, B, C, D, E, F\}$ and $\diamond \in \{<, >\}$ and $i > 0$, the claim does not hold. But in the execution of the level $i-1$ expansions of six of these functions with parameter i in the proof of Lemma 2, the other six cases being symmetrical, the displayed subarrays all satisfy the claim. Hence it must follow that a nondepicted subarray arising in the execution of the level 0 expansion of some $X'(\diamond', i-1)$, $X' \in \{A, B, C, D, E, F\}$ and $\diamond' \in \{<, >\}$, violates the claim. Regressing in this fashion all the way down to $i=0$, we contradict the established base case, and the claim is proven. \square

By the discussion preceding the claim we have established the lemma. \square

LEMMA 5. *Let T be a well formed array and let $\diamond \in \{<, >\}$ and $\alpha, \beta \in \{[,], \{, \}\}^*$. Then*

$$(T = \alpha \diamond []_j \beta \text{ or } T = \alpha []_k \diamond \beta) \Rightarrow (k = j = 1).$$

Proof. That $k = j$ follows already from Lemma 4. Considering the level 1 expansion of $A(>, \infty)$

$$A(>, \infty) = Y_1^{(1)}; Y_2^{(1)}; \dots; Y_j^{(1)}; \dots$$

and

$$T_{j-1}^{(1)} \xrightarrow{Y_j^{(1)}} T_j^{(1)} \text{ with } T_0^{(1)} = T^0,$$

we observe that it follows from the definitions of the various procedures that, for all well formed arrays $T_j^{(1)}$, $j \geq 0$, the lemma holds. Expanding each A, B, C, D, E, F function with parameter 1 to level 0, and examining the intermediate well formed arrays $T_j^{(0)} \neq T_j^{(1)}$, $j, j' \geq 0$, yields the lemma. \square

Lemma 4 and Lemma 5 show that a certain topological connectedness between the indexed brackets is preserved throughout the array at all times, and that, in particular, in each well formed array $\alpha \diamond \beta []_k []_l \gamma$ holds $k = j$ and $l = m = j+1$. So whenever there occurs a length four subarray “[] []” right of the headmarker the tagged cells concerned are in the correct consecutive left to right order. Without further proof we give a more exhaustive characterization of the topology. Let T be a well formed array. Then, for each $i \geq 1$, T satisfies precisely one of the following forms.

For $i = 1$:

$$\begin{aligned}
& \alpha []_1 \diamond \beta \\
& \alpha \diamond []_1 \beta \\
& \alpha [] \diamond []_1 \beta \\
& \alpha []_1 > \{ \}_j []_2 []_2 \cdots []_{j-1} []_{j-1} \}_j \beta & (\text{some } j \geq 2) \\
& \alpha []_1 > \}_{j+1} \{ \}_j []_1 []_2 []_2 \cdots []_{j-1} []_{j-1} \}_j \beta & (\text{some } j \geq 2) \\
& \alpha []_j []_{j-1} []_{j-1} []_{j-2} []_{j-2} \cdots []_2 []_2 []_1 \}_j < []_1 \beta & (\text{some } j \geq 2) \\
& \alpha []_j []_{j-1} []_{j-1} []_{j-2} []_{j-2} \cdots []_2 []_2 []_1 \}_j \{ \}_{j+1} < []_1 \beta & (\text{some } j \geq 2),
\end{aligned}$$

with the obvious modification for $j = 2$.

For each $i > 1$:

$$\begin{aligned}
& \alpha []_i []_{i-1} \beta \diamond \gamma \\
& \alpha \diamond \beta []_{i-1} []_i \gamma \\
& \alpha []_i \beta \diamond \gamma \{ \}_{i-1} []_j []_{j+1} []_{j+1} []_{j+2} []_{j+2} \cdots []_{i-2} []_{i-2} []_{i-1} \delta & (\text{some } j < i-1) \\
& \alpha \diamond \beta \{ \}_j []_{j+1} []_{j+1} []_{j+2} []_{j+2} \cdots []_{i-1} []_{i-1} \}_i \gamma & (\text{some } j \leq i-1) \\
& \alpha []_{i-1} []_{i-2} []_{i-2} []_{i-3} []_{i-3} \cdots []_{j+1} []_{j+1} []_j \}_{i-1} \{ \}_i \beta \diamond \gamma \}_i \delta & (\text{some } j < i-1) \\
& \alpha []_i []_{i-1} []_{i-1} []_{i-2} []_{i-2} \cdots []_{j+1} []_{j+1} []_j \}_i \beta \diamond \gamma & (\text{some } j \leq i-1),
\end{aligned}$$

with the obvious modification for $i-3 \leq j \leq i-1$. Here \diamond can be either “<” or “>” and $\alpha, \beta, \gamma, \delta \in \{ [,], \{, \} \}^*$. Considering the fact that

$$T^0 = > []_1 []_1 []_2 []_2 \cdots []_i []_{i+1} []_{i+1} \cdots$$

and that, by definition, for $i \geq 0$ and $t \leq t_i$,

$$T^t = \alpha \diamond \beta []_{i+1} []_{i+2} []_{i+2} \cdots []_{i+j} []_{i+j} []_{i+j+1} []_{i+j+1} \cdots,$$

$j \geq 3$, the formats express, but for the choice of \diamond as “<” or “>”, the format each well formed array T^t can have, by applying in sequence the requirements for $i+1, i, \dots, 1$. According to Lemma 4, whenever we scan a subarray “[] []” right of the headmarker, we know for sure that this is the subarray “[]_i []_{i+1} []_{i+1}” for some $i \geq 1$. In the next lemma we give an upper bound on the number of steps, that is, executions of SWITCH, in between scanning “[]_i []_{i+1} []_{i+1}” right of the headmarker, for all $i \geq 1$. To express the timing we consider expansions of $A(>, \infty)$ of level $i, i \geq 1$:

$$A(>, \infty) = Y_1^{(i)}; Y_2^{(i)}; \dots; Y_j^{(i)}; \dots$$

and define for all $j \geq 1$

$$T_{j-1}^{(i)} \xrightarrow{Y_j^{(i)}} T_j^{(i)} \text{ with } T_0^{(i)} = T^0.$$

The level 0 expansion of $Y_j^{(i)} = X(\diamond, i)$, with $X \in \{A, B, C, D, E, F\}$ and $\diamond \in \{<, >\}$, is fixed and, but for the headmarker arguments, is the same whether $\diamond = <$ or $\diamond = >$. Thus, by Lemma 3, the number of steps of M to execute $X(\diamond, i)$ equals the number of occurrences of A, B, C, E, F local rewriting rules in its level 0 expansion, and does not depend on the orientation of the headmarker \diamond , or the position j in the level i expansion of $A(>, \infty)$ where $Y_j^{(i)}$ occurs. We denote the number of steps, used by M , to execute $X(\diamond, i)$, by $T_X(i)$.

LEMMA 6. *There exists a function $S: \mathbb{N} \rightarrow \mathbb{N}$ such that for each $t \geq 0$ there is a $t' > t$ such that for some $\alpha, \alpha', \beta, \beta' \in \{[,], \{, \}\}^*$ and $\diamond, \diamond' \in \{<, >\}$*

$$(i) \quad T^t = \alpha \diamond []_1 \beta \Rightarrow T^{t'} = \alpha' \diamond' []_1 \beta' \text{ and } t' - t \leq S(1).$$

$$(ii) \quad T^t = \alpha \diamond []_1 []_2 \beta \Rightarrow T^{t'} = \alpha' \diamond' []_1 []_2 \beta' \text{ and } t' - t \leq S(2).$$

$$(iii) \quad \text{For all } i > 2 \text{ and } x \in \{\varepsilon, \{, \}\} \text{ there is a } x' \in \{\varepsilon, \{, \}\} \text{ such that: } T^t = \alpha \diamond x]_{i-2} []_{i-1} []_i \beta \Rightarrow T^{t'} = \alpha' \diamond' x']_{i-2} []_{i-1} []_i \beta' \text{ and } t' - t \leq S(i).$$

Moreover, $S(i) = 2T_A(i) + T_F(i)$, for all $i \geq 1$, is such a function.

Proof. Consider the level i expansion of $A(>, \infty)$, $i \geq 1$,

$$A(>, \infty) = Y_1^{(i)}; Y_2^{(i)}; \dots; Y_j^{(i)}; \dots$$

and

$$T_{j-1}^{(i)} \xrightarrow{Y_j^{(i)}} T_j^{(i)} \quad \text{with } T_0^{(i)} = T^0.$$

Then $T_j^{(i)}$ is of the form $\alpha \diamond []_l \beta$ or $\alpha []_l \diamond \beta$, for all $j \geq 0$. All such $T_j^{(i)}$'s, with $Y_j^{(i)}$ not G, H, J or K local rewriting rules, are i.d.'s of M . We restrict attention to the particular subsequence $T_{j_0}^{(i)}, T_{j_1}^{(i)}, \dots, T_{j_k}^{(i)}, \dots$ for which $T_{j_k}^{(i)}$ is of the form $\alpha < []_l \beta$ for all $k > 0$ and $T_{j_0}^{(i)} = T^0$. For each $k \geq 0$ there exists a sequence $Y_{j_{k+1}}^{(i)}; Y_{j_{k+2}}^{(i)}; \dots; Y_{j_{k+1}}^{(i)}$ such that

$$(1) \quad T_{j_k}^{(i)} \xrightarrow{Y_{j_{k+1}}^{(i)}; Y_{j_{k+2}}^{(i)}; \dots; Y_{j_{k+1}}^{(i)}} T_{j_{k+1}}^{(i)}.$$

By the use of the recursive expressions in Lemma 2 we can determine all such sequences. Subsequently, we have to determine which such sequences take the most steps to execute. So we first determine $T_X(i)$ for all $X \in \{A, B, C, D, E, F\}$. It follows from Lemma 3 that $T_X(i)$ equals the number of occurrences of A, B, C, E, F procedures in its level 0 expansion. We see from Lemma 2 that:

$$T_A(i) = T_A(i-1) + T_F(i-1),$$

$$T_B(i) = T_B(i-1) + T_F(i-1),$$

$$T_C(i) = T_C(i-1) + T_F(i-1),$$

$$T_D(i) = T_A(i-1) + T_E(i-1),$$

$$T_E(i) = T_B(i-1) + T_D(i-1) + T_E(i-1),$$

$$T_F(i) = T_C(i-1) + T_D(i-1) + T_E(i-1),$$

and $T_A(0) = T_B(0) = T_C(0) = T_E(0) = T_F(0) = 1$ and $T_D(0) = 0$. For this system of recurrence equations with initial values we find $T_A(i) = T_B(i) = T_C(i)$, for all $i \geq 0$, and consequently $T_E(i) = T_F(i)$, for all $i \geq 0$, which in its turn yields $T_D(i) = T_A(i)$, for all $i \geq 1$. Hence for all $i \geq 1$:

$$(2) \quad T_E(i) = T_F(i) \geq T_A(i) = T_B(i) = T_C(i) = T_D(i).$$

Now let $Y_{j_{k+1}}^{(i)}; Y_{j_{k+2}}^{(i)}; \dots; Y_{j_{k+1}}^{(i)}$ be a sequence of functions as in (1). Erasing the G, H, J and K procedures (because they do not contribute to the number of steps it takes to execute this sequence, by Lemma 3) and replacing all E 's by F 's and all B 's, C 's and D 's by A 's (because they take the same number of steps for $i \geq 1$) the resulting sequences are $F(<, i)$, $A(<, i)$; $A(>, i)$ and $A(<, i)$; $F(>, i)$; $A(<, i)$. So for $i = 1, 2$, $S(i) = 2T_A(i) + T_F(i)$ satisfies the lemma. For $i > 2$ we note that, for all $k \geq 0$ (with

the obvious modification for $k=0$),

$$T_{jk}^{(i)} = \alpha < [_{I-3}]_{I-3} [_{i-2}]_{i-2} [_{i-1}]_{i-1} [_{i}]_i \beta$$

$$\stackrel{Z_i}{\Longrightarrow} \alpha' [_{i-2}]_{i-2} [_{I-3}]_{I-3} > x]_{i-2} [_{i-1}]_{i-1} [_{i}]_i \beta$$

with Z_i and x one of the following:

$$Z_i \in \{J(<); A(>, i-3), K(<); A(>, i-3)\} \quad \text{and} \quad x = \varepsilon;$$

$$Z_i = B(<, i-3) \quad \text{and} \quad x = \{;$$

$$Z_i \in \{C(<, i-3), G(<); C(<, i-3), H(<); C(<, i-3)\} \quad \text{and} \quad x = \}\{.$$

In all cases, the execution time of Z_i is $T_A(i-3)$, which shows that $S(i) = 2T_A(i) + T_F(i)$ satisfies the lemma for all $i > 2$ too. \square

COROLLARY. *Let $S: \mathbb{N} \rightarrow \mathbb{N}$ be defined by $S(i) = 2T_A(i) + T_F(i)$, for all $i \geq 1$. Then:*

- (i) *For each $t \geq 0$ there exists a t' , $t < t' \leq t + S(1)$, such that the t' th i.d. of M has the form $T' = \alpha < [_{1}]_1 \beta$ for some $\alpha, \beta \in \{[,], \{, \}\}^*$.*
- (ii) *For each $t \geq 0$ there exists a t' , $t < t' \leq t + S(2)$, such that the t' th i.d. of M has the form $T' = \alpha < [_{1}]_1 [_{2}]_2 \beta$ for some $\alpha, \beta \in \{[,], \{, \}\}^*$.*
- (iii) *For each $i > 2$ and $t \geq 0$ there exists a t' , $t < t' \leq t + S(i)$, such that the t' th i.d. of M has the form $T' = \alpha > x]_{i-2} [_{i-1}]_{i-1} [_{i}]_i \beta$ for some $\alpha, \beta \in \{[,], \{, \}\}^*$ and $x \in \{\varepsilon, \{, \}\}$.*

It remains to determine S analytically.

$$\text{LEMMA 7. } S(i) = (1 + \sqrt{2})^{i+1} + (1 - \sqrt{2})^{i+1}.$$

Proof. From the system of recurrence equations, and the values for $i=0$, in the proof of Lemma 6, follows:

$$T_A(i) = 2T_A(i-1) + T_A(i-2) \quad \text{for } i > 2.$$

The solution for this homogeneous equation is of the form $T_A(i) = ax_1^i + bx_2^i$, where $x_{1,2}$ are the roots of $x^2 - 2x - 1 = 0$ and a and b follow from $T_A(1) = 2$ and $T_A(2) = 4$. So $x_{1,2} = 1 \pm \sqrt{2}$ and

$$a(1 + \sqrt{2}) + b(1 - \sqrt{2}) = 2, \quad a(1 + \sqrt{2})^2 + b(1 - \sqrt{2})^2 = 4$$

yielding $a = 1/\sqrt{2}$ and $b = -1/\sqrt{2}$. Hence

$$T_A(i) = \frac{1}{\sqrt{2}} (1 + \sqrt{2})^i - \frac{1}{\sqrt{2}} (1 - \sqrt{2})^i, \quad i \geq 1.$$

From the system of recurrence equations, and the identities amongst the functions, it appears that $T_F(i) = T_A(i) + T_A(i-1)$ whence the expression for $S(i)$ follows. \square

COROLLARY. *$S(i) < 3^{i+1}$ for all $i \geq 1$. Viz., $S(1) = 6$ and $\lim_{i \rightarrow \infty} S(i+1)/S(i) = 1 + \sqrt{2}$.*

Of course we can obtain that $S(i) < 3^{i+1}$ by a cruder argument. The present analysis, however, is quite straightforward and precise. Running the bracket manipulator on a computer, by way of empirical verification, confirmed the first nine values of S .

2.4. The real-time simulator. Having set the stage in the preceding sections, we now tie everything together to obtain the desired real-time simulator.

Let M be a one-head tape unit with a one-way infinite tape divided into two tracks: the *tag track* and the *count track*. The finite control of M has a special *register* containing the initial segment $C[0:5]$ of the array $C[0:\infty]$ representing the current

count as in §§ 2.1–2.2. The single head of M covers 14 squares and its *position* is the intersquare boundary in the center. Initially, the head covers the leftmost squares, all squares on the tape contain special blank symbols and the finite control is in a distinguished *initial* state, in particular $C[0:5]$ contains $\bar{0}$'s only. Since M can always initialize previously unscanned squares, still containing blanks, by keeping a parity bit in the finite control, we assume that the tape is initially divided in the two tracks as follows. Number the tape squares from left to right by $-7, -6, \dots, 0, 1, 2, \dots$. Square $i, i \geq 0$, contains initially on the count track $C^0[i+6] = \bar{0}$ and on the tag track a tag “[”, if i is even, and a tag “]”, if i is odd. So the initial situation can be visualized as in Fig. 6, with the initial headmarker “>” kept in the finite control. At each step the head rewrites the contents in the squares under scan, and shifts left, right or not at all. Since the head shifts will be governed by the local rewriting rules of the last section, the marker “>” or “<”, positioned on the center intersquare border of the scanned squares, can shift at most two squares left or right in a single step. Whether this marker is “>” or “<” can be maintained in the finite control; the initial marker is “>”.

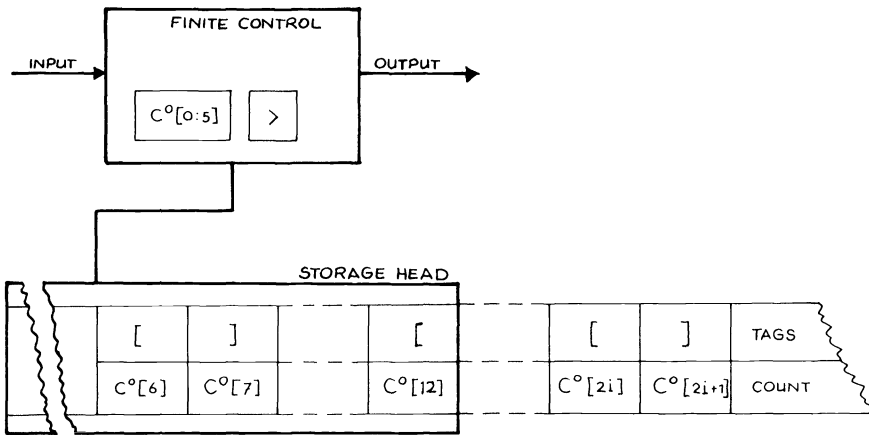


FIG. 6

Each step of M consists of essentially two parts: first execute COUNT on the representation of the currently stored integer, check whether this integer is zero, and secondly execute SWITCH to switch cells containing digits of the integer representation. The information in the two tracks of a square may be thought of as a cell containing the current digit $C[i]$, which is tagged by the tag on the tag track.

To execute COUNT, M inspects the scanned cells right of the headmarker, so as to determine $I(t)$ in the t th step, and also identify the squares containing $C[2i]$, $C[2i+1]$, $C[2i+2]$ and $C[2i+3]$ for all $i \in I(t)$. To this purpose first procedure COLLECT is executed. Let P be the current local tape contents, i.e.

$$P = \begin{matrix} \tau_1 & \tau_2 & \tau_3 & \tau_4 & \tau_5 & \tau_6 & \tau_7 \\ \gamma_1 & \gamma_2 & \gamma_3 & \gamma_4 & \gamma_5 & \gamma_6 & \gamma_7 \end{matrix}$$

is the tape contents on the seven squares right of the headmarker.

Procedure COLLECT (P):

Let the seven squares right of the headmarker contain the string $\tau_1\tau_2 \cdots \tau_7$ on the tag track and the string $\gamma_1\gamma_2 \cdots \gamma_7$ on the count track. Then we distinguish

essentially four cases, implicitly specified by t :

- (a) $\tau_1\tau_2 = [] \ \& \ \tau_3\tau_4 \neq [] \Rightarrow I(t) = \{0, 1, 2\} \ \& \ C[6:7] = \gamma_1\gamma_2;$
 (b) $\tau_1\tau_2\tau_3\tau_4 = [[]] \Rightarrow I(t) = \{0, 1, 2, 3\} \ \& \ C[6:9] = \gamma_1\gamma_2\gamma_3\gamma_4;$
 (c) $\tau_1\tau_2\tau_3\tau_4\tau_5 = [][][] \Rightarrow I(t) = \{0, 1, i\}, \ i > 3, \ \& \ C[2i:2i+3] = \gamma_2\gamma_3\gamma_4\gamma_5;$
 $\tau_1\tau_2\tau_3\tau_4\tau_5\tau_6 = \{[][[[]] \Rightarrow I(t) = \{0, 1, i\}, \ i > 3, \ \& \ C[2i:2i+3]$
 $\qquad\qquad\qquad = \gamma_3\gamma_4\gamma_5\gamma_6;$
 $\tau_1\tau_2\tau_3\tau_4\tau_5\tau_6\tau_7 = \{[][[[]] \Rightarrow I(t) = \{0, 1, i\}, \ i > 3, \ \& \ C[2i:2i+3]$
 $\qquad\qquad\qquad = \gamma_4\gamma_5\gamma_6\gamma_7;$
 (d) none of (a)–(c) $\Rightarrow I(t) = \{0, 1\}.$

Modulo the correctness of the implications in the definition of COLLECT, which remain to be proven, the execution of COLLECT(P) in the t th step of M both determines $I(t) = \{i, i_{-1}, \dots, i_1\}$, $i_i > i_{i-1} > \dots > i_1$, and identifies the locations where $C[2i_j]$, $C[2i_j+1]$, $C[2i_j+2]$ and $C[2i_j+3]$ currently reside, $1 \leq j \leq l$. Since these locations are either under scan on the tape, or in the finite control, viz. $C[0:5]$, the machine can in the t th step execute COUNT(t, δ) = UPDATE(i_i); UPDATE(i_{i-1}); \dots ; UPDATE(i_1); INPUT(δ) by executing the consecutive mappings in the decomposition on the relevant subarrays of $C[0:\infty]$, without explicitly knowing the value of t . Thus, in each single step, starting from the all-blank tape with the initial headmarker “>” positioned at the left end, the one-head tape unit M will do all of the following.

Procedure STEP:

- Step 1.** Initialize both tracks of right adjacent previously unscanned squares, still containing primeval blanks, by writing the correct square bracket on the tag track (check and update parity count in the finite control) and a blank “0” in the count track of such a square.
- Step 2.** Execute COLLECT(P).
- Step 3.** Let the current value of I determined by step 2 be $\{i_i, i_{i-1}, \dots, i_1\}$ with $i_i > i_{i-1} > \dots > i_1$. READ the current value of δ from the input terminal and execute COUNT (current step, δ), that is,
- $$\text{UPDATE}(i_i); \text{UPDATE}(i_{i-1}); \dots; \text{UPDATE}(i_1); \text{INPUT}(\delta).$$
- Step 4.** WRITE “count equal zero” or “count unequal zero” to the output terminal, depending on whether or not $C[0] = \bar{0}$, for the $C[0]$ resulting from step 3.
- Step 5.** Execute SWITCH. That is, switch the contents of the scanned squares, considering the combined contents of the tag track and the count track on a square as a single package. Interchange these packages amongst squares, shift the head position and change the brackets and headmarker, governed by the current headmarker, head position on the scanned squares, and the scanned brackets alone.

PROPOSITION 4. *The constructed one-head tape unit M is oblivious and real-time simulates the quintessential counter.*

Proof. The one-head tape unit M is oblivious since the head movement is governed by the tag track and the headmarker, independent of the input. Attaching imaginary indexes $i = 3, 4, \dots$ to the initial tag track contents, a shift of 2 from the ones in the initial i.d. in the previous section, the executions of SWITCH preserve that pairing of $C[2i]$ with opening bracket indexed i and of $C[2i+1]$ with closing bracket indexed i , $i \geq 3$. Since $C[0:5]$ resides immobile in the finite control, Lemmas 2–5 ensure that the identification of array elements by COLLECT(P) in each step remains correct

under the interchange of the mobile array elements of C on the count track by SWITCH. In the t th step, for all $t \geq 1$, COLLECT (P) determines the value $I(t)$ of the parameter selection function I , as well as the high to low order of elements in $I(t)$. By Lemma 6 and Corollary, for each $t \geq 0$ and each index $i \geq 2$, there exists a step t' , $t < t' \leq t + S(i-1)$, such that $i \in I(t')$. By the definition of COLLECT (P), $\{0, 1\} \subseteq I(t)$ for all $t \geq 1$. Since it follows from Lemma 7 that $S(i-1) < 3^i$ for all $i \geq 2$, the oblivious one-head tape unit M real-time simulates the quintessential counter by Propositions 2 and 3. \square

Let C be any k -counter machine, $k \geq 1$. Clearly, C can be thought of as a finite control connected with k quintessential counters S_1, S_2, \dots, S_k . At each step the finite control of C reads an input command from the input terminal if it is in a polling state, checks each S_i , $1 \leq i \leq k$, for zero contents, and governed by this information issues input commands “add δ_i ”, $\delta_i \in \{-1, 0, 1\}$, to each S_i , $1 \leq i \leq k$, and writes an output string to the output terminal. In the spirit of Proposition 1, we can real-time simulate C by an oblivious one-head tape unit M_C , which is just like M , but with k count tracks (one for each quintessential counter) and one tag track. Storing the first six digits of the representation of each count in the finite control, which is connected to the input and output terminals through C 's original finite control, we finally obtain.

THEOREM. *Each multicounter machine can be real-time simulated by an oblivious one-head tape unit using logarithmic space.*

Proof. By Propositions 1 and 4. That the space used is logarithmic in the simulated number of steps follows since the head is centered immediately left of the square containing tag “ I_{i+1} ” for the first time after executing $A(>, i)$, which takes $T_A(i)$ steps. To clean up some final details: we can get rid of the fat head, covering 14 squares and sometimes shifting its center two squares in a single step, by cutting out a piece of tape of 14 squares and buffering it in the finite control. The remains of the tape are glued together and the contents of the buffered piece are swapped from the buffer to the scanned tape square and vice versa, according to the desired head motion, cf. the speed-up technique in [3]. \square

On the required bits. Although the preceding simulation and its proof may not seem easy, the algorithm which does the work is pretty simple. As it happens, we are also frugal in the number of bits. On information-theoretical grounds we require about $k \log_2 2n$ bits to represent any k -tuple of integers of absolute values up to n . In the exhibited simulation, we can use four bits for each digit of a count, need not more than $\log_2 n$ digits for each count, and since there are but four tags, each tag can be encoded in two bits. Therefore, we use at most about $(4k+2) \log_2 n$ bits to represent k counts of absolute values at most n . By restricting the most significant nonzero digit to absolute value 1, and appropriately modifying the mappings UPDATE and INPUT, everything goes through as before but code $(c) \subseteq \{-2, -1, 0, 1, 2, -\bar{1}, \bar{0}, \bar{1}\}^\infty$, $c \in \mathbb{Z}$. Thus we only have to use $(3k+2) \log_2 n$ bits to represent k counts of absolute values at most n . Using only digits from $\{-2, -1, 0, 1, 2, \bar{0}\}$ also suffices, but complicates the proof. How good a real-time algorithm is can be measured in the size of the storage alphabet used. Realizing that actual machines use a constant size storage alphabet, we observe that a large, although finite, storage alphabet in an algorithm implies a greater constant delay. That is, the reverse of a speed-up by decreasing the alphabet size. At the cost of a deterioration of the constant delay, implicit in the real-time solution presented, we can do better than using $(3k+2) \log_2 n$ bits. Using in §§ 2.1 and 2.2 an analogous redundant symmetric r -ary representation, based on the digits $-r, -r+1, \dots, -1, 0, 1, 2, \dots, r-1, r$, we can get the bit count down to about $(1+4/\log_2 r) k \log_2 n$ bits for maintaining k counts of absolute value at most n . The implicit

constant delay, however, rises proportional to $\log r$. In the limit, for $r \rightarrow \infty$, we achieve about the information-theoretical minimum in bits, but the constant delay goes to infinity, that is, it takes infinite time to execute a single step.

Note, however, that for no fixed finite storage alphabet a real-time simulation of but a single counter on an oblivious multitape Turing machine can reach the information-theoretical bit minimum. Such a simulation must use $\Omega(\log n)$ size representations for counts of size n , and we can argue that for each n there must be at least $\log n$ representations. Hence we use at least $\log_2 2n + \log_2 \log n$ bits per count.

On the size of the fat head. In the simulation a head covering 10 squares suffices, which can be shown by a slight complication of the proof. Also, the head shift in a single step of M need not exceed one square.

On the initially zero counts. As argued subsequent to the proof of Proposition 3, the assumption of initially zero counts is not essential. The theorem holds also for multicounter machines with each count initialized to an arbitrary integer.

3. Conclusion. For various theoretical and practical reasons, multitape Turing machines, restricted in one or more resources, serve as a standard against which to calibrate the power of other devices, or to compare the power among themselves under different resource restrictions. The commonly considered resources are time, space, number of tapes/storage heads and oblivious versus nonoblivious. The present simulation is, perhaps, the first one which is optimal in all of these resources at once: the use of no resource can be improved by relaxing the other resource restrictions. Apart from the fact that the simulating device is real-time, oblivious and uses but a single storage head, it is worthwhile to recall that there *do not exist* on-line Turing machines using $S(n) \in o(\log n)$ space, $S(n)$ unbounded [4]. Thus, the simulation is performed by the simplest (with respect to the considered resources) Turing machine which is not an outright finite automaton. Another resource, which is sometimes considered, is the number of head reversals. Again, it is easy to see that each multitape Turing machine needs, in the worst case, a linear number of head reversals to on-line simulate a counter machine, as does the presented simulation. (Although a multihead Turing machine can simulate a multicounter machine without head reversals [8], the simulation of such a device by a multitape Turing machine needs a linear number of head reversals.)

Some immediate applications. In a computation using k stacks we may want to keep track of which pairs out of the k stacks are of equal height at any time. Without slowing down the computation, we formerly needed $k - 1$ stacks for doing so. Using the present method we need but one extra oblivious one-head tape unit, or two extra oblivious pushdown stores. A single pushdown store does not suffice. Similarly, we can keep track of the headpositions in multihead Turing machine computations.

Number representations. The reader may appreciate the following comment of *John Locke* on the intimate relation between counting and number representation.

For he that will count 20, or have any *idea* of that number, must know that 19 went before, with the distinct name or sign of every one of them, as they stand marked in their order; for wherever this fails, a gap is made, the chain breaks, and the progress in numbering can go no further. So that *to reckon right, it is required*: (1) that the mind distinguish carefully two *ideas*, which are different one from another only by the addition or subtraction of one unit; (2) that it retain in memory [a systematic method for deriving] the names or marks of the several combinations, from a unit to that number, and that not confusedly and at random, but in that exact order that the numbers follow one another; in either of which, if it trips, the whole business of numbering will be disturbed, and there will remain only the confused *idea* of multitude, but the *ideas* necessary to distinct numeration will not be attained to.

The one and only basic reason to denote numbers at all is for the purpose of comparing them, of whether the one is greater than the other, for without this capability no arithmetic is possible and with it all arithmetic is possible. Thus we must be able to distinctly represent all numbers, and if we have representations for all numbers up to a given one, then we must be able to derive the next one, or previous one, from the given one, while having a designated point of reference or benchmark number. This is the task expressed in the notion of a counter machine, and multicounter machines enable us to do arithmetic. The exhibited optimal implementation embodies a new representation for multituples of integers suitable for exercising that basic activity using minimal resources. Thus, for each $n = (n_1, n_2, \dots, n_k) \in \mathbb{Z}^k$, $k \geq 1$, each such representation for n consists of a linear string of symbols, and is about as compact as possible. Such a representation has a distinguished *access position* p , and by considering only the three symbols centered on the access position we can

- (i) add any vector $\delta = (\delta_1, \delta_2, \dots, \delta_k) \in \{-1, 0, 1\}^k$ to n to obtain such a representation for $n + \delta$;
- (ii) for all i , $1 \leq i \leq k$, determine whether $n_i + \delta_i = 0$;
- (iii) determine the new access position $p' \in \{p-1, p, p+1\}$, which is also independent of n and δ . In m successive additions the distance between the leftmost and rightmost intermediate access pointer positions is $O(\log m)$, for all $m > 1$.

Note that Gray codes, as representations of integers, have vaguely similar properties for the case $k = 1$. There, the representation of $n \pm 1$, $n \in \mathbb{Z}$, can be obtained from the representation of n by changing a single symbol. However, the symbol in the representation which must be changed to obtain $n + 1$ from n can lie arbitrary far from the symbol which must be changed to obtain $n - 1$ from n . Moreover, these positions depend on n and whether we add or subtract, and do not allow us to test n for 0. The representation derivable from the simulation in [1] is closer to the one above, for the case $k = 1$, but the new access position p' in (iii) depends on n and δ . None of these representations have any of the properties (i)–(iii) in case $k > 1$.

Augmented counter machines. Apart from the basic one-step multicounter operations, several other one-step operations can be synthesized using concealed auxiliary counters, such as tests for equality amongst counters (by maintaining all differences on auxiliary counters). It is known [2] that the operations “set counter i to zero” or “set counter i to the value of counter j ” ($i \neq j$) cannot be synthesized as one-step operations on a multicounter machine. At the end of § 2.2 we noted that the requirement of initially zero counters was not essential for the present simulation. It can be proved [9] (this issue, pp. 34–40) that with a suitable embellishment the present simulation can also support the one-step operation “set counter i to the value of counter j ” ($i \neq j$). Define an *augmented counter machine* (ACM) just like a multicounter machine but with the one-step input operations “set counter i to the value of counter j ” (for any pair of counters i, j) added and any initial counter contents in \mathbb{Z} allowed. Such a machine can execute quite powerful instructions in one step. For example:

L : if $(x < y \ \& \ y \geq c)$ then $(x \leftarrow z; z \leftarrow d)$ else $(x \leftarrow y; \text{goto } L')$ fi

with x, y, z integer variables, c, d integers and L, L' labels, is a one-step instruction for an ACM.

THEOREM. *Each augmented counter machine can be real-time simulated by an oblivious one-head tape unit in logarithmic space.*

Uniform space complexity. Viewed in space-time, the bracket manipulator head describes an interesting curve. This is perhaps best expressed by stating that the

two-dimensional space-time trajectory described by the center of the greatest tape segment, delimited by brackets with indices $j, j' \leq i$, is the same as that described by the center of the greatest tape segment, delimited by brackets with indices $j, j' \leq i-1$, $i > 1$, subsequent to multiplying the time scale of the latter by $S(i)/S(i-1)$ and the space scale of the latter by $i/(i-1)$. This shows that the number of distinct squares visited in each time interval of n steps, for all $n \geq 1$, is $\Theta(\log n)$. Generalizing this observation, we say that a multitape Turing machine M uses *uniform logarithmic space* if, for any unbounded input sequence, the total number of distinct squares, visited on M 's storage tapes, for each interval of n steps, for all $n \geq 1$, is $O(\log n)$. It can be shown [10] that each multitape Turing machine using uniform logarithmic space can be real-time simulated by an oblivious one-head tape unit using uniform logarithmic space.

Oblivious simulations. It seems to us that also the converse of the maxim leading to Proposition 1 holds generally. Viz., if we can simulate arbitrarily many storage devices by a fixed number of, possibly different, devices then we can do so obliviously retaining the same resource bounds. The point here is that if the multitude of head movements of an arbitrary number of heads can be accommodated by the motion of a fixed number of heads, then there is no reason to suppose that any trajectory of the latter can make significant use of particular input streams.

REFERENCES

- [1] M. J. FISCHER AND A. L. ROSENBERG, *Real-time solutions of the origin-crossing problem*, Math. Systems Theory, 2 (1968), pp. 257-263.
- [2] P. C. FISCHER, A. R. MEYER AND A. L. ROSENBERG, *Counter machines and counter languages*, Math. Systems Theory, 2 (1968), pp. 265-283.
- [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 AND J. D. ULLMAN, *Some results on tape-bounded Turing machines*, J. ACM, 16 (1969), pp. 168-177.
- [5] M. MINSKY, *Recursive unsolvability of Post's problem of 'tag' and other topics in the theory of Turing machines*, Ann. Math., 74 (1961), pp. 437-455.
- [6] N. PIPPENGER AND M. J. FISCHER, *Relations among complexity measures*, J. ACM, 26 (1979), pp. 361-381.
- [7] C. P. SCHNORR, *The network complexity and Turing machine complexity of finite functions*, Acta Informatica, 7 (1976), pp. 95-107.
- [8] P. M. B. VITÁNYI, *On efficient simulations of multicounter machines*, Inform. and Control, 55 (1982), pp. 20-39.
- [9] ———, *An optimal simulation of counter machines: the ACM case*, this Journal, this issue, pp. 34-40.
- [10] ———, *On the simulation of many storage heads by one*, Technical Report IW 228, Mathematisch Centrum, Amsterdam, 1983, preliminary version in Proc. 10th ICALP, Lecture Notes in Computer Science 154, Springer-Verlag, Berlin, 1983, pp. 687-694.

AN OPTIMAL SIMULATION OF COUNTER MACHINES: THE ACM CASE*

PAUL M. B. VITÁNYI†

Abstract. An Augmented Counter Machine (ACM) is a multicounter machine, with initially nonzero counters allowed, and the additional one-step instruction “set counter i to the value of counter j ”, for any pair of counters i and j . Each ACM can be real-time simulated by an oblivious one-head tape unit using the information-theoretical storage optimum.

Key words. counter machine, multicounter machine, augmented counter machine, real-time simulation by oblivious one-tape Turing machine, number representation, counting, coding

1. Introduction. In the companion paper [3] (this issue, pp. 1–33), a real-time implementation of multicounter machines on oblivious one-head tape units of optimal storage efficiency was exhibited. An *augmented counter machine* (ACM) is a multicounter machine, with each of its counters initialized to any integer, and with the additional one-step operation “set counter i to the value of counter j ”, for any pair of counters i and j . Several one-step operations, other than the basic ones, can be synthesized on a multicounter machine by the use of concealed auxiliary counters (such as “test equality of a pair of counters”, for any such pair, by maintaining the differences on auxiliary counters). It is known that the above assignment among counters cannot be so synthesized. A witness for this fact is the language L^* , with $L = \{0^p 1^m \mid p \geq m > 0\}$. Thus, in real-time, ACM’s are more powerful than multicounter machines [1]. The particular technique used in [3], to obtain an optimal simulation of counter machines, is well suited to extend that result to the more powerful ACM’s. Consequently, we shall demonstrate the next theorem.

THEOREM. *Each augmented counter machine can be real-time simulated by an oblivious one-head tape unit using the information-theoretical minimum in storage space. (Viz., for each $t \geq 0$ and $n \geq 1$, during the processing of the $(t+1)$ th through $(t+n)$ th input command, of the simulated ACM, the storage head of the simulating oblivious one-head tape unit accesses but $O(\log n)$ distinct tape squares.)*

In [3] the analogue of the theorem was derived for the weaker multicounter machines. The next section, containing the demonstration of the above theorem, continues and presupposes that paper.

Outline of the simulation. The simulation consists of the oblivious one-head tape unit constructed in [3], equipped with some additional features. The $(k+1)$ -track one-head tape unit M of [3], capable of real-time simulation of a k -counter machine, has one tag track, which does not concern us here, and k count tracks containing the momentary representations of the k stored integers, one per track. M would trivially be capable of real-time simulating an ACM, if it could replace the contents of any count track by that of any other in each single step. This is clearly impossible for a one-head tape unit, since the significant count track contents may be arbitrarily large. Yet we were able to update the individual tracks, with respect to unit addition/subtraction, by amortizing the propagation of the carries and borrows. The idea below is to do the same with respect to the replacement of one count track contents by that of

* Received by the editors March 15, 1983. This work is registered at the Mathematical Centre as IW 225/83.

† Centre for Mathematics and Computer Science, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands.

another. Thus, if at some step t the counter i is set to the value of counter j , we start to transfer the contents of count track j to count track i , from the low order digits to the high order digits, a few digits each step t' , $t' \geq t$. We do so by introducing a switch which, in each position it passes, overwrites the digit on count track i by the corresponding digit on count track j . Each such switch is introduced in the 0th position of the representation, and is shifted through simultaneously scanned adjacent positions to higher ones, preliminary to the propagation of carries and borrows, in each step. The effect is that the carry or borrow, resulting from an input at a later time than the input which was to replace the contents of counter i by that of counter j , will always be preceded by the replacing of the individual digits constituting the contents of count track i by their counterparts on count track j . Since in each interval of n steps, for all $n \geq 1$, the head visits but $O(\log n)$ distinct tape squares, each switch eventually overtakes all earlier created switches, but never passes them. We are thus confronted with arbitrarily long queues of switches clogging at some positions on the tape. It will be shown, however, that whenever one switch overtakes another one, we can replace the combination by a single switch. It thus suffices to equip the multicounter simulator of [3] with an extra track on its tape, and modify the algorithm it executes in each step, to derive the desired ACM implementation.

2. An optimal simulation of ACM's. Recall that, in the proof of the optimal simulation of multicounter machines in [3], the usual assumption of initially zero counters was not essential. The simulation presented there also works with each counter initially set to any integer. To turn such a machine into an ACM, we therefore only have to add operations which can instantly replace the contents of any counter by that of any other counter. This amounts to an operation which is more general than a permutation of the momentary contents amongst the various counters.

Define a *semipermutation* σ , among k objects o_1, o_2, \dots, o_k , for $\sigma = i_1 i_2 \dots i_k$ ($i_j \in \{1, 2, \dots, k\}$ for $1 \leq j \leq k$) by

$$\sigma(o_1, o_2, \dots, o_k) = (o_{i_1}, o_{i_2}, \dots, o_{i_k}).$$

A semipermutation is also called a *permutation with repetitions*. The semigroup (not group), of which the elements are semipermutations of k objects, the product of two semipermutations being the semipermutation resulting from applying each in succession, and the identity ε being the semipermutation which does not change anything, has k^k elements and is denoted by R_k .

Define an *augmented counter machine* (ACM) A as a k -counter machine with each counter i , $1 \leq i \leq k$, initialized to a value in the set of integers. Input commands to A are of the format (σ, δ) with $\sigma \in R_k$ and $\delta \in \{-1, 0, 1\}^k$. At any time, if (c_1, c_2, \dots, c_k) is the integer valued k -vector contained in A 's k counters and (σ, δ) is the currently polled input command then in one step A does all of the following:

- (i) $(c_1, c_2, \dots, c_k) \leftarrow \sigma(c_1, c_2, \dots, c_k)$;
- (ii) $(c_1, c_2, \dots, c_k) \leftarrow (c_1, c_2, \dots, c_k) + \delta$;
- (iii) OUTPUT, for all i , $1 \leq i \leq k$, "counter $i = 0$ " or "counter $i \neq 0$ " according to the new state of affairs.

Let M be the k -counter machine simulator as constructed in [3]. The i th count track contains the array $C[i, 6:\infty]$, and the i th register in the finite control contains $C[i, 0:5]$. The array $C[i, 0:\infty]$ represents the integer c_i , that is, the value of the i th counter, $1 \leq i \leq k$, just as the array $C[0:\infty]$ represented the value c of the quintessential counter in [3]. The initial arrays $C^0[i, 0:\infty]$, $1 \leq i \leq k$, are representations of the prescribed initial integers, each representation containing no digits of opposite sign,

cf. the conclusion of § 2.2 in [3]. The following adaptation, of procedure STEP in [3, § 2.4], trivially turns M into an ACM simulator. Replace **step 3** of procedure STEP by **step 3₁** below, turning it into a new procedure STEP₁. The resulting machine is M_1 , and the contents of the k count tracks are contained in a $k \times \infty$ matrix $C_1[1:k, 0:\infty]$, such that $C_1[i, 0:\infty]$ denotes the contents of the i th register in M_1 's finite control, followed by the i th count track on M_1 's tape, in the obvious way, $1 \leq i \leq k$.

Step 3₁: Let the current value of I , determined by **step 2**, be $\{i_t, i_{t-1}, \dots, i_1\}$ with $i_t > i_{t-1} > \dots > i_1$. READ the current command (σ, δ) from the input terminal. Let $\delta = (\delta_1, \delta_2, \dots, \delta_k)$. Execute:

```

for  $j = 0$  step 1 until  $\infty$ 
  do
     $C_1[1:k, j] \leftarrow \sigma C_1[1:k, j]$ 
  od;
for  $j = l$  step -1 until 1
  do for  $i = 1$  step 1 until  $k$ 
    do
       $C_1[i, 2i_j : 2i_j + 3] \leftarrow \text{UPDATE} (C_1[i, 2i_j : 2i_j + 3])$ 
    od
  od;
for  $i = 1$  step 1 until  $k$ 
  do
     $C_1[i, 0:1] \leftarrow \text{INPUT}_{\delta_i} (C_1[i, 0:1])$ 
  od

```

Step 3₁, however, contains an infinite **for** statement. (That statement is the only addition to the original **step 3**.) Since the cardinality of $I(t)$ happens to be at most 4, for all t , cf. [3], only a few positions of the arrays, representing the counters, can be updated by the actual machine in each step. Consequently, M_1 does not constitute a real machine, since it executes the procedure STEP₁, containing an infinite **for** statement, that accesses all of the infinite tape (c.q., $C_1[1:k, 0:\infty]$), each single step. We shall amortize the execution of the infinite **for** statements, implementing the semipermutations, by executing them in each position only when they are due.

We observe the notational conventions from [3], concerning superscripts on arrays. Thus, an array B , connected with a machine M_i , $i = 1, 2$, can be viewed as a variable or as an actual value. In the first case we do not use a superscript. In the latter case a superscript t is used to indicate the value of B , subsequent to the execution by M_i of the t th step (i.e., procedure STEP _{i}), for a given input command sequence (σ^1, δ^1) , $(\sigma^2, \delta^2), \dots, (\sigma^t, \delta^t), \dots$. That is, B 's value just before M_i processes the $(t+1)$ th input command $(\sigma^{t+1}, \delta^{t+1})$.

We associate, with each position $j \geq 0$, a *queue* $Q[j]$ of semipermutations. If $Q[j] = \sigma_m \sigma_{m-1} \dots \sigma_1$ then the constituent semipermutations $\sigma_1, \sigma_2, \dots, \sigma_m$ have been executed, in that order, on all positions j_1 , $0 \leq j_1 \leq j$, but none of them has as yet been executed on any position j_2 , $j_2 > j$. Queues of semipermutations can be *concatenated* to a single queue. That is, if $Q[j_1] = \sigma_p \sigma_{p-1} \dots \sigma_1$ and $Q[j_2] = \nu_q \nu_{q-1} \dots \nu_1$ then by definition:

$$Q[j_1]Q[j_2] = \sigma_p \sigma_{p-1} \dots \sigma_1 \nu_q \nu_{q-1} \dots \nu_1.$$

For each $j \geq 0$, the initial contents of $Q[j]$ is ε , that is, the *empty* queue. For each particular input command sequence, for each time $t \geq 0$, we denote, for all $j \geq 0$, the

queue in position j at time t by $Q^t[j]$. Thus, $Q^0[j] = \varepsilon$ for all $j \geq 0$. For any input sequence $(\sigma^1, \delta^1), (\sigma^2, \delta^2), \dots, (\sigma^t, \delta^t), \dots$, with $\sigma^t \in R_k$ and $\delta^t \in \{-1, 0, 1\}^k$, for all $t \geq 1$, we preserve the following invariant:

$$(E) \quad \forall_{t \geq 0} t [Q^t[0]Q^t[1] \cdots Q^t[j]Q^t[j+1] \cdots = \sigma^t \sigma^{t-1} \cdots \sigma^1 \& \forall_{i \geq 0} i [Q^t[2i] = \varepsilon]].$$

(Recall that, in [3], invariants (A)–(D) pertain to the representation $C[i, 0: \infty]$ of the contents of the i th simulated counter, for each i , $1 \leq i \leq k$.)

If $Q[j] = \sigma_m \sigma_{m-1} \cdots \sigma_1$ then by *application* of $Q[j]$ to a k -vector $\nu = (\nu_1, \nu_2, \dots, \nu_k)$, denoted as

$$(\nu_1, \nu_2, \dots, \nu_k) \leftarrow Q[j](\nu_1, \nu_2, \dots, \nu_k),$$

we mean the assignment embodied in the execution of:

```

for  $j = 1$  step 1 until  $m$ 
  do
     $(\nu_1, \nu_2, \dots, \nu_k) \leftarrow \sigma_j(\nu_1, \nu_2, \dots, \nu_k)$ 
  od

```

Now replace the third step of procedure STEP by **step 3₂**, so as to obtain a new procedure STEP₂. The corresponding machine is M_2 and, for any input sequence $(\sigma^1, \delta^1), (\sigma^2, \delta^2), \dots, (\sigma^t, \delta^t), \dots$, with $\sigma^t \in R_k$ and $\delta^t \in \{-1, 0, 1\}^k$, the matrix $C_2[1: k, 0: \infty]$ contains the contents of the k count tracks and k count registers of M_2 in the obvious way.

Step 3₂: Let the current value of I , determined in **step 2**, be $\{i_l, i_{l-1}, \dots, i_1\}$ with $i_l > i_{l-1} > \dots > i_1$. READ the current command (σ, δ) from the input terminal. Let $\delta = (\delta_1, \delta_2, \dots, \delta_k)$. Execute:

```

for  $j = l$  step  $-1$  until 1
  do
     $C_2[1: k, 2i_j + 2: 2i_j + 3] \leftarrow Q[2i_j + 1]C_2[1: k, 2i_j + 2: 2i_j + 3];$ 
     $Q[2i_j + 3] \leftarrow Q[2i_j + 1]Q[2i_j + 3];$ 
     $Q[2i_j + 1] \leftarrow \varepsilon;$ 
    for  $i = 1$  step 1 until  $k$ 
      do
         $C_2[i, 2i_j: 2i_j + 3] \leftarrow \text{UPDATE}(C_2[i, 2i_j: 2i_j + 3])$ 
      od
    od;
     $C_2[1: k, 0: 1] \leftarrow \sigma C_2[1: k, 0: 1];$ 
     $Q[1] \leftarrow \sigma Q[1];$ 
    for  $i = 1$  step 1 until  $k$ 
      do
         $C_2[i, 0: 1] \leftarrow \text{INPUT}_{\delta_i}(C_2[i, 0: 1])$ 
      od

```

Obviously, (E) is preserved by **step 3₂** for each input sequence.

LEMMA 1. For each input sequence it holds that for all $t \geq 0$ and all i , $1 \leq i \leq k$, we have:

$$C_1^t[i, 0] = \bar{0} \quad \text{iff} \quad C_2^t[i, 0] = \bar{0}.$$

Proof. Define a third $k \times \infty$ matrix C_3 , which *normalizes* C_2 , at any instant of time t , by executing the backlog of semipermutations which by that time have accumulated (in the queues for) the consecutive positions j , with respect to the k -vectors $C_2^t[1: k, j]$.

By definition then, for all $t \geq 0$ and all $j \geq 0$:

$$C_3'[1 : k, j] = (Q'[0]Q'[1] \cdots Q'[j-1])C_2'[1 : k, j].$$

The following claim expresses the essence of the amortization-of-execution-of-semi-permutations argument. Viz., for each input sequence, for every t and i , $C_3'[i, 0 : \infty]$ and $C_1'[i, 0 : \infty]$ represent the same integer.

CLAIM. For any particular input sequence $(\sigma^1, \delta^1), (\sigma^2, \delta^2), \dots, (\sigma^t, \delta^t), \dots$:

$$\forall_{t \geq 0} \forall_{1 \leq i \leq k} \exists_{c \in \mathbb{Z}} [C_3'[i, 0 : \infty] \in \text{code}(c) \text{ iff } C_1'[i, 0 : \infty] \in \text{code}(c)],$$

where \mathbb{Z} is the set of integers.

Proof of claim. By induction on the number of steps t , for any particular input sequence $(\sigma^1, \delta^1), (\sigma^2, \delta^2), \dots, (\sigma^t, \delta^t), \dots$.

Base case. $t=0$. Since $Q^0[j] = \varepsilon$, for all positions $j \geq 0$, and C_1^0 and C_2^0 both represent the same k -vector of prescribed integers according to the code function, cf. [3], the claim holds initially.

Induction. $t \geq 0$. Assume the claim holds for all $s \leq t$. Let $I(t+1) = \{i_t, i_{t-1}, \dots, i_1\}$, with $i_t > i_{t-1} > \dots > i_1$. Recall from [3] that, for each $t \geq 0$, the least element i_1 in $I(t+1)$ equals 0. This will be needed later in the proof. By the inductive assumption, for each i , $1 \leq i \leq k$, and for all s , $0 \leq s \leq t$, there is an integer c_i^s such that $C_3^s[i, 0 : \infty]$, $C_1^s[i, 0 : \infty] \in \text{code}(c_i^s)$, since M_1 obviously simulates an ACM A just as M in [3] simulates multicounter machines. During step $t+1$, the running variable j assumes the successive values $l, l-1, \dots, 1$ in **step 3₂** of procedure STEP₂. For each such j , the piece of code

$$\begin{aligned} & C_2[1 : k, 2i_j + 2 : 2i_j + 3] \leftarrow Q[2i_j + 1]C_2[1 : k, 2i_j + 2 : 2i_j + 3]; \\ (1) \quad & Q[2i_j + 3] \leftarrow Q[2i_j + 1]Q[2i_j + 3]; \\ & Q[2i_j + 1] \leftarrow \varepsilon \end{aligned}$$

in **step 3₂** does not change the normalized matrix $C_3[1 : k, 0 : \infty]$ at all. The execution of (1) also preserves (E), viz., in particular $Q[i] = \varepsilon$, for all even i . Now consider the next piece of **step 3₂**:

$$\begin{aligned} & \text{for } i = 1 \text{ step 1 until } k \\ (2) \quad & \text{do} \\ & C_2[i, 2i_j : 2i_j + 3] \leftarrow \text{UPDATE}(C_2[i, 2i_j : 2i_j + 3]) \\ & \text{od} \end{aligned}$$

Just before the execution of this **for** statement, the matrix $C_3[1 : k, 0 : \infty]$ consisted of three submatrices:

$$\begin{aligned} & C_3[1 : k, 0 : 2i_j - 1], \\ & C_3[1 : k, 2i_j : 2i_j + 3], \\ & C_3[1 : k, 2i_j + 4 : \infty]. \end{aligned}$$

Since $Q[2i_j] = Q[2i_j + 2] = \varepsilon$, by invariant (E), and $Q[2i_j + 1]$ has just been set to ε by the preceding subprogram (1), it follows from the definition of C_3 that, just before execution of (2), it holds that:

$$(3) \quad C_3[1 : k, 2i_j : 2i_j + 3] = (Q[0]Q[1] \cdots Q[2i_j - 1])C_2[1 : k, 2i_j : 2i_j + 3].$$

Only $C_2[1 : k, 2i_j : 2i_j + 3]$ is accessed and changed (row-by-row) according to UPDATE in (2). Therefore, by equality (3), the effect on the normalized matrix $C_3[1 : k, 0 : \infty]$,

of executing the subprogram (2) on $C_2[1:k, 0:\infty]$, is the same as the effect of executing:

```

for  $i = 1$  step 1 until  $k$ 
  do
     $C_3[i, 2i_j: 2i_j + 3] \leftarrow \text{UPDATE}(C_3[i, 2i_j: 2i_j + 3])$ 
  od

```

By [3, Propositions 1–4], therefore, if $C_3[i, 0:\infty] \in \text{code}(c_i)$, for some integer c_i , before the execution of (2), then $C_3[i, 0:\infty] \in \text{code}(c_i)$ after the execution of (2) too, for each i , $1 \leq i \leq k$. As noted above, for all $t \geq 0$, the least element of $I(t+1)$ is $i_1 = 0$, by [3]. So subsequent to the last execution of the subprogram (1); (2) in the $(t+1)$ th step, that is, the execution with $j = 1$ and therefore $i_j = i_1 = 0$, we have $Q[1] = \varepsilon$ while $Q[0] = \varepsilon$ by (E). Hence, by definition, $C_3[1:k, 0:1]$ now equals $C_2[1:k, 0:1]$, while, by the inductive assumption and the above reasoning, still $C_3[i, 0:\infty] \in \text{code}(c'_i)$, for all i , $1 \leq i \leq k$. In this situation

$$(4) \quad \begin{aligned} C_2[1:k, 0:1] &\leftarrow \sigma C_2[1:k, 0:1]; \\ Q[1] &\leftarrow \sigma Q[1] \end{aligned}$$

is executed. Thus, the array $C_3[1:k, 0:\infty]$, derivable from the new values of $C_2[1:k, 0:\infty]$ and $Q[0:\infty]$, yields, for $\sigma = j_1 j_2 \cdots j_k$, that $C_3[i, 0:\infty] \in \text{code}(c'_i)$, for $1 \leq i \leq k$, while $Q[0] = \varepsilon$. Consequently, under the inductive assumption, after the execution of (4) in the $(t+1)$ th step, we have $C_3[i, 0:\infty] \in \text{code}(c'_i)$, just as we trivially have $C_1[i, 0:\infty] \in \text{code}(c'_i)$, subsequent to the execution of

$$(5) \quad \begin{aligned} &\textbf{for } j = 0 \textbf{ step } 1 \textbf{ until } \infty \\ &\textbf{do} \\ &C_1[1:k, j] \leftarrow \sigma C_1[1:k, j] \\ &\textbf{od} \end{aligned}$$

in the $(t+1)$ th step of M_1 (using STEP_1 containing **step 3₁**). Meanwhile, we still have $C_3[1:k, 0:1] = C_2[1:k, 0:1]$, since $Q[0] = \varepsilon$. Therefore, subsequent to the final piece

$$(6) \quad \begin{aligned} &\textbf{for } i = 1 \textbf{ step } 1 \textbf{ until } k \\ &\textbf{do} \\ &C_2[i, 0:1] \leftarrow \text{INPUT}_{\delta_i}(C_2[i, 0:1]) \\ &\textbf{od} \end{aligned}$$

of **step 3₂**, yielding the new values of C_2 and C_3 , viz., C_2^{t+1} and C_3^{t+1} , we still have $C_3^{t+1}[i, 0:1] = C_2^{t+1}[i, 0:1]$, for all i , $1 \leq i \leq k$. Moreover, by the properties of **INPUT** in [3] we also have $C_3^{t+1}[i, 0:\infty] \in \text{code}(c'_i + \delta_i)$, for all i , $1 \leq i \leq k$. Trivially, in view of [3], for all i ($1 \leq i \leq k$), it holds that $C_1^{t+1}[i, 0:\infty] \in \text{code}(c'_i + \delta_i)$. This concludes the induction and the proof of the claim. \square

Since invariant (E) is preserved by **step 3₂**, and therefore $Q[0] \equiv \varepsilon$, we have by definition that $C_3[1:k, 0:1] \equiv C_2[1:k, 0:1]$. In [3, Proposition 2] it was shown that the lowest order digit of a representation in code (c) equals $\bar{0}$ iff c equals 0. Together with the Claim, these two observations imply the Lemma. \square

Since it is trivial that M_1 real-time simulates the required ACM, by Lemma 1 it follows from [3] that, if the machine M_2 can be realized, the Theorem holds.

LEMMA 2. M_2 can be constructed as a machine satisfying the specifications in the Theorem.

Proof. The only difficulty with M_2 concerns the storage, execution, transport and concatenation of arbitrary large queues of semipermutations. Since the semipermutations form the semigroup R_k under concatenation, no queue $Q[j]$, $j \geq 0$, ever needs

to contain more than a single element from R_k . Since every $Q[j]$, $j \geq 0$, initially contains the unity element ε , each subsequent execution of **step 3₂** can compute the single semipermutation which represents the current contents of $Q[j]$ in R_k , for any such $Q[j]$ involved. Storing $Q[j]$ in the cell containing $C[1:k, j]$, for all $j \geq 0$, so in the finite control of M_2 for $0 \leq j \leq 5$ and on its tape for $j \geq 6$, shows that M_2 has the same specifications as the multicounter simulator M in [3]. Hence the lemma. \square

The Theorem follows from [3, Propositions 1-4] and Lemmas 1 and 2 above, combined with the observation that M_1 trivially real-time simulates any ACM.

3. Final remarks.

Optimality. Since the ACM implementation, constructed above, has the same complexity, with respect to the measures concerned, as does the multicounter machine implementation in [3], it is *a fortiori* also optimal in all commonly considered complexity measures at once.

On the required number of bits. There are k^k semipermutations in R_k . To denote each of them, it suffices to use $k \log_2 k$ bits. Similar to [3], we note that, under the scheme outlined in § 2, it suffices to use $(4k + k \log_2 k + 2) \log_2 n$ bits to represent k counts, of absolute value not greater than n , in the ACM simulator. Using a redundant symmetric r -ary representation [3], based on the digits $-r, -r+1, \dots, 0, 1, \dots, r-1, r$, we can bring the bit count down to below $(1 + (4 + \log_2 k)/\log_2 r) k \log_2 n$ bits, and therefore arbitrary close to the information-theoretical minimum, to the detriment of the implicit constant delay, as in [3].

Simulations of ACM's on other devices. In [2] we gave optimal simulations of multicounter machines on RAM's, combinational logic networks, cyclic logic networks and VLSI. The method used above, of amortizing execution of semipermutations to extend the simulation of multicounter machines by tape units to a simulation of ACM's by the same, can also be used to extend the optimal simulations of multicounter machines, by the above devices as in [2], to optimal simulations of ACM's by these devices. As here, the complexity of the simulations of the ACM's, by these devices, is none other than the complexity of the corresponding simulations of multicounter machines.

REFERENCES

- [1] P. C. FISCHER, A. R. MEYER AND A. L. ROSENBERG, *Counter machines and counter languages*, Math. Systems Theory, 2 (1968), pp. 265-283.
- [2] P. M. B. VITÁNYI, *On efficient simulations of multicounter machines*, Inform. and Control, 55 (1982), pp. 20-39.
- [3] ———, *An optimal simulation of counter machines*, this Journal, this issue, pp. 1-33.

ON CIRCUIT-SIZE COMPLEXITY AND THE LOW HIERARCHY IN NP*

KER-I KO† AND UWE SCHÖNING‡

Abstract. Let A be a set having polynomial size circuits. If A is also known to be in NP, then we may conclude that the graph of the polynomial size circuits for A is actually in Π_2^P . Using this observation, we show that sets in NP which have polynomial size circuits are in L_3^P , the third level of the low hierarchy in NP. By a similar technique, we are able to show that some other intuitively low sets in NP are in L_2^P , and even in a certain refinement of L_2^P . As a consequence, sparse sets are not strong nondeterministic polynomial time Turing complete in NP unless the polynomial time hierarchy collapses to Δ_2^P .

Key words. NP, low hierarchy, polynomial size circuits, sparseness

1. Introduction. Recent studies on the structure of intractable sets revealed the incompatibility between NP-completeness and some structural properties such as sparseness. Berman [5] showed that tally sets cannot be polynomial time many-one complete (abbr. \leq_m^P -complete) for NP unless $P = NP$. Fortune [7] showed that co-sparse sets cannot be \leq_m^P -complete for NP unless $P = NP$. Mahaney [14] solved the original Hartmanis–Berman conjecture [4]: sparse sets cannot be \leq_m^P -complete in NP unless $P = NP$. Karp, Lipton and Sipser [9] showed that sets having polynomial size circuits cannot be polynomial time Turing complete (abbr. \leq_T^P -complete) in NP unless the polynomial time hierarchy collapses to Σ_2^P . Note that by a result of Meyer (stated in [4]) NP has polynomial size circuits if and only if there is a sparse \leq_T^P -hard set for NP. Similar results can be found in [10], [15], [18].

In recursion theory, certain properties which are incompatible with completeness have been classified as “lowness” properties. More precisely, let A be a recursively enumerable (abbr. r.e.) set and $A^{(n)}$ its n th jump. Then A is a *high_n* set if $A^{(n)}$ is Turing equivalent to $\emptyset^{(n+1)}$, and A is a *low_n* set if $A^{(n)}$ is Turing equivalent to $\emptyset^{(n)}$ [19]. Intuitively, the highness or lowness of an r.e. set indicates the relative information content of the set. An interesting application of this information content classification of r.e. sets is the characterizations of many complexity-theoretic and structural properties found by Bennison [2], [3] and Soare [19]. For instance, subcreativity and effective speedability are shown to be “weak high” properties, and nonspeedability a “weak low” property. (Weak highness and weak lowness are defined to be similar to highness and lowness except that a “weak jump” is used. For the exact definitions, see [19].) Since complete sets are known to be *high₀* (and weak *high₀*), it is an immediate consequence that nonspeedable sets cannot be complete.

It seems natural then, based on the results of Mahaney and Karp, Lipton and Sipser, to draw the analogy in the NP theory, and ask whether there is a natural definition of highness and lowness of sets in NP and whether sparseness is indeed a lowness property. An affirmative answer to the first question has been given by Schöning [16]. He defined a high and a low hierarchy in NP, based on a K -operator which is an analogue of the jump operator in recursion theory. The naturalness of these

* Received by the editors June 11, 1982, and in revised form September 19, 1983. This research was supported in part by the National Science Foundation under grants MCS-8103479 and MCS-8011979, and by Deutsche Forschungsgemeinschaft.

† Department of Computer Science, University of Houston, Houston, Texas 77004.

‡ Institut für Informatik, Universität Stuttgart, D-7000 Stuttgart 1, West Germany. This paper was written while this author was visiting the Department of Mathematics, University of California, Santa Barbara, California 93106.

hierarchies can be seen from the simple characterizations of some classes of high and low sets. Let $L_0^p \subseteq L_1^p \subseteq \dots \subseteq LH$ and $H_0^p \subseteq H_1^p \subseteq \dots \subseteq HH$ be the low and high hierarchies in NP, respectively. (See the next section and [16] for the definitions.) Schöning showed that $L_0^p = P$, $L_1^p = NP \cap \text{co-NP}$, $H_0^p = \{\leq_T^p\text{-complete sets in NP}\}$, $H_1^p = \{\leq_T^{snp}\text{-complete sets in NP}\}$, where \leq_T^{snp} represents the strong nondeterministic polynomial time Turing reducibility [13].

One of the open questions about these hierarchies is that they are not known to be proper hierarchies. That is, we do not know that $H_n^p \neq H_{n+1}^p$ or $L_n^p \neq L_{n+1}^p$ —even under the strong assumption that $\Sigma_n^p \neq \Sigma_{n+1}^p$. All we know is that $H_n^p \cap L_n^p = \emptyset$ if and only if $\Sigma_n^p \neq \Sigma_{n+1}^p$. Still it appears to be an interesting classification of sets in NP. In this paper we use Schöning's low hierarchy to attack the question of whether sparseness is a lowness property. We provide an information content classification of some intuitive lowness properties and derive, from this classification, results similar to those of Karp, Lipton and Sipser.

To be more precise, we prove that sets in NP having polynomial size circuits are in L_3^p . Since NP-complete sets with respect to many natural reducibilities appear to be in the high hierarchy (cf. [16]), the above result not only separates sets having polynomial size circuits from \leq_T^p -complete sets in NP (under the assumption that $\Sigma_2^p \neq \Sigma_3^p$), but also separates them from NP-complete sets of weaker types, e.g., \leq_T^{snp} -complete sets in NP. In addition to the above result, we also show the lowness of some other sets including (i) sparse sets in NP, (ii) sets in $\text{APT} \cap \text{NP}$ (APT is the class of sets having deterministic algorithms which run in polynomial time for all inputs except those in a sparse set [15]), (iii) sets in R (the class of sets having polynomial time probabilistic algorithms with small one-sided errors [1]), and (iv) weakly p -selective sets in NP that include both p -selective sets [18] and left cuts of NP real numbers as subclasses [10]. Figure 1 shows the inclusion structure of low sets in NP.

In summary, the contribution of this paper is to give a unified method of proving incompatibility results. The main application of this method is to obtain results of the form “if a complete set for NP has property π , then the polynomial time hierarchy collapses to $\Sigma_{n(\pi)}^p$ (or $\Delta_{n(\pi)}^p$)” (cf. [5], [7], [9], [10], [12], [14], [18]). In the next section we review the high and low hierarchies defined by Schöning, and give the necessary notation. Then we show our main results in the following sections.

2. Notation. All our sets are subsets of $\{0, 1\}^*$. Let s be a string in $\{0, 1\}^*$. Then $|s|$ denotes the length of s . We use $\langle \cdot, \cdot \rangle$ to denote a pairing function and generalize it to encode a finite number of strings. If $s = \langle s_1, \dots, s_k \rangle$, then we define $\text{set}(s) = \{s_1, \dots, s_k\}$ and say the string s encodes the set $\{s_1, \dots, s_k\}$. We assume that for two given strings s and t , the predicate $t \in \text{set}(s)$ is polynomial time computable. For a set A , let $|A|$ denote its cardinality.

Let A be a set and C a class of sets. We let $P(A)$ and $\text{NP}(A)$ denote the classes of sets accepted by polynomial time deterministic and nondeterministic oracle machines with oracle A , respectively. Let $P(C) = \cup \{P(A) | A \in C\}$ and $\text{NP}(C) = \cup \{\text{NP}(A) | A \in C\}$. The relativized polynomial time hierarchy may be defined as follows [20]:

$$\Sigma_0^{p,A} = \Pi_0^{p,A} = \Delta_0^{p,A} = P(A),$$

and for $n \geq 1$,

$$\Sigma_n^{p,A} = \text{NP}(\Sigma_{n-1}^{p,A}),$$

$$\Pi_n^{p,A} = \text{co-NP}(\Sigma_{n-1}^{p,A}) = \text{co-}\Sigma_n^{p,A},$$

$$\Delta_n^{p,A} = P(\Sigma_{n-1}^{p,A}).$$

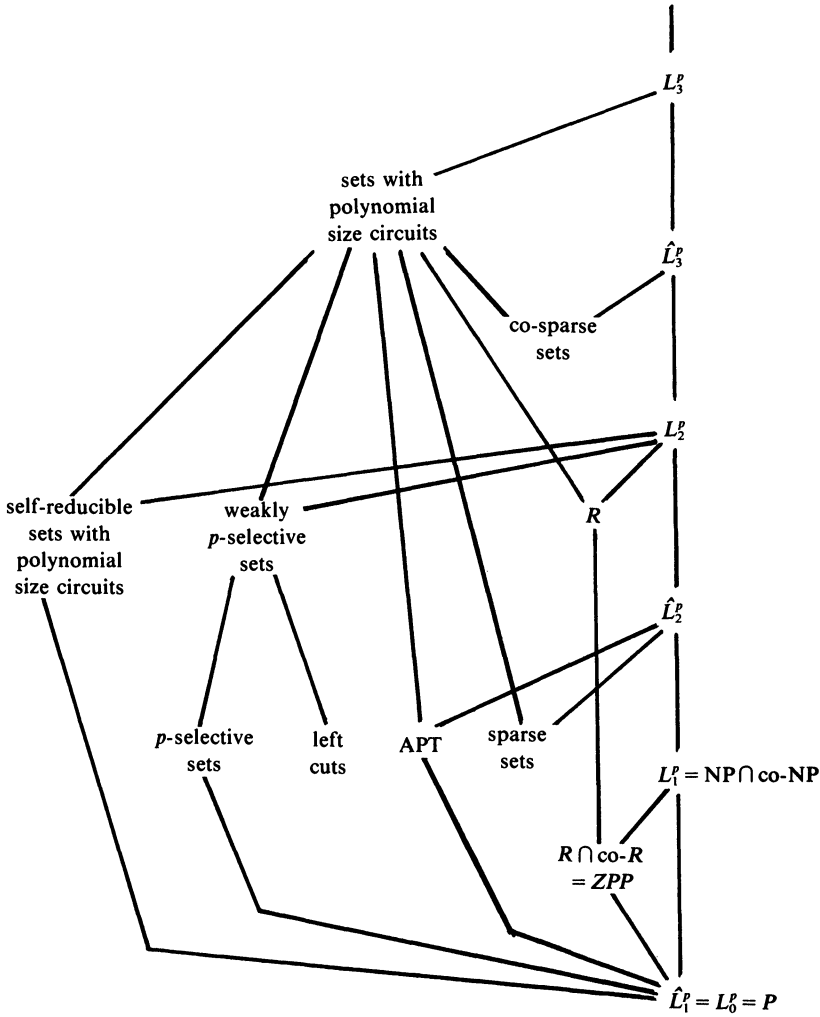


FIG. 1. Inclusion structure of low sets in NP.

When $A = \emptyset$, we write

$$\Sigma_n^p, \Pi_n^p \text{ and } \Delta_n^p \text{ for } \Sigma_n^{p,\emptyset}, \Pi_n^{p,\emptyset} \text{ and } \Delta_n^{p,\emptyset},$$

respectively. Furthermore, define

$$\text{PH} = \cup \{ \Sigma_n^p \mid n \geq 0 \}.$$

It is not known whether the polynomial hierarchy is a proper infinite hierarchy or whether it “collapses.” It is known that for $n \geq 1$, the hierarchy collapses to Σ_n^p ($\text{PH} = \Sigma_n^p$) if and only if $\Sigma_n^p = \Pi_n^p$, if and only if $\Sigma_n^p = \Sigma_{n+1}^p$, if and only if $\Pi_n^p = \Pi_{n+1}^p$, and $\text{PH} = \Delta_n^p$ if and only if $\Sigma_n^p = \Delta_n^p$, if and only if $\Pi_n^p = \Delta_n^p$, if and only if $\Delta_n^p = \Delta_{n+1}^p$.

Schöning used the K -operator to define the high and low hierarchies [16]. Here the following characterization is more useful.

DEFINITION 1. For $n \geq 0$, let $H_n^p = \{A \in \text{NP} \mid \Sigma_{n+1}^{p,A} \subseteq \Sigma_n^{p,A}\}$, and let $L_n^p = \{A \in \text{NP} \mid \Sigma_n^{p,A} \subseteq \Sigma_n^p\}$.

Observe that this characterization has been shown to be equivalent to that given in [16] with the exception of the trivial cases $A = \emptyset$ and $A = \{0, 1\}^*$, and the case of $n = 0$ for H_n^p . In [16], H_0^p is the class of \cong_m^p -complete sets in NP.

The relationship between these hierarchies is as follows:

LEMMA 1 [16]. *For all $n \geq 0$:*

- (a) $H_n^p \subseteq H_{n+1}^p, L_n^p \subseteq L_{n+1}^p$.
- (b) *If $PH \neq \Sigma_n^p$ then $L_n^p \cap H_n^p = \emptyset$.*
- (c) *If $PH = \Sigma_n^p$ then $L_n^p = H_n^p = NP$.*

The classes L_0^p, L_1^p and H_0^p, H_1^p have natural characterizations.

LEMMA 2 [16].

- (a) $L_0^p = P$, and $L_1^p = NP \cap \text{co-NP}$.
- (b) $H_0^p = \{\cong_T^p\text{-complete sets in NP}\}$, and $H_1^p = \{\cong_T^{SNP}\text{-complete sets in NP}\}$.

A set A is said to have *polynomial size circuits* if there exists a function $h: N \rightarrow \{0, 1\}^*$ and a set $B \in P$ such that:

- (a) for all $s \in \{0, 1\}^*$, $s \in A$ if and only if $\langle h(|s|), s \rangle \in B$; and
- (b) for some polynomial function p and all $n \geq 0$, $|h(n)| \leq p(n)$.

Let $A_n = \{s \in A \mid |s| \leq n\}$. A set A is *sparse* if there exists a polynomial function q such that $|A_n| \leq q(n)$ for all $n \geq 0$. By a result of Meyer in [4], a set A has polynomial size circuits if and only if $A \in P(S)$ for some sparse set S . A machine M is said to be *almost polynomial time* if there are a polynomial p and a sparse set B such that for all $s \notin B$, $M(s)$ runs in $p(|s|)$ steps. The class APT is the class of sets which are accepted by almost polynomial time machines. The class R is the class of sets A satisfying the following conditions: there exist a polynomial function p and a polynomial time computable predicate Q such that, for each n , if $|s| \leq n$ then:

- (a) if $s \in A$ then $|\{t \mid |t| \leq p(n) \text{ and } Q(s, t)\}| \geq \frac{1}{2} \cdot |\{t \mid |t| \leq p(n)\}|$;
- (b) if $s \notin A$ then $(\forall t, |t| \leq p(n)) [\text{not } Q(s, t)]$.

DEFINITION 2. A set A is *weakly p -selective* [10] if there is a function f of two arguments that can be computed in polynomial time, such that, for every $n \geq 0$, the set $\{x \in \{0, 1\}^* \mid |x| \leq n\}$ can be decomposed into at most $p(n)$ many pairwise disjoint subsets B_1, \dots, B_m , $m \leq p(n)$, for some polynomial p and

- (a) if x and y are in two different sets, $x \in B_i$ and $y \in B_j$, with $1 \leq i < j \leq m$, then $f(x, y) = f(y, x) = \#$ where $\#$ is a new symbol;
- (b) if x and y are in the same B_i , $1 \leq i \leq m$, then $f(x, y) = f(y, x) \in \{x, y\}$, and furthermore, if $x \in A$ or $y \in A$, then $f(x, y) = f(y, x) \in A$.

Ko shows in [10] that weakly p -selective sets include both p -selective sets and left cuts of real numbers.

It has been shown that sparse sets, sets in APT, sets in R , and weakly p -selective sets all have polynomial size circuits [4], [15], [1], [10].

Polynomial time many-one (\cong_m^p) and *polynomial time Turing* (\cong_T^p) *reducibilities* are defined in [11]. *Strong nondeterministic polynomial time Turing reducibility* (\cong_T^{SNP}) is defined in [13] and can be characterized as follows: $A \cong_T^{SNP} B$ if and only if $A \in NP(B) \cap \text{co-NP}(B)$.

We say that a predicate $T(x)$ has a Σ_n^p -form (or, Π_n^p -form) if it is of the form

$$(Q_1 x_1)(Q_2 x_2) \cdots (Q_n x_n) S(x, x_1, x_2, \dots, x_n),$$

where S is polynomial time computable, Q_1, Q_2, \dots, Q_n are alternating quantifiers starting with $Q_1 = \exists$ (or, $Q_1 = \forall$, respectively) and the variables x_i range over strings of lengths bounded by a polynomial in $|x|$. It is known [20] that a set S is in Σ_n^p (Π_n^p) if and only if the predicate expressing " $x \in S$ " has Σ_n^p -form (Π_n^p -form).

3. Polynomial size circuits and the low hierarchy. First we give some more notation. For any set B and any string w , let $B(w)$ denote the set $\{s \in \{0, 1\}^* \mid \langle w, s \rangle \in B\}$. That is, $B(w)$ is the set of strings which can be recognized with “circuits” w and “circuit interpreter” B . For any multi-valued function $h: N \rightarrow \{0, 1\}^*$, we let $\text{graph}(h)$ be the set $\{\langle 0^n, w \rangle \mid w \text{ is a value of } h(n)\}$. We say h is *total* if for each $n \geq 0$, there is a w such that $\langle 0^n, w \rangle \in \text{graph}(h)$, and h is *polynomial length-bounded* if there is a polynomial p such that for each $n \geq 0$, $|w| \leq p(n)$ for all values w of $h(n)$. For set $A \subseteq \{0, 1\}^*$, we define the class of sets

$$\text{CIR}(A) = \{\text{graph}(h) \mid h \text{ is total and polynomial length-bounded, and} \\ \text{there is a set } B \in P \text{ such that for each } n \geq 0, A_n = B(w)_n \\ \text{for all values } w \text{ of } h(n)\}.$$

In other words, $\text{CIR}(A)$ is the collection of all “polynomial size circuits” with which the membership questions of A can be answered in polynomial time. Note that A has polynomial size circuits if and only if $\text{CIR}(A) \neq \emptyset$. Furthermore, if A has polynomial size circuits, then it has circuits of complexity Π_1^P relative to A .

PROPOSITION 1. *For each set A having polynomial size circuits, $\text{CIR}(A) \cap \Pi_1^{P,A} \neq \emptyset$.*

Proof. Let $D = \text{graph}(h)$ be in $\text{CIR}(A)$ via some $B \in P$. Define a set C as $C = \{\langle 0^n, w \rangle \mid |w| \leq p(n) \text{ and } A_n = B(w)_n\}$. Observe that $D \subseteq C$ and $C \in \text{CIR}(A)$. Further, by the definition of C ,

$$\langle 0^n, w \rangle \in C \quad \text{iff} \quad (\forall x, |x| \leq n)[x \in A \leftrightarrow \langle x, w \rangle \in B],$$

hence $C \in \Pi_1^{P,A}$. \square

Conversely the complexity of sets in $\text{CIR}(A)$ determines that of sets in the polynomial time hierarchy relative to A .

THEOREM 1. *Let A have polynomial size circuits. Assume that $\text{CIR}(A) \cap \Sigma_k^P \neq \emptyset$ for some $k \geq 1$. Then, $\Sigma_k^{P,A} \subseteq \Sigma_k^P$.*

Proof. Let $D \in \Sigma_k^{P,A}$. Then there exist a polynomial function α and a predicate S in $P(A)$ such that for each x in $\{0, 1\}^*$,

$$x \in D \quad \text{iff} \quad (\exists y_1, |y_1| \leq \alpha(|x|)) \cdots (Q_k y_k, |y_k| \leq \alpha(|x|)) S(x, y_1, \cdots, y_k),$$

where $Q_k = \exists$ if k is odd and $Q_k = \forall$ if k is even. Since $S \in P(A)$, there exists an oracle machine M_S such that, for inputs (x, y_1, \cdots, y_k) of sizes $|x| = n$ and $|y_1|, \cdots, |y_k| \leq \alpha(n)$ and with oracle A , M_S accepts (x, y_1, \cdots, y_k) in $\beta(n)$ steps for some polynomial function β if and only if $S(x, y_1, \cdots, y_k)$.

Let $C \in \text{CIR}(A) \cap \Sigma_k^P$. Assume that $C = \text{graph}(h) \in \text{CIR}(A)$ via some set $B \in P$ and some polynomial bound γ . Define a polynomial time computable predicate T as follows. $T(x, y_1, \cdots, y_k, w) = 1$ if and only if the oracle machine M_S accepts (x, y_1, \cdots, y_k) with the oracle $B(w)$. Note that if $\langle 0^{\beta(n)}, w \rangle \in C$ then $A_{\beta(n)} = B(w)_{\beta(n)}$ and hence for all x, y_1, \cdots, y_k , with $|x| = n$ and $|y_1|, \cdots, |y_k| \leq \alpha(n)$, $T(x, y_1, \cdots, y_k, w) = S(x, y_1, \cdots, y_k)$ because the machine M_S on (x, y_1, \cdots, y_k) can query strings of lengths at most $\beta(n)$.

Now we can reduce the predicate “ $x \in D$ ” as follows. Let $|x| = n$.

$$x \in D \\ \text{iff } (\exists y_1, |y_1| \leq \alpha(n)) \cdots (Q_k y_k, |y_k| \leq \alpha(n)) S(x, y_1, \cdots, y_k) \\ \text{iff } (\exists w, |w| \leq \gamma(\beta(n))) [\langle 0^{\beta(n)}, w \rangle \in C \text{ and} \\ (\exists y_1, |y_1| \leq \alpha(n)) \cdots (Q_k y_k, |y_k| \leq \alpha(n)) S(x, y_1, \cdots, y_k)].$$

This is valid because A has polynomial size circuits and h is total. From our discussion above about the relationship between the predicates S and T , we can further reduce the predicate “ $x \in D$ ” to the following:

$$(\exists w, |w| \leq \gamma(\beta(n)))[\langle 0^{\beta(n)}, w \rangle \in C \text{ and} \\ (\exists y_1, |y_1| \leq \alpha(n)) \cdots (Q_k y_k, |y_k| \leq \alpha(n)) T(x, y_1, \dots, y_k, w)].$$

It is clear now from the above reduced predicate that $C \in \Sigma_k^p$ implies $D \in \Sigma_k^p$, and the proof is completed. \square

We now study the complexity of $\text{CIR}(A)$ for several subclasses of sets with polynomial size circuits.

LEMMA 3. *Let A have polynomial size circuits and let $k \geq 1$.*

- (a) *If $A \in \Sigma_k^p$ then $\text{CIR}(A) \cap \Pi_{k+1}^p \neq \emptyset$.*
- (b) *If $A \in R$ then $\text{CIR}(A) \cap \Pi_1^p \neq \emptyset$.*
- (c) *If $A \in \Sigma_k^p$ and A is weakly p -selective then $\text{CIR}(A) \cap \Delta_{k+1}^p \neq \emptyset$.*

Proof. (a) Immediate from Proposition 1.

(b) Let $A \in R$. Then there are a polynomial function β and a polynomial time computable predicate T such that $(\forall n)(\forall s, |s| \leq n) [[s \in A \text{ implies } \{|t| \mid |t| \leq \beta(n) \text{ and } T(s, t)\} \cong \frac{1}{2} \cdot \{|t| \mid |t| \leq \beta(n)\}]$ and $[s \notin A \text{ implies } (\forall t, |t| \leq \beta(n)) \text{ not } T(s, t)]$. Moreover, Adleman [1] observed that for each n , we can find a set $W_n \subseteq \{|t| \mid |t| \leq \beta(n)\}$ and $(\exists s, |s| \leq n) T(s, t)$ of size $\leq n$ such that $(\forall s, |s| \leq n) [s \in A \text{ iff } (\exists t \in W_n) T(s, t)]$.

Define $B = \{\langle w, s \rangle \mid (\exists t \in \text{set}(w)) T(s, t)\}$ and h a multi-valued function having graph $(h) = \{\langle 0^n, w \rangle \mid |w| \leq n \cdot \beta(n) \text{ and } A_n = B(w)_n\}$. Then $h \in \text{CIR}(A)$ because $B \in P$ and for each n the string x_n which encodes the set W_n satisfies $\langle 0^n, x_n \rangle \in \text{graph}(h)$. We claim that $\text{graph}(h) \in \Pi_1^p$.

We reduce the predicate $A_n = B(w)_n$ as follows.

$$\begin{aligned} A_n = B(w)_n \\ \text{iff } (\forall s, |s| \leq n) [s \in A \leftrightarrow \langle w, s \rangle \in B] \\ \text{iff } (\forall s, |s| \leq n) [(\exists t, |t| \leq \beta(n)) T(s, t) \leftrightarrow (\exists u \in \text{set}(w)) T(s, u)] \\ \text{iff } (\forall s, |s| \leq n) [(\exists t, |t| \leq \beta(n)) T(s, t) \rightarrow (\exists u \in \text{set}(w)) T(s, u)] \\ \text{iff } (\forall s, |s| \leq n) (\forall t, |t| \leq \beta(n)) [T(s, t) \rightarrow \langle w, s \rangle \in B]. \end{aligned}$$

It is thus a Π_1^p -form predicate, and hence $\text{graph}(h) \in \Pi_1^p$.

(c) Let A be weakly p -selective. Then there is a function f of two arguments that can be computed in polynomial time and satisfies the conditions in the definition of weak p -selectivity. Following [10, Thm. 3], for some polynomials β and γ and for each n , there exists a set $W_n \subseteq A_{\beta(n)}$ of cardinality at most $\gamma(n)$, such that $(\forall s, |s| \leq n) [s \in A \text{ iff } (\exists y \in W_n) f(y, s) = s]$.

Let $B = \{\langle u, s \rangle \mid (\exists t \in \text{set}(u)) f(t, s) = s\}$ and $p(n) = \beta(n) \cdot \gamma(n)$. Then $B \in P$. Define $C = \{\langle 0^n, w \rangle \mid |w| \leq p(n), \text{set}(w) \subseteq A \text{ and } A_n = B(w)_n\}$. Then the function h such that $\text{graph}(h) = C$ is total because for each n , the string x_n which encodes the set W_n satisfies $A_n = B(x_n)_n$. That is, $C \in \text{CIR}(A)$. We claim that $C \in \Delta_{k+1}^p$ if $A \in \Sigma_k^p$.

First observe that $\text{set}(w) \subseteq A$ implies $(\forall s)[\langle w, s \rangle \in B \rightarrow s \in A]$ because, by the definition of weak p -selectivity, $[u \in A \text{ and } f(u, s) = s]$ implies $s \in A$.

Now we reduce the predicate $\langle 0^n, w \rangle \in C$ as follows.

$$\begin{aligned} \langle 0^n, w \rangle \in C \\ \text{iff } \text{set}(w) \subseteq A \text{ and } (\forall s, |s| \leq n) [s \in A \leftrightarrow \langle w, s \rangle \in B] \\ \text{iff } \text{set}(w) \subseteq A \text{ and } (\forall s, |s| \leq n) [\langle w, s \rangle \in B \rightarrow s \in A] \\ \quad \text{and } (\forall s, |s| \leq n) [s \in A \rightarrow \langle w, s \rangle \in B] \\ \text{if } \text{set}(w) \subseteq A \text{ and } (\forall s, |s| \leq n) [s \in A \rightarrow \langle w, s \rangle \in B]. \end{aligned}$$

Since $A \in \Sigma_k^p$, set $(w) \subseteq A$ is a Σ_k^p -predicate, and $(\forall s, |s| \leq n)[s \in A \rightarrow \langle w, s \rangle \in B]$ is a Π_k^p -predicate. Hence C is the intersection of a set in Σ_k^p and a set in Π_k^p . It follows that $C \in \Delta_{k+1}^p$. \square

Combining Theorem 1, Lemma 3 and the observation that for each $k \geq 0$, Π_k^p and Δ_{k+1}^p are included in Σ_{k+1}^p , we have the following:

THEOREM 2. (a) $\{A \in \text{NP} \mid A \text{ has polynomial size circuits}\} \subseteq \Sigma_2^p$.

(b) $R \subseteq \Sigma_2^p$.

(c) $\{A \in \text{NP} \mid A \text{ is weakly } p\text{-selective}\} \subseteq \Sigma_2^p$.

COROLLARY 1. (a) $\{A \in \text{NP} \mid A \text{ is } p\text{-selective}\} \subseteq \Sigma_2^p$.

(b) $\{A \in \text{NP} \mid A \text{ is a left cut of a real number}\} \subseteq \Sigma_2^p$.

Proof. Corollary 1 follows immediately from Theorem 2(c) and the results in [10] that p -selective sets and left cuts are weakly p -selective. \square

Observe that, by the same proof methods, it is possible to obtain results analogous to Theorems 2(b) and 2(c) for the classes $\{A \in \text{NP} \mid A \text{ is sparse}\}$ and $\text{APT} \cap \text{NP}$. However, for these classes we are able to obtain even stronger results in the next section.

COROLLARY 2. (a) *A set having polynomial size circuits cannot be \leq_T^{SNP} -complete in NP unless $\text{PH} = \Sigma_3^p$.*

(b) *A set which is in R (or weakly p -selective) cannot be \leq_T^{SNP} -complete in NP unless $\text{PH} = \Sigma_2^p$.*

Proof. Follows immediately from Lemma 2 and Theorem 2. \square

Comparing Corollary 2 with Karp, Lipton and Sipser's [9] result that if A has polynomial size circuits and A is \leq_T^p -complete in NP then $\text{PH} = \Sigma_2^p$, our result has a weaker conclusion as well as a weaker hypothesis. The reason we cannot show, in Corollary 2(a), that $\text{PH} = \Sigma_2^p$ is that a \leq_T^{SNP} -complete set in NP does not necessarily have the self-reducible property [15] which is a critical condition in the proof of Karp, Lipton and Sipser's result. In fact, with the help of self-reducibility, we can derive the stronger conclusion that $\text{PH} = \Sigma_2^p$. We call a set A *self-reducible* if there is an oracle machine M such that it, with the oracle A , accepts A in polynomial time, and on input x , M queries its oracle only about the strings of lengths less than $|x|$. This simple definition of self-reducibility captures the essential idea of the seemingly more general one given in [10], [15].

COROLLARY 3. *Assume $A \in \text{NP}$ has polynomial size circuits and is self-reducible. Then $A \in \Sigma_2^p$. As a consequence, A cannot be \leq_T^{SNP} -complete for NP unless $\text{PH} = \Sigma_2^p$.*

Proof. Let M be the polynomial time oracle machine that witnesses A 's self-reducibility. It suffices to follow the proof of Lemma 3(a) and show that the predicate $A_n = B(w)_n$ is equivalent to a Π_1^p -form predicate. Indeed we claim that

$$\begin{aligned} A_n = B(w)_n & \\ \text{iff } (\forall s, |s| \leq n)[s \in A \leftrightarrow \langle w, s \rangle \in B] & \\ \text{iff } (\forall s, |s| \leq n)[M \text{ accepts } s \text{ with oracle } B(w) \leftrightarrow \langle w, s \rangle \in B]. & \end{aligned}$$

First we observe that

$$(*) \quad [A_k = B(w)_k \text{ and } |s| = k + 1] \text{ implies } [s \in A \leftrightarrow M \text{ accepts } s \text{ with oracle } B(w)],$$

because M on s queries only strings of length $\leq k$.

The forward direction of the claim then follows from the observation immediately. Conversely, the backward direction can be proved by induction. That is, assume that $(\forall s, |s| \leq n)[M \text{ accepts } s \text{ with oracle } B(w) \leftrightarrow \langle w, s \rangle \in B]$, then we show that $A_k = B(w)_k$ for $k = 0, 1, \dots, n$. The basic step follows from the fact that M on s , $|s| = 0$, does not query any string. The inductive step follows directly from the above observation (*).

Thus $A_n = B(w)_n$ is equivalent to a Π_1^p -form predicate and it follows from Theorem 1 that $A \in L_2^p$. \square

Observe that Karp, Lipton and Sipser's result follows immediately from Corollary 3. Whether or not the hypothesis that A is self-reducible can be dropped is an interesting open question.

4. Refinement of the high and low hierarchies. The material in this section is essentially from an earlier paper [17]. First, we give a definition of a new kind of low and high hierarchy within NP.

DEFINITION 3. For each $n \geq 1$,

$$\hat{L}_n^p = \{A \in \text{NP} \mid \Delta_n^{p,A} \subseteq \Delta_n^p\}, \quad \hat{H}_n^p = \{A \in \text{NP} \mid \Delta_{n+1}^p \subseteq \Delta_n^{p,A}\}.$$

Note that $\hat{L}_1^p = L_0^p = \text{P}$ and $\hat{H}_1^p = H_0^p = \{\leq_T^p\text{-complete sets in NP}\}$. The analogue of this definition in recursive function theory coincides exactly with the analogue of Definition 1. Again, in the context of NP complexity and the polynomial time hierarchy, it is not clear whether these hierarchies coincide; that is, whether Definitions 1 and 3 are equivalent. The relationship between these hierarchies is as follows:

LEMMA 4. For all $n \geq 1$:

- (a) $L_{n-1}^p \subseteq \hat{L}_n^p \subseteq L_n^p$.
- (b) $H_{n-1}^p \subseteq \hat{H}_n^p \subseteq H_n^p$.
- (c) if $\text{PH} \neq \Delta_n^p$ then $\hat{L}_n^p \cap \hat{H}_n^p = \emptyset$.
- (d) if $\text{PH} = \Delta_n^p$ then $\hat{L}_n^p = \hat{H}_n^p = \text{NP}$.

Proof. (a) and (b) follow immediately from the definition.

Suppose $A \in \hat{L}_n^p \cap \hat{H}_n^p$. Then it follows that $\Delta_{n+1}^p \subseteq \Delta_n^{p,A} \subseteq \Delta_n^p$, and hence $\text{PH} = \Delta_n^p$. This proves (c).

Now suppose $\text{PH} = \Delta_n^p$, hence $\Delta_n^p = \Delta_{n+1}^p$. Let A be an arbitrary set in NP. Then $\Delta_n^{p,A} \subseteq \Delta_{n+1}^p = \Delta_n^p$ and hence $A \in \hat{L}_n^p$. Further, $\Delta_{n+1}^p = \Delta_n^p \subseteq \Delta_n^{p,A}$ and hence $A \in \hat{H}_n^p$. This proves (d). \square

Recall that A has polynomial size circuits if and only if $\text{CIR}(A) \neq \emptyset$, where $\text{CIR}(A)$ is a collection of graphs of multi-valued functions. In Theorem 1, we showed that if the graph of a multi-valued function is in $\text{CIR}(A)$ and can be checked in time Σ_k^p , and A is in NP, then A is in L_k^p . In the next theorem, we show a similar result that if there is a single-valued function h such that $\text{graph}(h) \in \text{CIR}(A)$ and h is in Δ_{k+1}^p (i.e., h can be evaluated (cf. [21]) in polynomial time relative to an oracle in Σ_k^p with the inputs in unary notation), then $\Delta_{k+1}^{p,A} \subseteq \Delta_{k+1}^p$. Thus, if we assume that $A \in \text{NP}$ then it follows that $A \in \hat{L}_{k+1}^p$.

THEOREM 3. Let A have polynomial size circuits. Assume that there is a (single-valued) function h such that $\text{graph}(h) \in \text{CIR}(A)$ and $h \in \Delta_{k+1}^p$ for some $k \geq 1$. Then, $\Delta_{k+1}^{p,A} \subseteq \Delta_{k+1}^p$.

Proof. It suffices to show that $\Sigma_k^{p,A} \subseteq \Delta_{k+1}^p$, since $\text{P}(\Delta_{k+1}^p) = \Delta_{k+1}^p$.

Let $D \in \Sigma_k^{p,A}$. Then proceed as in the proof of Theorem 1, and obtain a polynomial time computable predicate T . Then the set

$$E = \{\langle w, x \rangle \mid (\exists y_1, |y_1| \leq \alpha(|x|)) \cdots (Q_k y_k, |y_k| \leq \alpha(|x|)) T(x, y_1, \dots, y_k, w)\}$$

is in Σ_k^p .

Note that $D = \{x \mid \langle h(0^{\beta(|x|)}), x \rangle \in E\}$. Therefore D is in Δ_{k+1}^p , because $E \in \Sigma_k^p$ and $h \in \Delta_{k+1}^p$. \square

For several classes of sets related to the notion of sparseness, we can show the existence of functions which satisfy the conditions in Theorem 3. First we establish a

lemma on the complexity of circuits of sparse sets. The proof technique is similar to that in [12] and [14].

LEMMA 5. *Let D be a sparse set in Σ_k^p for some $k \geq 1$. Then there is a function $h: \{0\}^* \rightarrow \{0, 1\}^*$ in Δ_{k+1}^p such that $(\forall n)$ $[\text{set}(h(0^n)) = D_n]$.*

Proof. First, for a given set D define $\text{PREFIX}(D) = \{\langle x, 0^n \rangle \mid (\exists y, |y| = n) [y \in D \text{ and } x \text{ is a prefix of } y]\}$. Observe that if $D \in \Sigma_k^p$ for some $k \geq 1$, then $\text{PREFIX}(D) \in \Sigma_k^p$.

The following deterministic oracle machine computes a function $t: \{0\}^* \rightarrow \{0, 1\}^*$, using as oracle $\text{PREFIX}(D)$, such that for each n , set $(t(0^n)) = \{x \in D \mid |x| = n\}$.

On input 0^n :

begin

put the empty string into an initially empty list;

$T := \emptyset$;

while list not empty **do**

begin

$z :=$ first word on the list;

cancel z from the list;

if $|z| = n$ **and** $\langle z, 0^n \rangle \in \text{PREFIX}(D)$ **then** $T := T \cup \{z\}$;

if $\langle z0, 0^n \rangle \in \text{PREFIX}(D)$ **then** put $z0$ into the list;

if $\langle z1, 0^n \rangle \in \text{PREFIX}(D)$ **then** put $z1$ into the list;

end {while};

output the encoding of T ;

end.

Observe that the machine runs in polynomial time relative to $\text{PREFIX}(D)$ if D is sparse. By applying the above machine on inputs $0, 0^2, \dots, 0^n$, we get a function h with set $(h(0^n)) = D_n$, and $h \in \Delta_{k+1}^p$. \square

LEMMA 6. *Let A have polynomial size circuits and $A \in \Sigma_k^p$, $k \geq 1$. Then, there exists a function h such that $\text{graph}(h) \in \text{CIR}(A)$ and*

(a) $h \in \Delta_{k+1}^p$ if A is sparse;

(b) $h \in \Delta_{k+1}^p$ if $A \in \text{APT}$;

(c) $h \in \Delta_{k+2}^p$ if A is co-sparse.

Proof. (a) Let h be a function such that $\text{set}(h(0^n)) = A_n$, and $B = \{\langle w, s \rangle \mid s \in \text{set}(w)\}$. Then $B \in P$, and for some polynomial p , $|h(0^n)| \leq p(n)$. Therefore, $\text{graph}(h) \in \text{CIR}(A)$. Since A is sparse and $A \in \Sigma_k^p$, it follows from Lemma 5 that $h \in \Delta_{k+1}^p$.

(b) Since $A \in \text{APT}$, there exists a deterministic Turing machine M that accepts A , and a polynomial p such that the set $D = \{s \mid M(s) \text{ does not halt in } p(|s|) \text{ moves}\}$ is sparse. Note that $D \in P$. Define $B = \{\langle w, s \rangle \mid s \in \text{set}(w) \text{ or } M \text{ accepts } s \text{ in } p(|s|) \text{ moves}\}$, and $h: \{0\}^* \rightarrow \{0, 1\}^*$ such that $\text{set}(h(0^n)) = (A \cap D)_n$ for all n . Then $B \in P$ and, for each n , $|h(0^n)| \leq q(n)$ for some polynomial q . Therefore, $\text{graph}(h) \in \text{CIR}(A)$. Since $A \in \Sigma_k^p$ and $D \in P$, we have $A \cap D \in \Sigma_k^p$ and hence, by Lemma 5, $h \in \Delta_{k+1}^p$.

(c) Similar to (a) except that we use h to encode \bar{A} instead of A . Since $\bar{A} \in \Pi_k^p \subseteq \Sigma_{k+1}^p$, we have $h \in \Delta_{k+2}^p$. \square

Theorem 3 and Lemma 6 yield the following:

THEOREM 4. (a) $\{A \in \text{NP} \mid A \text{ is sparse}\} \subseteq \hat{L}_2^p$.

(b) $\text{APT} \cap \text{NP} \subseteq \hat{L}_2^p$.

(c) $\{A \in \text{NP} \mid A \text{ is co-sparse}\} \subseteq \hat{L}_3^p$.

COROLLARY 4. (a) $A \not\equiv_T^{sp}$ -complete set in NP cannot be sparse (or in APT), unless $\text{PH} = \Delta_2^p$.

(b) $A \not\equiv_T^{sp}$ -complete set in NP cannot be co-sparse unless $\text{PH} = \Delta_3^p$.

Corollary 4(a) is stronger than Mahaney's result in [14] that a \leq_T^p -complete set in NP cannot be sparse unless $\text{PH} = \Delta_2^p$.

As for the case of co-sparse sets, we do not know whether co-sparse sets in NP are in \hat{L}_2^p (or at least in L_2^p). Long [12] has proved that a \leq_T^p -complete set cannot be co-sparse unless $\text{PH} = \Delta_2^p$. However, Long's proof used, again, the self-reducibility structure which is apparently not available here. It has recently been noticed that there is an oracle set A such that there are co-sparse sets in $\text{NP}^A - \text{P}^A$ but there are no sparse sets in $\text{NP}^A - \text{P}^A$ [8]. This result suggests that sparse sets and co-sparse sets in NP are not symmetric, and it gives a partial explanation of our inability to show that co-sparse sets in NP are in \hat{L}_2^p .

Theorem 4(a) can be used to yield some separation results for the low hierarchy. Let EXPTIME (NEXPTIME) be the class of sets that can be accepted in time 2^{cn} , $c \geq 0$, by a deterministic (nondeterministic) Turing machine. By an easy padding argument (see [6]), it can be seen that if $\text{EXPTIME} \neq \text{NEXPTIME}$, then there exist tally (hence sparse) sets $\text{NP} - \text{P}$, and similarly, if $\text{NEXPTIME} \neq \text{co-NEXPTIME}$, then there exist tally sets in $\text{NP} - \text{co-NP}$.

COROLLARY 5. (a) *If $\text{EXPTIME} \neq \text{NEXPTIME}$, then $\text{P} = \text{L}_0^p = \hat{L}_1^p \neq \hat{L}_2^p$.*

(b) *If $\text{NEXPTIME} \neq \text{co-NEXPTIME}$, then $\text{NP} \cap \text{co-NP} = \text{L}_1^p \neq \hat{L}_2^p$.*

Hartmanis, Sewelson and Immerman [8] have recently shown that there exist sparse sets in $\text{NP} - \text{P}$ if and only if $\text{EXPTIME} \neq \text{NEXPTIME}$. This suggests the following open question: Does the converse of Corollary 5 hold?

Acknowledgment. The authors would like to thank Ron Rook for his encouragement and the reading of an earlier version of the manuscript.

REFERENCES

- [1] L. ADLEMAN, *Two theorems on random polynomial time*, Proc. 19th IEEE Symposium on the Foundations of Computer Science, 1978, pp. 75–83.
- [2] V. L. BENNISON, *Some lowness properties and computational complexity sequences*, Theoret. Comput. Sci., 6 (1978), pp. 233–254.
- [3] ———, *Information content characterizations of complexity theoretic properties*, Proc. 4th GI Conference on Theoretical Computer Science, Lecture Notes in Computer Science 67, Springer-Verlag, New York, 1978, pp. 58–66.
- [4] L. BERMAN AND J. HARTMANIS, *On isomorphisms and density of NP and other complete sets*, this Journal, 6 (1977), pp. 305–322.
- [5] P. BERMAN, *Relationship between density and deterministic complexity of NP-complete languages*, Proc. 5th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science, 62, Springer-Verlag, New York, 1978, pp. 63–71.
- [6] R. V. BOOK, *Tally languages and complexity classes*, Inform. Contr., 26 (1974), pp. 186–193.
- [7] S. FORTUNE, *A note on sparse complete sets*, this Journal, 8 (1979), pp. 431–433.
- [8] J. HARTMANIS, V. SEWELSON AND N. IMMERMANN, *Sparse sets in $\text{NP} - \text{P}$: EXPTIME versus NEXPTIME*, Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 382–391.
- [9] R. KARP AND R. LIPTON, *Some connections between nonuniform and uniform complexity classes*, Proc. 12th ACM Symposium on Theory of Computing, 1980, pp. 302–309.
- [10] K. KO, *On self-reducibility and weak p -selectivity*, J. Comput. System Sci., 26 (1983), pp. 209–221.
- [11] R. E. LADNER, N. A. LYNCH AND A. L. SELMAN, *A comparison of polynomial time reducibilities*, Theoret. Comput. Sci., 1 (1975), pp. 103–123.
- [12] T. J. LONG, *A note on sparse oracles for NP*, J. Comput. System Sci., 24 (1982), pp. 224–232.
- [13] ———, *Strong nondeterministic polynomial time reducibilities*, Theoret. Comput. Sci., 21 (1982), pp. 1–25.
- [14] S. R. MAHANEY, *Sparse complete sets of NP: solution of a conjecture by Berman and Hartmanis*, J. Comput. System Sci., 25 (1982), pp. 130–143.

- [15] A. R. MEYER AND M. S. PATERSON, *With what frequency are apparently intractable problems difficult?*, MIT/LCS/TM-126, Lab for Computer Science, 1979 Massachusetts Institute of Technology, Cambridge.
- [16] U. SCHÖNING, *A low and a high hierarchy within NP*, J. Comput. System Sci., 27 (1983), pp. 14–28.
- [17] ———, *On sparse and strong nondeterministic Turing complete sets in NP*, unpublished manuscript.
- [18] A. L. SELMAN, *P-selective sets, tally languages and the behavior of polynomial time reducibilities on NP*, Math. System Theory, 13 (1979), pp. 55–65.
- [19] R. I. SOARE, *Computational complexity, speedability and levelable sets*, J. Symbolic Logic, 42 (1977), pp. 545–563.
- [20] L. J. STOCKMEYER, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1–22.
- [21] L. VALIANT, *Relative complexity of checking and evaluating*, Inform. Proc. Letters, 5 (1976), pp. 20–23.

THE COMPLEXITY OF DISTRIBUTED CONCURRENCY CONTROL*

PARIS C. KANELLAKIS† AND CHRISTOS H. PAPADIMITRIOU‡

Abstract. We present a formal framework for distributed databases, and we study the complexity of the concurrency control problem in this framework. Our transactions are partially ordered sets of actions, as opposed to the straight-line programs of the centralized case. The concurrency control algorithm, or scheduler, is itself a distributed program. Three notions of performance of the scheduler are studied and interrelated: (1) its parallelism, (2) the computational complexity of the problems it needs to solve and (3) the cost of communication between the various parts of the scheduler. We show that the number of messages necessary and sufficient to support a given level of parallelism is equal to the minimax value of a combinatorial game. We show that this game is PSPACE-complete. It follows that, unless $NP = PSPACE$, a scheduler cannot simultaneously minimize communication and be computationally efficient. This result, we argue, captures the quantum jump in complexity of the transition from centralized to distributed concurrency control problems.

Key words. distributed database, concurrency control, games, complexity, PSPACE-complete

1. Introduction. There is now considerable literature, both theoretical and applied, concerning the *database concurrency control problem*—that is, maintaining the integrity of a database in the face of concurrent updates. Most of the theoretical work so far has been concerned with the *centralized* problem, in which the database resides at one site, and the update requests are submitted to a single process, called the *scheduler*, which implements the concurrency control policy of the database [4], [8], [11], [15], [17], [18]. There is also some interesting applied work on *distributed* databases [1], [2], [13], [16]. It is often said that the concurrency control problem is much trickier and harder in the distributed case than in the centralized case. This is evidenced by the existing solutions, which are extremely complex and sometimes incorrect.

In this paper we present a model of distributed databases, which captures the intricacies of distributed computation that are most pertinent to the database domain. Some novelties of our model are:

(a) *Transactions* are *partial orders* of atomic steps, thus generalizing the straight-line programs of the centralized case [8]. The partial order corresponds to both time-precedence and information flow, and it captures the notion of *distributed time* [10].

(b) The *scheduler*, the concurrency control agent of the system, is itself a *distributed program*, consisting of communicating sequential processes [6], one for each site.

(c) *Redundancy* (the requirement that two entities stored at different sites be copies of the same “virtual entity”) is not treated at the syntactic level, but is considered as part of the integrity constraints of the database. Redundancy was at the root of the complexities of most previous attempts to formalize distributed databases.

* Received by the editors October 28, 1983. This research was supported by the National Science Foundation under grants ECS-79-19880 and MCS-79-08965.

† Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. Present address, Department of Computer Science, Brown University, Providence, Rhode Island 02912.

‡ Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, and National Technical University of Athens, Athens, Greece.

As a consequence, there are *three measures of performance* in a distributed database (centralized theory deals with the first two):

- (1) *parallelism*, measured by the set of allowable interleavings of atomic steps,
- (2) *complexity* of the computational problems that the scheduler must solve,
- (3) *communication*, measured as number of messages exchanged by scheduler processes.

There are some interesting tradeoffs here. For example, let us fix (1) (think of it as the parallelism specifications of the system). By expending many messages, we can reduce the problem of distributed concurrency control to the centralized one (by broadcasting each request) and thus solve it in polynomial time for most reasonable parallelism specifications [11]. It turns out that, based on a priori information about transactions, we can minimize the number of messages sent in exponential time (and polynomial space; this is the upper bound of our main result). Finally we cannot have a scheduler simultaneously using the minimum number of messages and running in polynomial time at each site, unless $NP = PSPACE$ (this follows from the lower bound).

Specifically our main result states that: *for a certain parallelism specification, which in fact can be fixed to be the popular serializability principle, minimizing communication costs is a computational problem complete for PSPACE* [3], [5], [14]. Thus, our result appears to be concrete mathematical evidence suggesting that distributed concurrency control is indeed an inherently more complex problem than centralized concurrency control (under quite general conditions, centralized schedulers can be implemented in polynomial time and always in nondeterministic polynomial time [11], [15], [17], [18]).

Our result also adds to the literature on *distributed computation*, independently of its database context. It states, loosely speaking, that one cannot tell efficiently whether distributed processes can cooperate successfully for performing an (otherwise easy) *on-line* computational task, at fixed communication cost. It can therefore be considered as complementing the result of Ladner for lockout properties of “antagonistic” processes [9]. On the other hand, A. Yao has asked [19] whether minimizing communication costs for some distributed combinatorial computation is computationally intractable; NP-complete for the *off-line* problem. We answer this question for *on-line* computation. Yao’s original conjecture was recently answered in the affirmative [12].

We provide both upper and lower bounds. For the upper bound, we need a characterization (Theorem 3) of the incomplete executions of transactions that can be completed within a fixed number of messages. This upper bound holds for most parallelism specifications that can be achieved efficiently in a centralized manner. For the lower bound we relate distributed scheduling to a game played on graphs (the conflict graph of the transactions). Intuitively one player (Player I) is an adversary who submits update requests so as to force the scheduler to use as many messages as possible, whereas the other player (Player II) is the distributed scheduler. Player I wants to prolong the game as much as possible, whereas Player II tries to bring it to an end as soon as possible; other than that there is no winner or loser. The rules are related in a simple way to the cycles of the graph. The minimax length of the game corresponds to the optimal communication cost. We prove that this game is complete for PSPACE, and then show that our constructs can faithfully reflect a special kind of distributed concurrency control situation. This new kind of game may be of independent interest.

Section 2 describes the model used, § 3 the upper bound and the game on graphs, and finally § 4 has the PSPACE-completeness reduction (Theorem 4) and its implications.

2. A model of distributed database concurrency control.

2.1. Distributed database. A distributed database is a collection of sites. Each site has its own processor and data. The sites are interconnected by a network and are controlled by a distributed database management system (DDBMS). In Fig. 1 we show the architecture of a two-site system; distributed programs on this system consist of communicating sequential processes [6], one for each site, (horizontal arrows join parts of the same distributed program). Formally, a distributed database is defined as follows:

DEFINITION 1. A *distributed database* (DD) is a triple $\langle G, D, \text{stored-at} \rangle$ where:

(a) $G = (U, L)$ is an undirected graph, where every node corresponds to a *site* and every link to a two-way communication link between sites.

(b) D is a set of *entities*, denoted $\{x, y, z, \dots\}$.

(c) $\text{stored-at}: D \rightarrow U$ is a function determining the site, where each entity is stored.

The entities are the *physical* data items. Multiple copies of the same *logical* data item are considered as different physical data items stored at different sites. The fact that they are copies and must remain identical for reasons of consistency is part of the *integrity constraints* [1], and is not treated separately. We assume that the DD is fixed and given.

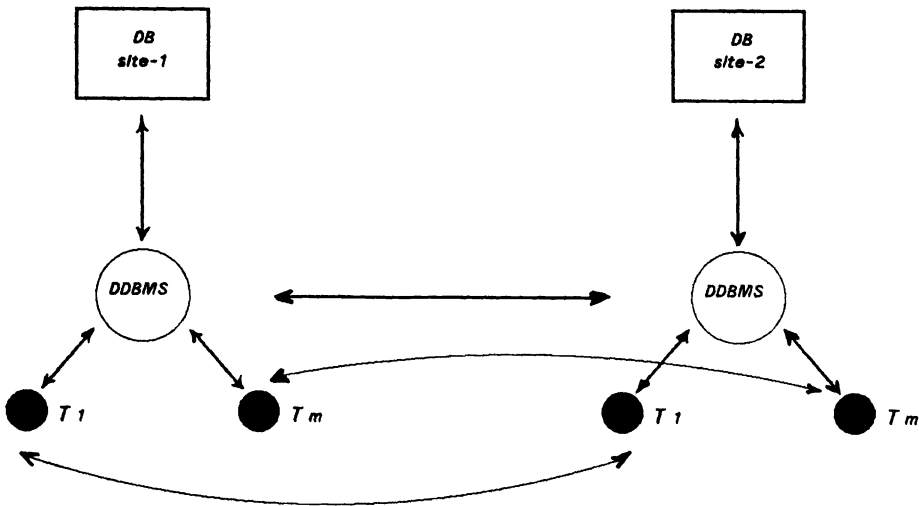


FIG. 1. Architecture of a two-site system.

2.2. Transactions and schedules. The users interact with the database using transactions. In our model a transaction is a distributed program, not identified with a particular site.

DEFINITION 2. A *transaction* T , in a DD, is a directed acyclic graph (dag) $T = (N, A)$ such that:

(a) Every node p is associated with one of the sites of the system, $\text{site}(p) \in U$ and with an entity x_p for which $\text{stored-at}(x_p) = \text{site}(p)$.

(b) Nodes associated with the same site are totally ordered in A , (we denote the partial order imposed by T on its nodes as \cong_T). A *transaction system* \mathbf{T} is a set of transactions $\{T_i, 1 \leq i \leq m\}$.

An example is shown in Fig. 2. Nodes are also called *actions*, since they are intended to represent update actions on the corresponding entity. An action p represents an indivisible read and write operation on x_p [8] (we do not distinguish between read and write operations as in [11]). Action p , as in [8], depends only on actions preceding it in its transaction. Each transaction T represents a distributed program, consisting of communicating sequential processes [6], one per site. Let the t_i 's be variables local to this program, and the f_i 's be uninterpreted function symbols, then the semantics of action p of transaction T is the indivisible execution of the two instructions: $t_p := x_p; x_p := f_p(t_p, t_{p_1}, t_{p_2}, \dots, t_{p_k})$, where p_1, p_2, \dots, p_k are all the actions preceding p in \cong_T .

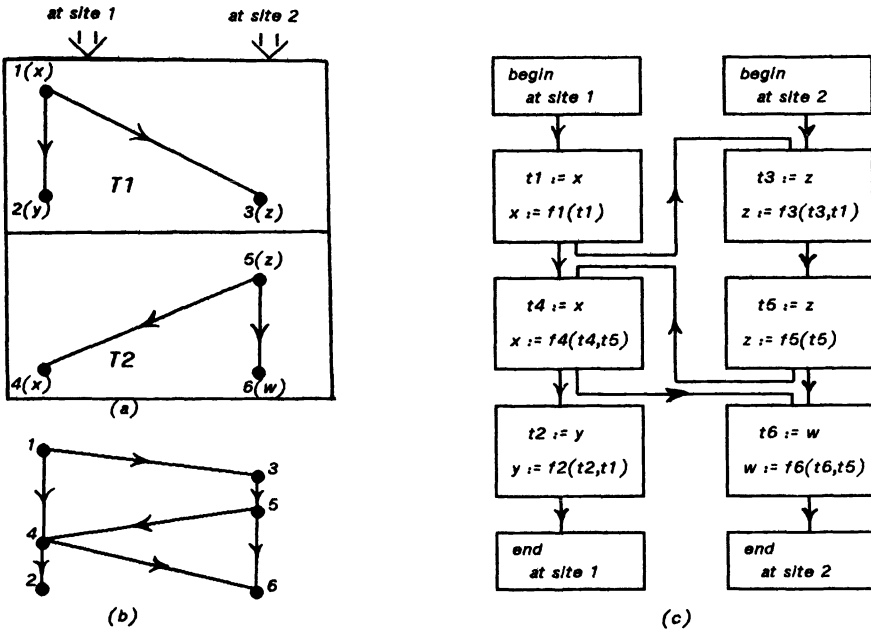


FIG. 2. (a) Transaction system $T = \{T_1, T_2\}$ (e.g., action 1 updates x). (b) Schedule $s = (T, \pi)$. (c) The semantics of the actions in schedule s .

Precedence between actions in a transaction T denotes both temporal precedence and a transfer of information (i.e., in Fig. 2a action 3 needs data from action 1 and is executed after action 1). Arcs in a transaction T between actions at different sites are called *cross-arcs defined in T* . A cross-arc defined in T indicates information transfer between processes of T at different sites.

A schedule is a description of a set of transactions and the process of their execution on the system. In a distributed system it is in general impossible to tell which one of two events occurred first (because communication is not always instantaneous). Because of this uncertainty, we describe the execution order of the actions by a partial order. If two events are incomparable in this partial order, any one could have preceded the other. There are two restrictions on the partial orders. First, what happens at every site is totally ordered; this is consistent with the centralized problem and guarantees that the result of the execution is uniquely determined, as in the case of individual transactions. Second, precedences specified by the transactions are always respected.

Formally:

DEFINITION 3. A *schedule* is a pair $\langle \mathbf{T}, \pi \rangle$, where $\mathbf{T} = \{T_i, 1 \leq i \leq m\}$ is a transaction system and π is a directed acyclic graph (dag) on the nodes of the transactions T_i such that:

- (a) Nodes p with the same $site(p)$ are totally ordered.
- (b) For any transaction T_i and actions $p, q \in T_i$ with $p \geq_{T_i} q$ we have that $p \geq_{\pi} q$ (where \geq_{π} denotes the partial order of π).

A *prefix* of a schedule $s = \langle \mathbf{T}, \pi \rangle$ is a pair $\langle \mathbf{T}, \alpha \rangle$, where α is the subgraph of π induced by a subset of its nodes and such that if action $q \in \alpha$ all $p \geq_{\pi} q$ belong to α .

Let S denote the set of all schedules. Recall that a partial order can be considered as a set of total orders (those compatible with it). Let S^+ denote the set of all schedules $\langle \mathbf{T}, \pi \rangle$, where π is a total order. Therefore a schedule s represents a particular subset of this basic set S^+ . Arcs in a schedule, between actions at different sites are called *cross-arcs*. The schedules with only transaction defined cross-arcs are maximal when considered as sets of total orders. Yet schedules can have other cross-arcs also (e.g., arc (4, 6) in Fig. 2b), whose presence restricts the represented total orders of actions. The goal of *concurrency control* is to recognize *on-line* large sets of correct total orders.

As in the centralized case, synchronization is necessary only between actions of a transaction system, which operate on the same entities (i.e., conflict). These conflicts are represented by the conflict graph $G(\mathbf{T})$. We denote undirected edges by ij and arcs by (ij) .

DEFINITION 4. For the transaction system $\mathbf{T} = \{T_i, 1 \leq i \leq m\}$, the *conflict graph* $G(\mathbf{T})$ is an undirected multigraph (V, E) , with a partial order \geq_i on the edges incident upon each node i , such that:

- (a) $V = \{i | 1 \leq i \leq m\}$, where node i corresponds to transaction T_i .
- (b) E is a multiset of edges.

$$E = \{\text{copies of edge } ij | \text{for every copy of } ij \text{ there is a distinct pair of actions } p, q \text{ with } p \in T_i, q \in T_j, i \neq j \text{ and } x_p = x_q\}.$$

- (c) For two edges incident at node i we have $ij \geq_i ik$ iff the action in T_i corresponding to ij precedes the action in T_i corresponding to ik .

Note that an edge in E denotes a conflict between two transactions. Every edge ij in E corresponds to a pair of actions $\{p, q\}$ which update the same entity. Based on where this entity is stored we can partition E into as many multisets as there are sites: **red** and **green** edges for two sites. An example is presented in Figs. 3a and 3b.

An *ordered mixed multigraph* $G = (V, E, A, \{\geq_i\})$ is a mixed multigraph, with E a multiset of edges, A a multiset of arcs and $\{\geq_i\}$ partial orders at each node i of the edges and arcs incident at the node. An *ordered undirected multigraph* has $A = \emptyset$ (e.g., conflict graphs are such combinatorial objects). An *ordered directed multigraph* has $E = \emptyset$.

Since a conflict (an edge in $G(\mathbf{T})$) corresponds to two actions at the same site and a schedule $s = \langle \mathbf{T}, \pi \rangle$ has a total order of the actions at each site, we say that a schedule *resolves all conflicts*. That is, if edge ij corresponds to the pair of actions $\{p, q\}$, $p \in T_i, q \in T_j, i \neq j$, we direct ij as (ij) iff $p \geq_{\pi} q$. Thus the schedule s determines a unique ordered directed multigraph $G^{\pi}(\mathbf{T})$.

DEFINITION 5. A *prefix* $\langle \mathbf{T}, \alpha \rangle$ assigns a direction (ij) to an edge ij of the conflict graph $G(\mathbf{T})$ iff *all* schedules, which have $\langle \mathbf{T}, \alpha \rangle$ as prefix, assign ij the direction (ij) . Therefore a prefix $\langle \mathbf{T}, \alpha \rangle$ determines an *assignment of directions* to some edges of $G(\mathbf{T})$. Conversely an assignment of directions to edges of the conflict graph is *realizable* by a prefix, if there is a prefix of a schedule assigning these directions and no others.

Thus a prefix $\langle T, \alpha \rangle$ determines a unique ordered mixed multigraph $G^\alpha(T)$, which is $G(T)$ with some of its edges directed. In Fig. 3c we have a nonrealizable assignment of directions. Moreover we have the following complete characterization of realizable assignments of directions, which are, after all, the assignments of interest.

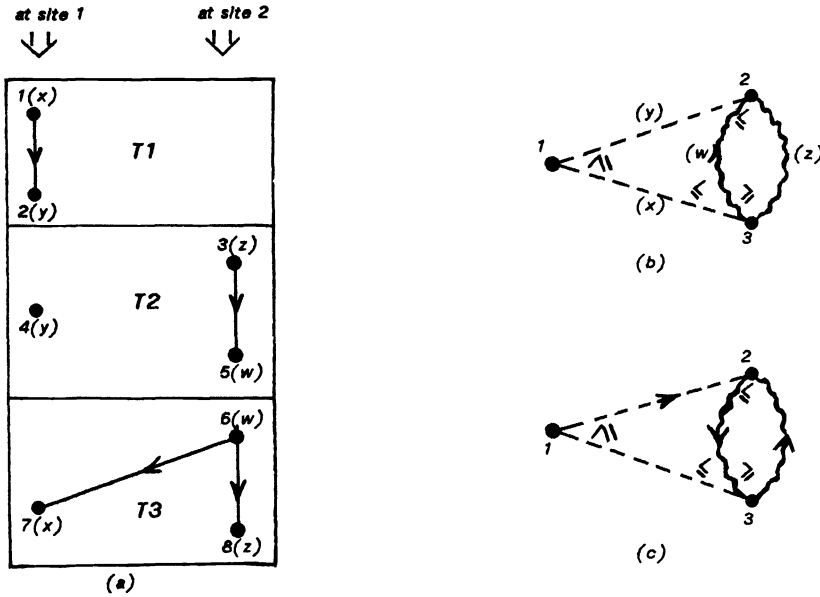


FIG. 3. (a) Transactions. (b) Conflict graph. (c) A nonrealizable assignment; red $\bullet \text{---} \bullet$ = conflicts at site 1, green $\bullet \text{---} \bullet$ = conflicts at site 2.

LEMMA 1. Given a conflict graph $G(T) = (V, E, \emptyset, \{\cong_i\})$, an assignment of directions to a multiset X of its edges, producing the ordered mixed multigraph $(V, E \setminus X, A, \{\cong_i\})$ is realizable iff:

- (a) $ij \in X$, is directed as $(ij) \in A$, and $ik \cong_i ij \Rightarrow ik \in X$.
- (b) A has no directed cycles $(i_1 i_2 i_3 \cdots i_n i_1)$ such that

$$i_1 i_2 \cong_{i_2} i_2 i_3, \quad i_2 i_3 \cong_{i_3} i_3 i_4, \quad \cdots, \quad i_n i_1 \cong_{i_1} i_1 i_2.$$

Proof. "only if". Given a prefix $\langle T, \alpha \rangle$ of a schedule let us first assign the direction (ij) to any edge ij in $G(T)$, corresponding to a pair of conflicting actions $\{p, q\}$, with $p \in T_i, q \in T_j$, under the following conditions:

$$p \in \alpha \quad \text{and} \quad \text{if } q \in \alpha \text{ then } p \cong_\alpha q.$$

It is easily seen that both conditions (a) and (b) hold for the directions constructed above. Obviously all schedules, which have $\langle T, \alpha \rangle$ as prefix, resolve these conflicts in the same way. Moreover if an edge has not been given a direction then both its actions p^*, q^* are not in α . We can complete $\langle T, \alpha \rangle$ using two different schedules, one having p^* before q^* and the other q^* before p^* . One schedule results from completely executing the transaction of p^* first and the other is symmetric. This proves that the directions we have constructed are exactly those assigned by $\langle T, \alpha \rangle$.

Sufficiency. Given an assignment A we construct the following digraph (V^*, A^*) from A and T

$$V^* = \{p \mid \exists (ij) \in A, \text{ where the conflicting action of } ij \text{ in } T_i \text{ is } p \text{ or one of } p\text{'s predecessors in } T_i\},$$

$$A^* = \{(pq) \mid \text{if } (pq) \text{ is part of some } \cong_{T_i} \text{ or if } (pq) \text{ corresponds to an } (ij) \in A\}.$$

Since (b) is true (V^*, A^*) is acyclic, and since (a) is true transaction precedences are respected. Thus (V^*, A^*) has the same nodes as some prefix and respects all its conflict resolving orderings. \square

2.3. Serializability. Only a subset of the possible schedules are considered correct for the operation of the database. The object of concurrency control is to develop algorithms, which monitor the execution of transactions, and disallow incorrect schedules. Actually, our results can be stated in a manner independent of the notion of correctness used in the system. We can show, however, that our negative results hold even when this correctness criterion is a practically important one, that of serializability, which we introduce next.

Serializability can be defined *semantically* [8], [11]. Since we are interested in simplifying our model, in order to bring out the complications inherent to distributed databases, we shall adopt instead a simple *syntactic* definition of serializability. This definition will not require our formally dealing with the semantics of actions and was, interestingly, the first to be proposed [4]. It turns out to be equivalent to the semantic one, if we think of the nodes of the transactions as indivisible read and write operations (see [8]), as opposed to operations that entail either reading or writing an entity [11]. The example of Fig. 2c illustrates the semantics of updates, in terms of program schemes [8], [11]. In fact, the following syntactic definitions suffice for the results presented in this paper.

DEFINITION 6. Two schedules $\langle T, \pi \rangle, \langle T, \rho \rangle$ are *equivalent* if they determine the same ordered directed multigraph, (i.e., $G^\pi(T) = G^\rho(T)$).

DEFINITION 7. A schedule $\langle T, \pi \rangle$ is *serial* iff

(a) The execution of actions at every site introduces a total order of transactions at that site (i.e., there are no $T_i, T_j, i \neq j$ with actions $p, q \in T_i, r \in T_j$ at the same site with $p \cong_\pi r$ and $r \cong_\pi q$).

(b) If T_i precedes T_j at one site it does so at all sites, where both transactions have actions.

A schedule is *serializable* iff it is equivalent to a serial schedule.

We denote the set of serializable schedules by SR ($SR \subseteq S$). What is remarkable, is that deciding whether a schedule is serializable in a centralized or distributed model are practically identical tasks [11]. We state this as follows:

THEOREM 1. A schedule $\langle T, \pi \rangle$ is *serializable* iff it resolves conflicts without creating directed cycles in $G(T)$ (i.e., $G^\pi(T)$ is acyclic). Similarly, a prefix $\langle T, \alpha \rangle$ has a *serializable completion* iff the already resolved conflicts do not create a directed cycle in $G(T)$ (i.e., $G^\alpha(T)$ has no directed cycles).

Proof. Easily follows from the analysis of [11]. \square

2.4. Schedulers. Up until now the distributed problem appears to be a straightforward generalization of the centralized case. What is considerably more complex in the distributed case is the subject of schedulers and their design to meet performance specifications. For an exposition of the relatively simple theory for the centralized case see [11].

Our schedulers will be distributed programs characterized by the parallelism they provide and by their efficiency. We will measure parallelism using the subset C of schedules, which the scheduler allows to be executed as requested. The efficiency of the scheduler will be measured by the worst-case number of steps it executes and the worst-case number of messages it sends. We will be interested in the following kinds of C 's:

DEFINITION 8. Consider a set of schedules $C \subseteq S$, such that for each $s \in C$ the only cross-arcs are defined by the transactions. Such a C we shall call a *concurrency control principle*.

Each schedule s corresponds to a set of *total orders* $\{\sigma \mid \sigma \text{ is a total order compatible with } s\}$. This set is also denoted by s . If C is a set of schedules, we let $C^+ = \bigcup_{s \in C} s$. Recall that S is the set of all schedules and S^+ the set of all total orders. For a particular transaction system T , with n actions, $\sigma \in S^+$ is a string of length n over N , where N is the set of T 's actions. The j th symbol of σ is denoted σ_j .

The cardinality of $C^+ \subseteq S^+$ will be the measure of parallelism. The larger C^+ is, the higher the level of parallelism supported by this concurrency control principle. For example, if SR are the serializable schedules then $SR^+ = \bigcup_{s \in SR} s$. Note that, SR^+ is also the set of total orders of a concurrency control principle, the serializable schedules with only transaction defined cross-arcs; this easily follows from Theorem 1 and Lemma 1. We will hence use the notation SR for this concurrency control principle, without any loss of generality. Similarly serial execution provides another example of a concurrency control principle, which obviously supports less parallelism. Thus concurrency control principles are very natural classes of schedules measuring parallelism, although not all subsets of S can be expressed as such.

A *scheduler* A is a *distributed program*. We do not explicitly specify the model of computation; we use a model equivalent to [6], although we employ a simple concurrent language notation as needed (e.g., a *send-message* instruction). Our distributed programs consist of a set of communicating sequential processes [6], one for each site. Their instructions may denote:

- (a) local computation;
- (b) receiving an execution request for an action q ;
- (c) granting an execution request of an action q ;
- (d) sending a message to another site;
- (e) receiving a message from another site.

We shall now formalize the input-output behavior of the scheduler. Intuitively, a scheduler receives a schedule as its input and outputs another schedule. There is a difficulty though in defining this mapping precisely, because it is essentially a nondeterministic mapping. Although the scheduler has perfectly deterministic algorithms as its processes, the interaction of these algorithms is conducted via *messages*, whose delivery time is unpredictable. M. Fischer uses the term *indeterminism* [20] for this kind of unpredictable behavior (nondeterminism would not be an appropriate term, since we wish to produce correct computations in all cases). To model indeterminism of a scheduler, we must somehow introduce some notion of *time*.

(1) *The input of a scheduler is a string in S^+* . Thus we assume that the arrivals of the requests for executions of the nodes of the schedule-input are totally ordered in time. This is only a simplifying tool (a formalism of the familiar notion of a *timestamp* [10]), and is not used by the scheduler, whose processes still perceive the world in terms of partial orders. We therefore have introduced a *global clock*, whose ticks are the arrivals of the action requests.

(2) What is the output of a scheduler? It is a schedule, of course. However, it

must also add some more information. Namely, it must tell us whether an action was granted before or after the arrival of another request. *The output of the scheduler is an n -tuple of strings $(\tau_1, \tau_2, \dots, \tau_n) \in (N^*)^n$.* Here τ_j denotes the sequence of granted requests between the j th and $(j+1)$ st (after the j th if $j = n$) arrivals of requests. N^* is the set of all strings constructed from the set of actions N and includes the empty string. The concatenation of the n strings, $\text{conc}(\tau_1, \tau_2, \dots, \tau_n)$, should be in S^+ .

(3) We shall now formalize the indeterministic part of the scheduler, namely the communication delays. A *delay vector* \mathbf{d} is a sequence of nonnegative real numbers. Intuitively, the j th component is the delay of the j th message sent by the scheduler. With a given delay vector the operation of a scheduler \mathbf{A} on some input σ is completely specified (exactly as the operation of a nondeterministic algorithm becomes specified if we supply a sequence of choices for the nondeterministic steps). To find the resulting output, we do the following. For each site, we keep a calendar of events (i.e., arrivals of actions or messages, operations of the scheduler), with the precise times at which they occur. An event may trigger a finite sequence of operations of the scheduler, which we execute. If an operation involves sending a message to another site, we add the next component of \mathbf{d} to the time of the present event and we insert the arrival of this message in the calendar of the other site at the time of the sum. We thus assume that, *all local operations of the scheduler take 0 time.* We break ties on the times of events in a systematic fashion (e.g., arrivals of actions first, then messages from site 1, etc). We can now produce the output of the scheduler for this input σ and this delay vector \mathbf{d} in the obvious way from the calendars of events. This output $(\tau_1, \tau_2, \dots, \tau_n)$, we denote by $\mathbf{A}_{\mathbf{d}}(\sigma)$. Not all delay vectors can lead to meaningful executions, however. What can go wrong is that a long delay can postpone the granting of an action p until after the successor q of p in its transaction has been received. Delay vectors for which no such anomaly occurs for an input σ are called *feasible* for σ . The *zero sequence* $\mathbf{d} = \mathbf{0}$ is always feasible.

Therefore the operation of a scheduler is formulated by the function $\mathbf{A}_{\mathbf{d}}: S^+ \rightarrow (N^*)^n$.

Consider a concurrency control principle C . We say that scheduler \mathbf{A} implements C if, intuitively, all outputs of \mathbf{A} are in C and, furthermore, if \mathbf{A} is fed with a schedule in C and all delays are 0, then \mathbf{A} grants all requests immediately upon receipt. It is argued in [11] that these are traits, in the centralized case, of all schedulers that are *on-line* and *optimistic* (i.e., the scheduler does not intervene to unnecessarily delay an action if the input schedule is so far correct). The same arguments are applicable to justify Definition 9.

DEFINITION 9. We say that \mathbf{A} is an *implementation* of concurrency control principle C iff

- (a) $\text{conc}(\mathbf{A}_{\mathbf{d}}(\sigma)) \in C^+$ for all $\sigma \in S^+$ and feasible delay vectors \mathbf{d} , and
- (b) $\mathbf{A}_{\mathbf{0}}(\sigma) = (\sigma_1, \dots, \sigma_n)$ for all $\sigma \in C^+$.

There is a fundamental asymmetry in Definition 9. If the input is in C^+ , then condition (b) is in effect, and the scheduler must leave it intact, unless forced to do otherwise because of the delays. If, however, the input is not in C^+ , then the output can be any schedule in C^+ . In practice, we would expect of a scheduler to change a schedule not in C^+ as little as possible in order to transform it into one in C^+ . Unfortunately, there does not seem to be a clean way to express this mathematically in the distributed or centralized case. We have adopted the above convention in the interest of keeping our model and subsequent proofs as simple as possible.

DEFINITION 10. The *computational complexity* of \mathbf{A} is the sum of the step-counts of all local computations by \mathbf{A} over all processes of \mathbf{A} , maximized over all σ and

feasible \mathbf{d} . The *communication complexity* of \mathbf{A} is the number of all *send-message* instructions executed by all processes of \mathbf{A} , maximized over all σ and feasible \mathbf{d} .

Note that apart from the messages generated by the scheduler processes of the system, there is also user defined communication, implied by transaction defined cross-arcs (e.g., some action at site 2 needs data from site 1). *This communication is assumed free, since it is unavoidable.* Such messages, between the processes of a transaction, can be used to pass information between scheduler processes at no cost.

A scheduler \mathbf{A} is polynomial-time bounded (or *computationally efficient*) if its computational complexity is bounded by a polynomial in n (where $n = |N|$ and N is the set of actions of \mathbf{T}). Similarly, with [11] we can prove:

THEOREM 2. *C has a computationally efficient implementation iff the set of prefixes of C is in P .*

Proof. By broadcasting each arrival of a request we can reduce the distributed to the centralized problem, and use [11, Thm. 10]. Note that this solution is wasteful in terms of messages. \square

Finally in order to characterize communication complexity we define the following classes of prefixes $M_c(b)$:

DEFINITION 11. For concurrency control principle C , its set of prefixes $PR(C)$ and integer $b \geq 0$,

$$M_c(b) = \{\text{prefixes not in } PR(C)\} \\ \cup \{\langle \mathbf{T}, \alpha \rangle \mid \langle \mathbf{T}, \alpha \rangle \in PR(C) \text{ and there is an implementation } \mathbf{A} \text{ of } C, \\ \text{which, given that } \langle \mathbf{T}, \alpha \rangle \text{ has been granted,} \\ \text{proceeds using at most } b \text{ send-message's}\}.$$

Let $b^*(\mathbf{T})$ be the least b for which $\langle \mathbf{T}, \emptyset \rangle \in M_c(b)$. A scheduler which achieves $b^*(\mathbf{T})$, for every \mathbf{T} , is called *communication-optimal* with respect to C .

Note that for $b < 0$ we can define $M_c(b) = \emptyset$ and then for all b we have $M_c(b) \subseteq M_c(b+1)$. If $\langle \mathbf{T}, \alpha \rangle$ is a prefix of $\langle \mathbf{T}, \beta \rangle$ and $\langle \mathbf{T}, \alpha \rangle \in M_c(b)$, then also $\langle \mathbf{T}, \beta \rangle \in M_c(b)$. By our convention if $\langle \mathbf{T}, \alpha \rangle$ is not a prefix of a schedule in C then $\langle \mathbf{T}, \alpha \rangle \in M_c(0)$. Intuitively, if all sites know of an incorrect input they can output a predetermined correct completion without communication.

In essence, what Definition 11 says is that: the scheduler might use a priori information, available to all scheduler processes, in order to enhance the communication performance (worst-case number of messages used at run time) of the concurrency control mechanism. For example, a scheduler that implements serializability (for all transaction systems \mathbf{T}), might also examine the available syntax of transaction system \mathbf{T} , in order to develop a more economical communication strategy between its processes. This is analogous to the conflict graph analysis used to improve parallelism in SDD-1 [1], [2]. A communication-optimal scheduler is the limit in message performance attainable, subject to a parallelism requirement C . In the following section we will show, in a constructive fashion, that such schedulers exist for concurrency control principles.

3. Communication-optimal schedulers and games. The performance measure of a concurrency control algorithm is a set of schedules C . We require C to be a concurrency control principle (see Definition 8). Let $PR(C)$ be the set of prefixes of schedules in C . We assume that we have an efficient (polynomial time in n) test of membership of a prefix in $PR(C)$. For example, if $C = SR$ Theorem 1 provides us with such a test. If no such test is possible, concurrency control is quite hopeless, even

in the centralized case [11]. We also assume that we have a two-site system. This is no loss for the negative results of the next section. As for the positive results, they can be restated without much difficulty, although less succinctly, for the general case.

Let us briefly review the notation used. A prefix is denoted as a pair $\langle T, \alpha \rangle$, or simply α when there is no ambiguity. In order to make our notation simple we will omit T , the obvious transaction system, whenever possible. We use $M_c(b)$, for the set of all prefixes $\langle T, \alpha \rangle$ of C such that there is an implementation of C , which, when started with $\langle T, \alpha \rangle$, sends b or fewer messages. Now let α be a prefix of β , then $(\beta/\alpha)_i$ denotes the prefix of β , that contains α and all actions of β at site i . We call this the *projection* of β at site i given α (see Fig. 4 for an example of this important notion).

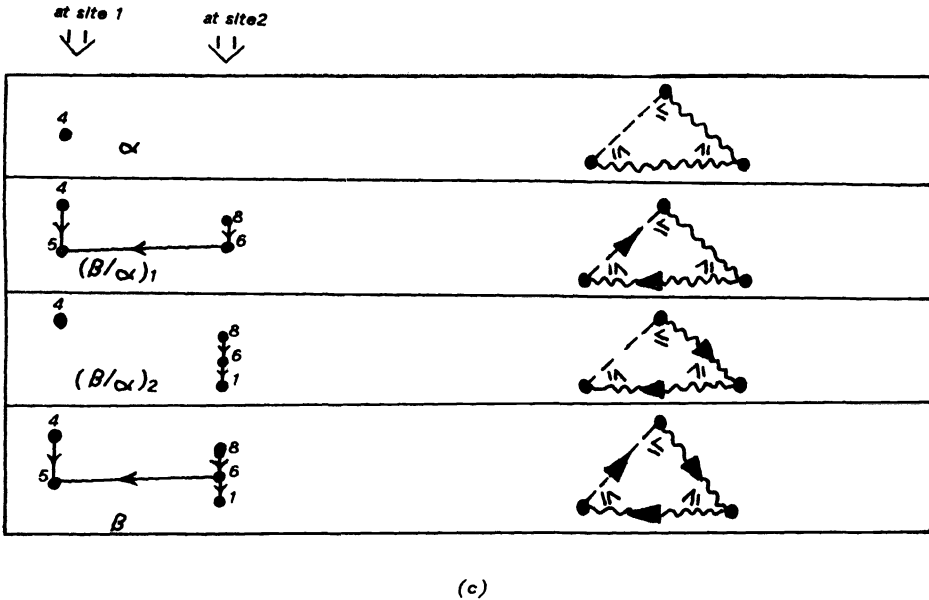
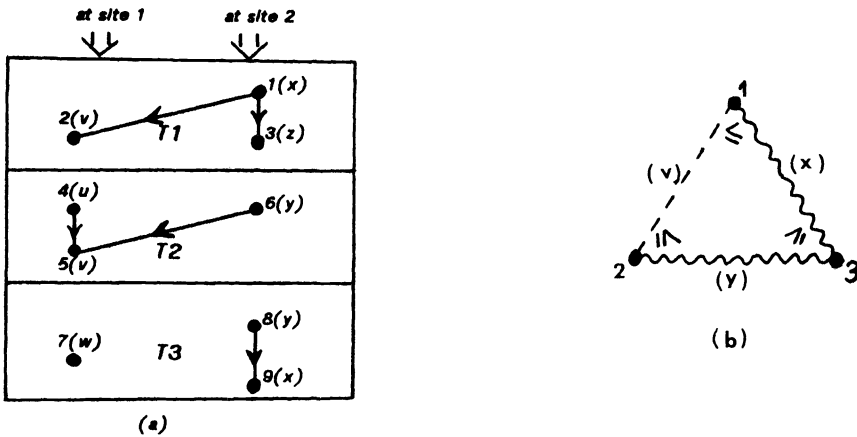


FIG. 4. (a) Transactions (u, v, w at site 1, x, y, z at site 2). (b) Conflict graph (red $\bullet \text{---} \bullet$ = conflicts at site 1, green $\bullet \text{~~~~} \bullet$ = conflicts at site 2). (c) Illustrating a bad β . Left: prefixes. Right: assignments of directions.

DEFINITION 12. Let $\langle T, \alpha \rangle \in PR(C)$, and let α be a prefix of β . We call β *bad*, with respect to α , if

- (a) $(\beta/\alpha)_1, (\beta/\alpha)_2 \in PR(C)$; and
- (b) $\beta \notin PR(C)$.

It is only bad prefixes that force the scheduler to communicate, in rounds of two messages. This, as well as a description of the possible strategies for guarding against bad prefixes, is captured by the following theorem.

THEOREM 3. Let C be a concurrency control principle, $\langle T, \alpha \rangle \in PR(C)$ and $b \geq 0$. Let i denote the site number, $i \in \{1, 2\}$. Then statements (I) and (II) are equivalent:

- (I) $\alpha \in M_c(b)$.
- (II) For all bad β , with respect to, α : (1) $(\beta/\alpha)_i \in M_c(b)$ for $i = 1, 2$ and (2) at least one of the $(\beta/\alpha)_i \in M_c(b-2)$.

The intuitive interpretation of the theorem is the following: Suppose that there is a possible scenario (see Fig. 4 for an example) in which both sites see projections $(\beta/\alpha)_i$, that are perfectly legal locally (i.e., both are in $PR(C)$) and still, they are not legal when put together (i.e., β is not in $PR(C)$). This clearly calls for communication. The theorem says that, in the worst case, two messages are both necessary and sufficient to overcome this problem.

Proof. “only if”. To show that (I) implies (II), suppose that a scheduler \mathbf{A} can start from α and implement C with only b messages. Let β be bad with respect to α . It is easy to see that (II.1) is satisfied. We must now show that (II.2) is also true.

What should site i do if it is presented with requests for the actions in $(\beta/\alpha)_i$? Clearly, it should have a way of granting them, perhaps after certain communication, since \mathbf{A} is supposed to implement C (i.e., see Definition 9 for the on-line property). If site i grants the requests without waiting for any messages, then site $j = 3 - i$ must guard against this eventuality, when presented with $(\beta/\alpha)_j$, by asking site i 's state. This takes two messages, which synchronize the two processes, and thus \mathbf{A} must implement C starting from $(\beta/\alpha)_j$ within $b - 2$ messages; thus property (II.2) holds. This leaves us with the case in which site i waits for a message before granting $(\beta/\alpha)_i$. It cannot wait for a message triggered by any event at site j other than an arrival of a message from i ; this follows from the fact that \mathbf{A} must implement C . We are therefore reduced to the previously examined case.

“if”. To show that condition (II) is sufficient, we shall construct an explicit algorithm that implements C , starting from α and using no more than b messages, assuming (II) holds. The algorithm is recursive, and is shown in Fig. 5.

The algorithm, **localscheduler**, is the process run by each site. Its arguments are the prefix $\langle T, \alpha \rangle$ of granted actions at the instant it takes over and the number b of messages that it can use. For example if no actions have been granted, both sites start by running **localscheduler**($\langle T, \emptyset \rangle, b$).

The variable *localstate* represents the actions that the site knows are granted (through its own granting actions and other messages), whereas *commonstate* is the information this site knows the other site already has. The values of these variables are prefixes in $PR(C)$. They are both initialized to $\langle T, \alpha \rangle$ and updated appropriately whenever:

- (a) An action is granted at this site, through the function **grant**(p).
- (b) A message is exchanged by scheduler processes, through the functions **askstate** and **reportstate**.
- (c) A message is exchanged by transaction processes, because of a transaction defined cross-arc.

In the last case the *localstate* at one site, may be passed to the other at no communication

cost. The detailed code for performing these updates or the functions **grant**, **askstate** and **reportstate** is straightforward and is not shown in Fig. 5. The low level details of all these functions can be found in [7].

When a request p arrives, the scheduler first decides whether it is necessary to communicate. This is the first test in Fig. 5. Communication is forced just in the case that a prefix β exists, such that:

- (a) it violates the concurrency control principle C (i.e., $\beta \notin PR(C)$);
- (b) its projection at the other site given *commonstate* is in $PR(C)$;
- (c) its local projection is *localstate*// p (where // denotes concatenation), and moreover it is amenable to scheduling with $b-2$ messages. In other words, condition (II.2) of the theorem is satisfied with i equal to the present site.

```

procedure localscheduler( $\langle T, \alpha \rangle, b$ )
  localstate := commonstate :=  $\langle T, \alpha \rangle$ ;
  on request-arrival  $p$  do
    if there is a prefix  $\beta \notin PR(C)$ , whose projection at the other site
      given commonstate is in  $PR(C)$ , and whose local projection is
      localstate// $p \in M_c(b-2)$ 
      then
        begin
          localstate := askstate( );
          if localstate// $p \in PR(C)$ 
            then grant( $p$ ); localscheduler(localstate,  $b-2$ )
            else tablelookup( )
          end
        else if localstate// $p \in PR(C)$ 
          then grant( $p$ )
          else tablelookup( )
        end localscheduler

```

FIG. 5. The process **localscheduler**.

By convention, any prefix not in $PR(C)$ needs 0 messages and therefore the prefix *localstate*// $p \notin PR(C)$ would pass the test only if $b > 0$. Except for this case a β , such that the above conditions are true, is one satisfying (II) of the theorem.

If the above conditions are met, the scheduler decides to communicate. The function **askstate** learns the state of the other site at the cost of two messages. Presumably the return message is sent by a function **reportstate** at the other site, which also does the appropriate updating. If p is found to be safe, it is granted, and **localscheduler** is called recursively with the new arguments (note that *localstate* would be appropriately updated by **grant**). Since the test succeeded, we know that it can carry out its task within $b-2$ messages. If now *localstate*// $p \notin PR(C)$, then the arriving stream of requests is not in C , and therefore we have no contract to fill (recall the paragraph right after Definition 9); both sites continue scheduling by some **tablelookup**, agreed upon in advance between the **sites**.

If the first test fails, then we must proceed with locally available information. If p looks safe, we grant it. We know we are not risking anything since, by (II), the other site will pass the test, and will communicate before it grants its part of any bad β . If p is not safe, we again resort to **tablelookup**, but now since $b = 0$ and (II) is true both sites can proceed independently with no risk.

The formal proof that the algorithm, as specified above, correctly schedules a given prefix within the given number of messages is now straightforward, by induction on the number of actions in any suffix of $\langle \mathbf{T}, \alpha \rangle$. It should also be noted that we use the fact that C is a concurrency control principle when we test if a $localstate//p$ is in $PR(C)$. Since schedules in C have only transaction defined cross-arcs this test can be done locally. \square

COROLLARY 3.1. *If C , a concurrency control principle, has a computationally efficient implementation, then it has a communication-optimal implementation, which uses space polynomial in n ($n = \text{number of actions of } \mathbf{T}$).*

Proof. The hardest computation performed by **localscheduler** in the proof of Theorem 3 is testing whether a prefix is in $M_c(b)$. This, however, can be expressed as a predicate with polynomial matrix and b alternations of quantifiers. It is therefore in PSPACE [3]. \square

Distributed scheduling is related to a game on prefixes called PREFIX. The rules of this game are displayed in Fig. 6. In this game Player I corresponds to a malicious adversary who wishes to force communication. His move is a continuation β of the current position α , which satisfies the conditions of Theorem 3. Player II corresponds to the two cooperating scheduler processes. Each one of his choices i^* indicates, which of the two processes has the responsibility of guarding against the continuation β (by questioning the other process before proceeding). Player I wants to prolong the game as much as possible, whereas Player II tries to bring it to an end as soon as possible (other than that there is no winner or loser). Players I and II take turns moving.

COROLLARY 3.2. *The minimum number of messages used by a communication-optimal implementation of C equals the length of PREFIX($\langle \mathbf{T}, \emptyset \rangle$) if both players play optimally, (we call this the minimax length).*

Proof. It follows from Theorem 3 and the theory of alternation [3]. Note that although in general we define PREFIX from an arbitrary initial position $\langle \mathbf{T}, \alpha \rangle$, we are in fact interested in $\alpha = \emptyset$, (\mathbf{T} represents the static a priori information on transactions, that is used to optimize communication). As a result the question: “ $\langle \mathbf{T}, \alpha \rangle \notin M_c(b)$?” is equivalent to “can Player I make PREFIX($\langle \mathbf{T}, \alpha \rangle$) last more than b moves?” \square

PREFIX($\langle \mathbf{T}, \alpha \rangle$)

Position before player I's move: A prefix $\langle \mathbf{T}, \alpha \rangle$

Player I: Select a prefix β , which has α as a prefix **such that:**

(1) $(\beta/\alpha)_1, (\beta/\alpha)_2 \in PR(C)$

(2) $\beta \notin PR(C)$

Player II: Select $i^* \in \{1, 2\}$ and set $\alpha := (\beta/\alpha)_{i^*}$

FIG. 6. The game PREFIX.

4. The complexity of PREFIX. In this section we prove the following theorem:

THEOREM 4. *Let $C = SR$. Given \mathbf{T} and $b \geq 0$, determining whether the minimax length of the game PREFIX($\langle \mathbf{T}, \emptyset \rangle$) equals b is PSPACE-complete.*

This theorem, as is pointed out explicitly in a series of corollaries, is a fundamental negative complexity result for distributed concurrency control.

It turns out that PREFIX, with $C = SR$, is closely related to a game played on the conflict graph of \mathbf{T} . Recall that the conflict graph is an ordered undirected multigraph with edges colored red or green. The game, called CONFLICT, is displayed in Fig. 7.

CONFLICT(G)

Position before player I's move: An ordered mixed multigraph $G = (V, E, A, \{\cong_i\})$, with E partitioned into red and green, and A closed.

Player I: Select a set X of edges and assign directions to them. Let $X_r(X_g)$ be a subset of X containing all its red (green) edges and let $A_r(A_g)$ be the corresponding arcs. The sets A_r, A_g must be **such that:**

- (1) $A \cup A_r, A \cup A_g$ are acyclic and closed;
- (2) $A \cup A_r \cup A_g$ has a cycle and is closed;

Player II: Select $c \in \{r, g\}$ and set $E := E \setminus X_c; A := A \cup A_c$.

FIG. 7. The game CONFLICT.

A round (i.e., of moves by the two players) starts with a position, which is an ordered mixed multigraph $G = (V, E, A, \{\cong_i\})$. Player I gives directions to certain undirected edges X , with subsets X_r, X_g , such that already existing arcs (i.e., A) and each new directed subset A_r or A_g do not create a cycle, whereas all arcs together do. Player II picks a color (i.e., red or green) and fixes the directions proposed by I (i.e., creates a new A). In the absence of the partial orders \cong_i , the moves of Player I are very simple: He picks a *two-color cycle* that contains some red edges (X_r), some green edges (X_g) and possibly some arcs, and the arcs are all directed with the same sense around the cycle. These rules are complicated a little by the existence of the partial orders on edges and arcs. Again Player I chooses a set of undirected edges X and assigns directions to them, but now the sets of arcs $A \cup A_r, A \cup A_g$ and $A \cup A_r \cup A_g$ must be *closed* (e.g., each one of X_r, X_g contains all edges of one color in X and might contain some edges of the other color) where formally:

“arc (ij) is in a *closed* set of arcs and $ik \cong_i (ij) \Rightarrow ik$ is in this set as arc (ik) or (ki) ”

Again, as in PREFIX, Player I's goal is to prolong and Player II's is to shorten the game. The intuition behind CONFLICT and its relation to concurrency control is the following:

Concurrency control means to direct somehow all edges of the conflict graph, without forming directed cycles. (The color, red or green, of an edge is the site that is responsible for directing it.) To carry out this task in a distributed fashion, we may have to communicate, in order to prevent two-color cycles. Single-color cycles are benign, since they can be detected locally and prevented without communication. Player I's move is an orchestrated stream of requests for conflict resolutions, that forces such a communication. Player II, the distributed scheduler, chooses the site (color) that will send a message, trying to block long sequences of legal moves for I (i.e., trying to save messages). The connection between the concurrency control problem and PREFIX was established in Corollary 3.2. The connection between PREFIX and CONFLICT discussed above, can be formalized in the following, straightforward lemma:

LEMMA 2. *The minimax length of the game PREFIX($\langle T, \emptyset \rangle$), with $C = SR$, equals the minimax length of the game CONFLICT($G(T)$), (i.e., $G(T)$ is the conflict graph of T).*

Proof. The correspondence between PREFIX($\langle T, \alpha \rangle$), and CONFLICT($G^\alpha(T)$) is easily seen to be as follows:

- α corresponds to A (i.e., the conflicts of $G(T)$ resolved by α);
- β corresponds to $A \cup A_r \cup A_g$ (i.e., a nonserializable input);
- $(\beta/\alpha)_1$ corresponds to $A \cup A_r$ (i.e., a serializable projection at site 1 given α);
- $(\beta/\alpha)_2$ corresponds to $A \cup A_g$ (i.e., a serializable projection at site 2 given α).

$A, A \cup A_r \cup A_g, A \cup A_r, A \cup A_g$ have to be closed, because the moves in CONFLICT must be realizable by prefixes (see Lemma 1, § 2.2). \square

It is easy to see that CONFLICT is in PSPACE (that is, computing the minimax length is in PSPACE). To show Theorem 4, we shall first prove that CONFLICT is PSPACE-complete. We start by proving a weaker result, whose proof is indicative of the method used [3], [5], [14].

LEMMA 3. *Computing the minimax length of CONFLICT is Π_2^P -hard, (even when the initial mixed graph has no orders on the edges).*

Proof. Let F be an AE-quantified Boolean formula

$$F = \forall x_2 \forall x_4 \cdots \forall x_n \exists x_1 \exists x_3 \cdots \exists x_{n-1} F^*(x_1, \dots, x_n),$$

where F^* is a 3CNF formula with n variables (n is even) and m clauses. We shall construct a mixed graph G such that the minimax number of rounds of CONFLICT (rounds of moves by the two players), started on G , is equal to $(n/2) + 1$ iff F is true. G is constructed as follows:

For each existentially quantified variable $x_i, i = 1, 3, \dots, n-1$, we add to G a copy of the \exists -graph shown in Fig. 8c. For each universally quantified variable $x_i, i = 2, 4, \dots, n$, we add to G a copy of the \forall -graph in Fig. 8a. Finally, for each clause C_k , we add to G the C -graph in Fig. 9. All these subgraphs are connected as indicated from vertex names (i.e., "in tandem"), with \exists -graphs alternating with \forall -graphs, followed by the C -graphs (that is, $S_{n+1} = C_1$). The "cycle" is closed by a green edge $S_1 C_{m+1}$ (see Fig. 10 for an example).

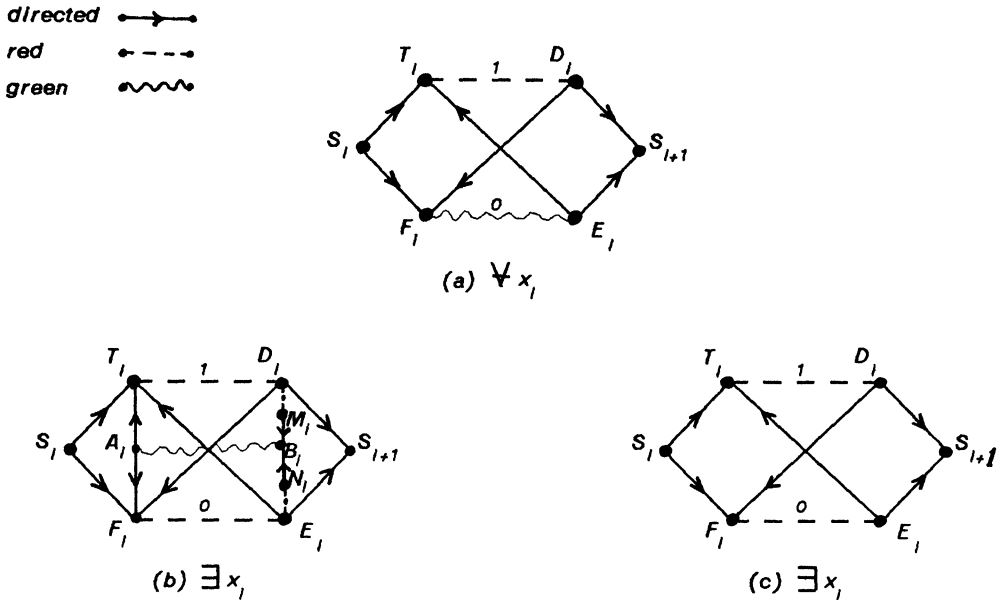


FIG. 8

So far we have only taken into account the numbers n and m . To encode the structure of F^* into G , we must look at the C -graphs of Fig. 9 in some detail. The C -graph consists of 7 paths, numbered from 001 to 111. These are the 7 truth assignments to the literals u, v, w of the clause, that satisfy the clause. Thus each of the 21 red edges of a C -graph, say e , is associated with a literal $l(e)$ and a truth value

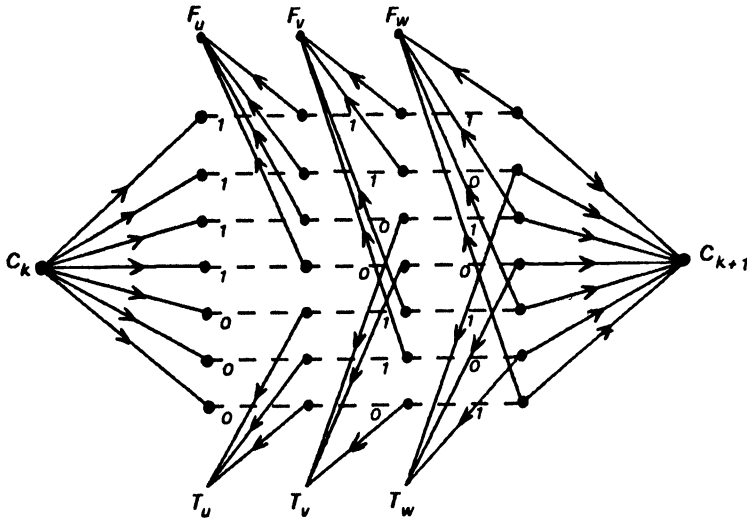


FIG. 9

$t(e)$. We now connect e 's right endpoint with appropriate \exists - and \forall -graphs. We draw an arc from the right endpoint of the edge e

- to F_j if $(l(e) = x_j \text{ and } t(e) = 1)$, or $(l(e) = \neg x_j \text{ and } t(e) = 0)$; and
- to T_j if $(l(e) = x_j \text{ and } t(e) = 0)$, or $(l(e) = \neg x_j \text{ and } t(e) = 1)$.

These arcs are called *backarcs*.

This completes the construction of G (i.e., all orders \cong_i are empty). In Fig. 10 we have an example of the construction if we ignore the nodes A_i, B_i, M_i, N_i $i = 1, 3$ and A_{13}, A_{12}, A_{34} .

We now claim that, from the mixed graph G the minimax number of rounds (rounds of two moves each) is $(n/2) + 1$ iff F is true. Clearly, since there are $(n/2) + 1$ green edges, this number is at most $(n/2) + 1$. We shall show that Player I can force $(n/2) + 1$ rounds iff F is true.

Since the orders are empty, Player I's moves consist of choosing two-color directed cycles. These contain just one green edge (if I is to play $n/2 + 1$ times), and, if we disregard this green edge, there is no directed cycle in the graph with the proposed directions of red edges. It is easy to see that each green edge can be used only in one move, even if Player II does not explicitly direct it after this move (i.e., if his choice is red, in the new A , he has created a directed path between the endpoints of the green edge, and thus implicitly fixed its direction). Without loss of generality, the first $n/2$ moves will involve the green edges $F_i E_i$ of the \forall -graphs. The two-color cycle $(F_i E_i T_i D_i F_i)$ is such a possibility. The choices of Player II can be thought of as fixing the direction of $F_i E_i$ to: $(F_i E_i) - (x_i = 0)$ or $(E_i F_i) - (x_i = 1)$.

The claim is that Player I has an $[n/2 + 1]$ st move, no matter what Player II plays, iff F is true. Player I has a $[n/2 + 1]$ st move iff at the end there is a two-color cycle, which contains the only green edge left, $(C_{m+1} S_1)$, some red edges, some directed edges and no directed cycle without the green edge. Picking red edges is no problem—one has to do this to "pass through" the C -graphs and the \exists -graphs. In the \forall -graphs, the path must follow either $(S_i T_i D_i S_{i+1})$ or $(S_i F_i E_i S_{i+1})$. It follows the latter iff $(F_i E_i)$ was picked by Player II in the corresponding move—otherwise a cycle $(F_i E_i T_i D_i F_i)$ would be created. In the \exists -graphs, this choice can be thought of as an assignment of the truth value to x_i by Player I (i.e., 1 if $(S_i T_i D_i S_{i+1})$ was picked, 0 if $(S_i F_i E_i S_{i+1})$ was

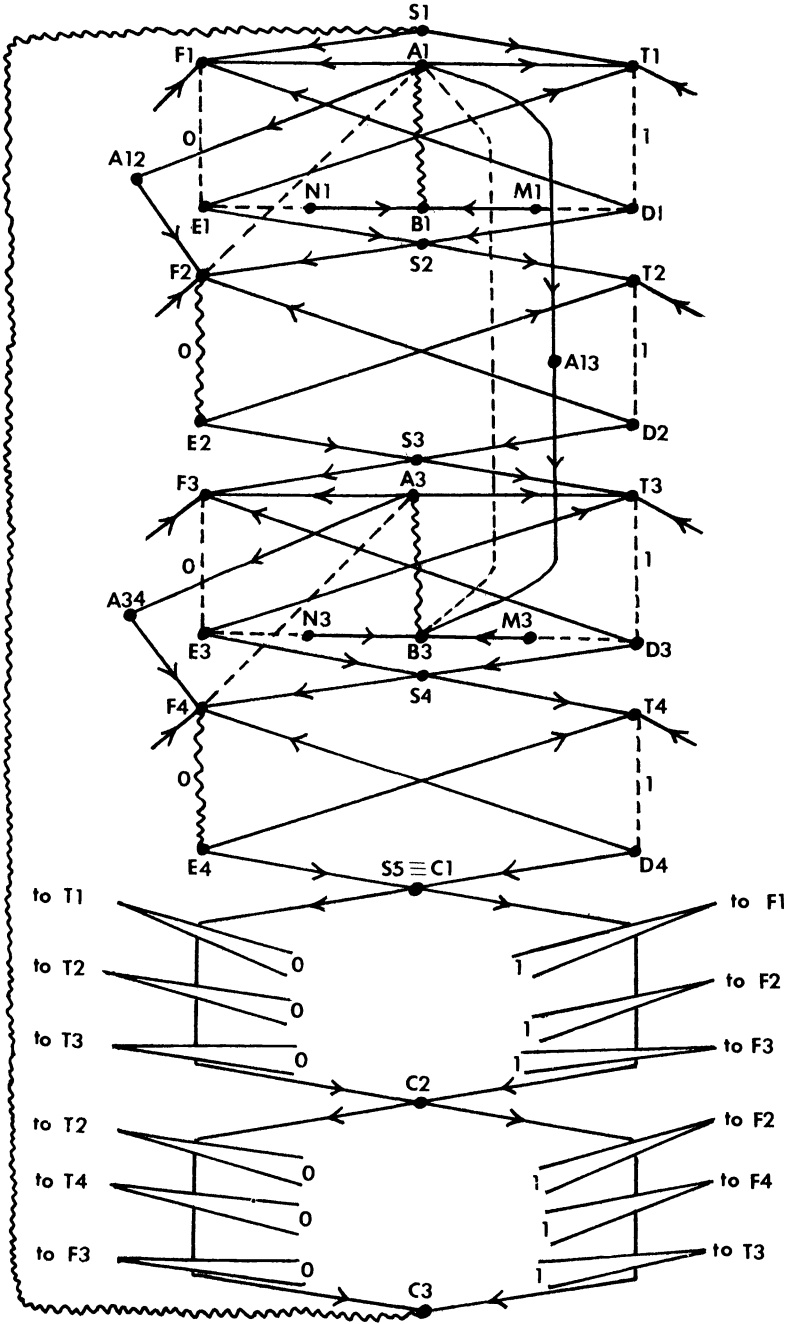


FIG. 10. $\exists x_1 \forall x_2 \exists x_3 \forall x_4 (x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee x_4 \vee \bar{x}_3)$.

picked). Finally, in each of the *C*-graphs, Player I must pick one of the 7 paths, which would not create cycles because of the backarcs. Therefore this path corresponds to a truth assignment, which agrees with the one chosen at the \forall - and \exists -graphs. It follows that such a path (indeed, such an $\lceil n/2 + 1 \rceil$ st move by Player I) exists iff F^* is satisfiable no matter what the values of x_2, x_4, \dots, x_n are, or, equivalently iff F is true. \square

LEMMA 4. *Computing the minimax length of CONFLICT is PSPACE-complete, (even when the initial ordered graph is undirected).*

Proof. There are two directions in which we must extend the previous proof. First, we must encode in G the n alternations of quantifiers. We do this by designing a more elaborate \exists -graph (containing a green edge too) and using the partial orders $\{\cong_i\}$. Second, we must get rid of the directed arcs of G . We do this last, by replacing each directed arc by a triangle, and using the partial orders.

Starting from the *QBF* instance $F = \exists x_1 \forall x_2 \exists x_3 \cdots \exists x_{n-1} \forall x_n F^*(x_1, \dots, x_n)$ we construct an ordered mixed graph G by putting together the \exists -graphs of Fig. 8b (not 8c), the \forall -graphs of Fig. 8a and the C -graphs of Fig. 9, as in Lemma 3. We also have the following edges connecting neighboring \forall - and \exists -graphs:

$$\begin{array}{ll} \text{arcs} & (A_i A_{i+2}), (A_{i+2} B_{i+2}), \quad i = 1, 3, \dots, n-3, \\ & (A_i A_{i+1}), (A_{i+1} F_{i+1}), \quad i = 1, 3, \dots, n-1, \\ \text{red edges} & A_i B_{i+2}, \quad i = 1, 3, \dots, n-3, \\ & A_i F_{i+1}, \quad i = 1, 3, \dots, n-1. \end{array}$$

(These connections will guarantee that the order of moves by Player I will respect the order of quantification.) A full example is shown in Fig. 10.

Notice that, so far, we have not specified the orders $\{\cong_i\}$. The orders for the arcs can be empty and for the undirected edges arbitrary total orders exist at all nodes except for the A_j, B_j, F_j nodes. There they are designed in such a way that Player I must play the green edges in their quantificational order (if the closure properties are to hold):

$$\begin{array}{ll} \text{at } A_i, & A_i B_i \cong A_i F_{i+1} \cong A_i B_{i+2}, \quad i = 1, 3, \dots, n-1 \text{ (the last for } i \neq n-1), \\ \text{at } F_{i+1}, & F_{i+1} A_i \cong F_{i+1} E_{i+1}, \quad i = 1, 3, \dots, n-1, \\ \text{at } B_{i+2}, & B_{i+2} A_i \cong B_{i+2} A_{i+2}, \quad i = 1, 3, \dots, n-3. \end{array}$$

We can indicate these total orders by assigning the integers 1, 2, 3 to the undirected edges at each node and using the ordering of these numbers (see Fig. 11a).

We claim that the minimax number of rounds equals $n+1$ (again, the number of green edges in G) iff F is true. This would prove the lemma, modulo the presence of directed edges. The proof parallels that of Lemma 3, but is slightly harder.

It is easy to see that if Player I wishes to play $n+1$ rounds each one of his moves has to contain exactly one green edge, whose direction has not been fixed by previous moves. Therefore, as in Lemma 3, a game in which Player I can force $n+1$ rounds is essentially a permutation of the $n+1$ green edges. We will thus name his moves after their green edge. We will demonstrate that $A_i B_i$ -moves $i = 1, 3, \dots, n-1$ will correspond to Player I assigning values for the \exists -variables of F and $F_i E_i$ -moves $i = 2, 4, \dots, n$ to Player II assigning values to the \forall -variables of F . Moreover in a game where both players play optimally these choices alternate. The matter will consequently be reduced to the existence of an $[n+1]$ st round, which will be equivalent to the validity of F .

Necessity. Assume the *QBF* instance F is false. We will describe a strategy for Player II, that will make the $[n+1]$ st round impossible.

If Player I wishes to play $n+1$ rounds his game will be constrained in a variety of ways:

(a) Every $A_{i-2} B_{i-2}$ -move must precede the $A_i B_i$ - and $F_{i-1} E_{i-1}$ -moves $i = 3, 5, \dots, n+1$. Since the arcs of G have to be respected, we can only have $(B_i A_i) \in A_g$ and $(F_{i-1} E_{i-1}) \in A_g$ for legal assignments in these moves. This is because $A_g \cup A_r \cup A$

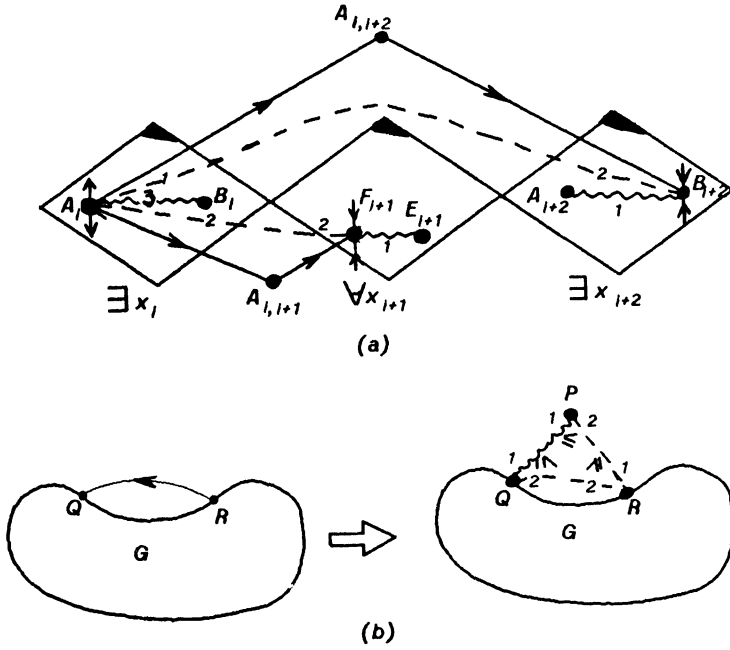


FIG. 11. (a) Forcing alternation. (b) Forcing directions: red $\bullet \dashrightarrow \bullet$; green $\bullet \rightsquigarrow \bullet$; directed $\bullet \rightarrow \bullet$.

must contain a cycle whose orientation is determined by the existing arcs. Now we can justify the construction in Fig. 11a. If $(B_i A_i) \in A_g$ from the \cong_{B_i} order and the closure property of moves we have that $(A_{i-2} B_i) \in A_g \cup A$ (e.g., the direction of $A_{i-2} B_i$ is fixed because of the existing directed path $(A_{i-2} A_{i-2,i} B_i)$ in G). From the $\cong_{A_{i-2}}$ order we have now that $A_{i-2} B_{i-2}$ must already be assigned a direction. Thus the $A_{i-2} B_{i-2}$ -move must have already taken place. A similar argument holds for $F_{i-1} E_{i-1}$. It is easy to see also that the $C_{m+1} S_1$ -move has to follow the $F_n E_n$ -move.

(b) The $F_i E_i$ -move corresponds to Player II assigning a value to x_i , $i = 2, 4, \dots, n$; as in Lemma 3.

(c) The $A_i B_i$ -move corresponds to Player I assigning a value to x_i , $i = 1, 3, \dots, n-1$. The only possible choices of cycles are $(B_i A_i T_i D_i M_i B_i)$ corresponding to $x_i = 1$ and $(B_i A_i F_i E_i N_i B_i)$ corresponding to $x_i = 0$. For $x_i = 1$ ($x_i = 0$ is symmetric) the choice is forced by the existing arcs and because:

- $(B_i A_i B_{i+2} A_{i+2} \dots)$ would use up $B_{i+2} A_{i+2}$;
- $(B_i A_i T_i D_i F_i E_i \dots)$ would introduce a cycle in $A_r \cup A$;
- $(B_i A_i T_i D_i S_{i+1} \dots)$ would fix the direction of $F_{i+1} E_{i+1}$.

The strategy of Player II in response to these moves will be always to play red, fixing the directions of $T_i D_i$ and $F_i E_i$ and making vertex A_i inaccessible from S_i .

Obviously the best Player I can do is assign a value to x_1 (by the $A_1 B_1$ -move), force Player II to show his hand by assigning a value to x_2 (by the $F_2 E_2$ -move), assign a value to x_3 etc. As a result the choices for the $C_{m+1} S_1$ -move are constrained as in Lemma 3. Consequently the existence of a legal $[n+1]$ st round depends on whether the assignment of values to the x_i 's has made $F^*(x_1, \dots, x_n)$ true. Since the QBF instance F is not valid Player II can always pick values for x_i , $i = 2, 4, \dots, n$ that make $F^*(x_1, \dots, x_n)$ false and the $[n+1]$ st round impossible.

Sufficiency. Assume F is true. Player I's game follows the same structure as above. Only now, because of the validity of F , he can choose an assignment for $x_i, i = 1, 3, \dots, n-1$ which will make $F^*(x_1, \dots, x_n)$ true and the $[n+1]$ st round possible.

Finally, we must eliminate the arcs of the graph in the construction above. We accomplish this by replacing each arc (RQ) by an undirected *triangle* RQP , where P is a new node, PQ is green and RQ and RP are red (Fig. 11b). At the nodes R, P, Q the three edges are ordered as indicated in Fig. 11b. The triangles themselves ($K_{n,m}$ in number) can be ordered. We can add $kK_{n,m}$ to the numbers 1, 2 at the edges of the k th rectangle, that indicate the orderings. Thus all $\{\cong_i\}$ become total orders. We have therefore constructed an ordered undirected graph G^* from an arbitrary QBF instance F . We claim that the minimax number of rounds equals the number of green edges in G^* iff F is true.

Let us look at legal PQ -moves, that is moves whose green unfixed edge belongs to a triangle. If this move $(A_r \cup A_g \cup A)$ produces a cycle $(RQPR)$, we can infer the following: The arc (RQ) must belong to $A_r \cup A$ and $A_g \cup A$. This is because $A_r \cup A$ must contain a directed path $(P \cdots Q)$ and $QR \cong_Q QP$. (Recall that QP is the only green edge without a previously fixed direction.) Thus no matter what the response of Player II is to such a PQ -move the arc (RQ) becomes part of A . On the other hand a PQ -move producing a cycle $(QR PQ)$ is never legal. This is because $A_g \cup A$ must contain $\{(PQ), (QR), (RP)\}$ a cycle. The existence of a path $(Q \cdots P)$ in $A_r \cup A$ and the fact that $RQ \cong_R PR \cong_P QP$ force this situation. Thus PQ -moves fix the direction of QR to (RQ) . Finally if Player I were ever to use a QR in the direction (QR) , in some other e -move (e a green unfixed edge), then a response of red by Player II would consume two green edges (i.e., e and PQ).

Now in order for Player I to play as many times as there are green edges in G^* , he must move using the green edges in the triangles and forcing the desired directions. This completes the proof of Lemma 4. \square

Proof of Theorem 4. The theorem now follows by observing that the ordered graph $G = (V, E, \emptyset, \{\cong_i\})$ in Lemma 4 is indeed the conflict graph of a transaction system \mathbf{T} . For each vertex i in V there is a transaction T_i in \mathbf{T} . For each edge $e = ij$ in E , there is an entity x_e updated by both T_i and T_j . If e is red, x_e is stored at site 1, if green at site 2. For the (total) orders \cong_b , we simply order the actions of transaction T_i accordingly. \square

As more-or-less immediate consequences of Theorem 4 and its proof we can obtain complexity characterizations for several special cases. Let us slightly abuse our notation, and use $\text{PREFIX}(\langle \mathbf{T}, \alpha \rangle, b)$ to denote the decision problem:

Is the minimax length of game $\text{PREFIX}(\langle \mathbf{T}, \alpha \rangle)$ larger than b ?

We have the following cases depending on the structure of $\langle \mathbf{T}, \alpha \rangle$ and b .

COROLLARY 4.1. (a) $\text{PREFIX}(\langle \mathbf{T}, \emptyset \rangle, b)$ is PSPACE-complete.

(b) $\text{PREFIX}(\langle \mathbf{T}, \alpha \rangle, b)$ is PSPACE-complete and $\text{PREFIX}(\langle \mathbf{T}, \alpha \rangle, 0)$ is NP-complete, even if \mathbf{T} contains no cross-arcs.

(c) $\text{PREFIX}(\langle \mathbf{T}, \emptyset \rangle, 0)$, if \mathbf{T} contains no cross-arcs, is in P.

Furthermore, (a) and (b) hold even when there are no more than six actions per transaction.

Proof. Note that (a) follows directly from Theorem 4 and (b) can be easily shown by extending the proofs of Lemmas 3 and 4. By minor modifications [7] to the subgraphs of Figs. 8 and 9 we can make the nodes (after substituting triangles for directed edges) have at most degree 6. For case (c) all we have to test for is if $G(\mathbf{T})$ contains a two-color cycle. \square

We finally obtain the following result on the complexity of distributed concurrency control.

COROLLARY 4.2. *Unless $NP = PSPACE$, there is no scheduler for SR , which is both computationally efficient and communication-optimal; even if we restrict T to sets of transactions which are total orders and have six actions each.*

Proof. If such a general scheduler existed, we would have a nondeterministic polynomial-time algorithm for solving the $PSPACE$ -complete problem $PREFIX(\langle T, \emptyset \rangle, b)$, as follows:

On input $\langle T, \emptyset \rangle, b$:

1. *Guess* a schedule in SR , check it in polynomial time.
2. Simulate (in a centralized manner) the operation of the scheduler on this schedule. Whenever a *send-message* instruction occurs, *guess* a delay d , and increase a message count. (The delay d can be chosen to be a number bounded by a polynomial in size of the input).
3. In the end, if more than b messages were used, then report “yes”, else report “no”. \square

5. Conclusions. Our main result shows that concurrency control, an on-line problem clearly in NP (P for serializability) in the centralized case, is $PSPACE$ -complete in the distributed case. This result is quite strong, in that it holds for transaction systems of rather ordinary appearance (e.g., transactions which are total orders with at most six actions each). Also, the negative implications of our result (Corollary 4.2) are quite robust. For example, even if the scheduler is equipped with a powerful oracle belonging anywhere in the polynomial hierarchy, it still cannot minimize communication efficiently, unless the polynomial hierarchy collapses.

In the process of proving this negative result, we have related distributed concurrency control to certain combinatorial games played on graphs. It could be that this connection is of some practical value. There is a more-or-less immediate heuristic for approximating an optimal strategy in the game $CONFLICT$. This heuristic is based on the following purely combinatorial problem:

Given an undirected graph with its edges colored red and green, find a “small” set of edges that have to be deleted in order for the resulting graph to have no two-color cycle.

Acknowledgments. We would like to thank Prof. R. G. Gallager for many helpful discussions and Prof. P. Elias, Dr. A. Bruss, Vassos Hadzilacos and the anonymous referees for their comments on the presentation of these results.

REFERENCES

- [1] P. A. BERNSTEIN AND N. GOODMAN, *Concurrency control in distributed database systems*, ACM Computing Surveys, 13 (1981), pp. 185–223.
- [2] P. A. BERNSTEIN, D. W. SHIPMAN AND J. B. ROTHNIE, *Concurrency control in a system of distributed databases (SDD-1)*, ACM Trans. Database Systems, 5 (1980), pp. 18–51.
- [3] A. K. CHANDRA AND L. J. STOCKMEYER, *Alternation*, Proc. 17th Foundations of Computer Science Conference, 1976, pp. 98–108.
- [4] K. P. ESWARAN, J. N. GRAY, R. A. LORIE AND I. L. TRAIGER, *The notions of consistency and predicate locks in a database system*, Comm. ACM, 19 (1976), pp. 624–634.
- [5] S. EVEN AND R. E. TARJAN, *A combinatorial problem which is complete in polynomial space*, J. Assoc. Comput. Mach., 23 (1976), pp. 710–719.
- [6] C. A. R. HOARE, *Communicating sequential processes*, Comm. ACM, 21 (1978), pp. 666–677.
- [7] P. C. KANELLAKIS, *The complexity of concurrency control for distributed databases*, MIT/LCS/TR-269, Aug. 1981.

- [8] H. T. KUNG AND C. H. PAPADIMITRIOU, *An optimality theory of database concurrency control*, Proc. 1979 SIGMOD, pp. 116–126.
- [9] R. E. LADNER, *The complexity of problems in systems of communicating sequential processes*, Proc. 11th ACM Symposium on Theory of Computing, 1979, pp. 214–223.
- [10] L. LAMPORT, *Time, clocks and the ordering of events in a distributed system*, Comm. ACM, 21 (1978), pp. 558–565.
- [11] C. H. PAPADIMITRIOU, *Serializability of database updates*, J. Assoc. Comput. Mach., 26 (1979), pp. 631–653.
- [12] C. H. PAPADIMITRIOU AND J. TSITSIKLIS, *On the complexity of designing distributed protocols*, manuscript, Aug. 1982.
- [13] D. J. ROSENKRANTZ, R. E. STEARNS AND P. M. LEWIS, *System level concurrency control for distributed database systems* ACM Trans. Database Systems, 3 (1978), pp. 178–198.
- [14] T. J. SCHAEFER, *Complexity of some perfect information games*, J. Comput. System Sci., 16 (1978), pp. 185–225.
- [15] R. S. STEARNS, P. M. LEWIS AND D. J. ROSENKRANTZ, *Concurrency control for database systems*, Proc. 16th Foundations of Computer Science Conference, 1976, pp. 19–32.
- [16] R. H. THOMAS, *A majority consensus approach to concurrency control for multiple copy databases*, ACM Trans. Database Systems, 4 (1979), pp. 180–209.
- [17] J. D. ULLMAN, *Principles of Database Systems*, Computer Science Press, Potomac, MD, 1982 second edition.
- [18] M. YANNAKAKIS, C. H. PAPADIMITRIOU AND H. T. KUNG, *Locking policies: safety and freedom from deadlock*, Proc. 20th Foundations of Computer Science Conference, 1979, pp. 283–287.
- [19] A. C. YAO, *Some complexity questions related to distributive computing*, Proc. 11th ACM Symposium on Theory of Computing, 1979, pp. 209–213.
- [20] *Eden Project Proposal: Research in Integrated Distributed Computing*, Dept. of Computer Science, Univ. Washington, Pullman, TR 80-10-1, October 1980.

UNBOUNDED SPEED VARIABILITY IN DISTRIBUTED COMMUNICATIONS SYSTEMS*

JOHN H. REIF† AND PAUL G. SPIRAKIS‡

Abstract. This paper concerns the fundamental problem of synchronizing communication between distributed processes whose speeds (steps per time unit) vary dynamically. Communication must be established in matching pairs, which are mutually willing to communicate. We show how to implement a distributed local scheduler to find these pairs. The only means of synchronization are boolean "flag" variables, each of which can be written by only one process and read by at most one other process.

No global bounds in the speeds of processes are assumed. Processes with speed zero are considered dead. However, when their speed is nonzero then they execute their programs correctly. Dead processes do not harm our algorithms' performance with respect to pairs of other running processes. When the rate of change of the ratio of speeds of neighbour processes (i.e., relative acceleration) is bounded, then any two of these processes will establish communication within a constant number of steps of the slowest process with high likelihood. So, our implementation has the property of achieving relative real time response. We can use our techniques to solve other problems such as resource allocation and implementation of parallel languages such as CSP and Ada. Note that we do not have any probability assumptions about the system behaviour, although our algorithms use the technique of probabilistic choice.

Key words. distributed networks, communicating sequential processes, handshake communication, synchronization, real-time response, probabilistic choice, randomized algorithms

1. Introduction. Recently, Reif and Spirakis [1984] showed how to achieve inter-process communication with real time response using probabilistic synchronization techniques, assuming that the speeds of all processes were bounded between fixed nonzero bounds. This leads to real time resource allocation algorithms and real time implementation of message passing in CSP (see Reif and Spirakis [1984, Appendix I, II] and also Reif and Spirakis [1982a], [1984a]).

In this paper (a preliminary version of the paper appeared in Reif and Spirakis [1982]) we assume *no global bounds on the processors' speeds*. Their speeds can vary dynamically from zero to an upper bound which may be different for each processor, and not known by the other processors. We allow a possibly infinite number of processes, so that there may not be a global upper bound on the speeds. Processes may die (have zero speed) but when they have nonzero speed then we assume they execute their programs correctly. We are interested in *direct* interprocess communication (rather than packet switching) which is of the form of *handshake* (rather than buffered), as in Hoare's CSP (Hoare [1978]). The essential technique that we utilize is that of *probabilistic choice*. This technique, introduced to synchronization problems by Rabin [1980], Lehman and Rabin [1981] and Francez and Rodeh [1980], was also utilized in our previous work.

The use of probabilistic choice in the algorithms leads to considerable improvements in the space and time efficiency (Rabin [1980], Reif and Spirakis [1984]); we feel that this may be because of the locality of the decisions and because complex sequences of the processes' steps prohibiting communication have very low probability of occurrence. We assume that each process makes probabilistic choices independent

* Received by the editors May 3, 1983, and in final revised form March 12, 1984. This paper appeared in the Ninth ACM Symposium on Principles of Programming Languages, January 25-27, 1982, Albuquerque, New Mexico. This work was supported in part by the National Science Foundation under grants NSF-MCS82-00269 and NSF-MCS83-00630, and by the Office of Naval Research under contract N00014-80-C-0647.

† Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts 02138.

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

of other processes. We also introduce new *adaptive techniques*, where the processes estimate the speeds of neighbour processes and select processes to communicate with probabilities depending on the speeds, penalizing the slowest processes. These adaptive techniques do not seem to have ever been utilized in the previous synchronization literature.

This paper proposes a new high level synchronization construct associated with interprocess communication. The construct is implemented very efficiently by the use of boolean flag variables, each of which can be written by only one process and read by at most one other process. (We do not use any standard high level synchronization construct such as shared variables with a mutual exclusion mechanism since these have no known efficient implementation. There is not even any known bounded time implementation of a mutual exclusion mechanism when processes run on different processors.)

If processes are bounded in speed then it is natural to define real time response to be a response to a communication request that uses no more than constant number of units of real time. This measure is inapplicable in our case in which there is no global upper bound and no nonzero lower bound on speeds. Thus we introduce the notion of *relative real time response* which is establishment of communication between *any pair* of neighbouring processes within constant number of local rounds. (A *local round* of neighbour processes, i, j is the minimum time interval which contains at least one step of each process and exactly one step of at least one of i, j .) Local rounds are calculated relative to given time intervals. Let $t_s \geq 0$ be a particular time instant. The *local round of processes i, j beginning at t_s* (and ending at $t_e \geq t_s$), is the smallest time interval $[t_s, t_e)$ such that

- (1) $[t_s, t_e)$ contains at least one (full) step of both i, j .
- (2) $[t_s, t_e)$ contains exactly one full step of at least one of i, j .

(Note: The next local round with respect to a time interval Δ , starting at t_s , is the local round of i, j beginning at t_e .)

We achieve communication between any pair of neighbouring processes within constant number of local rounds by our probabilistic algorithms with some probability of error which can be made arbitrarily low. The best *deterministic* symmetry algorithms which attempt to form matchings in distributed systems have a relative response depending linearly on the network's diameter.¹

The paper is organized as follows: In the next section we define our model for communication. In § 3 we discuss applications of this model. In § 4 we give a relative real time implementation of communication in this model. In § 5 we give correctness properties of our proposed implementation and time analysis.

2. The model VS-DCS (Variable Speed Distributed Communication System).

2.1. The model. We develop here a theoretical model related to, but more general than, the Distributed Communication System (DCS) of Reif and Spirakis [1984]. A detailed description of the fundamental issues can be found in Reif and Spirakis [1984], [1984a].

We assume a possibly infinite collection of processes $\pi = \{1, 2, \dots\}$. Events of the system are totally ordered on the real-time line $[0, \infty)$. The processes of π are

¹ Also Arjomandi, Fischer and Lynch [1981] have shown that some synchronization problems which are global (in contrast to our problem) cannot be done in real time and require time proportional to the logarithm of the total number of processors in the network. A typical situation where this could occur is the problem of detecting connected components of processes whose speeds are within given positive bounds.

asynchronous; their speeds may dynamically vary arbitrarily over time and may even be 0. The processes have no access to any global clock giving the time.

We assume that the effect of a read or write is instantaneous and that these events occur at distinct time instants, so there are never any read/write conflicts. In general, a *step* of a process is a finite time interval Δ in which a single instruction is instantaneously executed at the last moment of Δ . A process i is the slowest of two processes i, j during a step $s = [t_1, t_2]$ of i , if at least one step of j is a proper subset of s . We can extend this notion to bigger time intervals, by counting steps of i and j contained in those time intervals. Each process consists of a fixed set of synchronous parallel subprocesses. (The subprocesses of any given process have the same speeds.) The asynchronous processes wish at various times to communicate with other processes but have no means of communication except via the communication system. This is implemented by many *poller* subprocesses (seven for each target process) which are synchronous with themselves. We assume a fixed connections graph H which is undirected and has the set π as its vertex set. An edge $\{i, j\}$ indicates that process i is physically able to communicate with process j (but not necessarily willing to). H is assumed to have finite valence. We also assume for each time t the *willingness digraph* G_t , which indicates the willingness of a given process i at a given time t . (We indicate this by the edge $i \rightarrow_t j$ and say i is a *willing neighbour* of j .) Note that $i \rightarrow_t j$ only if $\{i, j\} \in H$. Let $i \leftrightarrow_t j$ if $i \rightarrow_t j$ and $j \rightarrow_t i$. The edges of the graph G_t are stored distributedly so that the edges departing from a given process are only known to that process. We assume that the out-degree of each vertex of G_t is upper bounded by a fixed constant v .

For each time interval Δ on $[0, \infty)$ let $i \rightarrow_{\Delta} j$ if $i \rightarrow_t j$ for all $t \in \Delta$ and let $i \leftrightarrow_{\Delta} j$ if both $i \rightarrow_{\Delta} j$ and $j \rightarrow_{\Delta} i$.

For each $t \geq 0$ the (possibly infinite) digraph M_t with vertex set π and directed edges $i \rightsquigarrow_t j$ denotes which processes *open communication* to which other processes at time t . We denote $i \rightsquigarrow_t j$ if both $i \rightarrow_t j$ and $j \rightsquigarrow_t i$. Thus $i \rightsquigarrow_t j$ denotes i, j *achieve mutual communication* at time t . M_t is the digraph that implementations of distributed synchronization achieve. Also, we extend the notation to intervals Δ on $(0, \infty)$ as for G_t . We assume that

(A1) Two way communication between any two processes $i, j \in \pi$ requires only one step of i and j . (Thus, processes communicate in short “bursts”.)

(A2) If $i \rightarrow_{t_1} j$ and not $i \rightarrow_{t_2} j$, $t_2 > t_1$, then $i \rightsquigarrow_{\Delta} j$ for some $\Delta \in [t_1, t_2]$, where Δ contains at least one step of each i and j (i.e. processes can withdraw willingness to communicate only after communication between i and j has been established and completed).

In practice, assumption A2 can be easily circumvented. Suppose a process i is initially willing to communicate with process j , but later decides that it is no longer interested in communicating with j , before mutual communication has been achieved. By assumption A2, process i must not withdraw willingness to communicate until the implementation has achieved mutual communication between i and j . However, at this time a null value can simply be sent.

We wish implementations to be *proper* in the sense that

(a) $i \rightsquigarrow_t j$ only if $i \leftrightarrow_t j$ (neighbours try to speak only if they are mutually willing to).

(b) \rightsquigarrow_t must be a partial matching: If $i \rightsquigarrow_t j$ then not $j' \rightsquigarrow_t i$ for any j' in $\pi - \{j\}$. (No process is allowed to achieve communication with more than one neighbour at the same time.)

A process with speed 0 is *dead*, and otherwise is *awake*. We assume that processes can suddenly die but when they are awake they execute their programs correctly. We furthermore assume that each process has a fixed upper bound on its speed which may be different from the other processes and not known to them.

For any two processes i, j take two consecutive steps of i , on time intervals Δ_1 and Δ_2 . Let S_1 and S_2 be the number of steps of j overlapping with Δ_1, Δ_2 respectively. Then, the *relative acceleration of j with respect to i* is $(|S_2 - S_1|)/2$. Let us denote it by α_{ji} . Let α , the *relative acceleration bound* for processes i and j , be the worst case value of the maximum of α_{ij}, α_{ji} over all times. The correctness of our synchronization algorithms does not depend on whether processes are acceleration bounded; however, we assume fixed acceleration bound α in our time complexity analysis. Thus the relative acceleration of one neighbour with respect to another is bounded by a constant α or can be $-\infty$ if the process dies.

We assume an “adverse” oracle \mathcal{A} which at time 0 chooses the speeds of all the processes for all times. \mathcal{A} is also able to dynamically change the willingness relation \rightarrow_t (subject to assumption A2) so as to achieve the worst case performance of the implementation of VS-DCS. Note that, in practice, we can assume each process has a *director subprocess* which dynamically changes the willingness relation \rightarrow_t and at time 0 determines the speed of that process for all times. Thus, in this case, the oracle \mathcal{A} is defined *distributedly* by the director subprocesses. It should be noted that the oracle \mathcal{A} is useful to us because it may be explicitly used to define the worst case performance of the system, when communication requests happen at times most difficult for our implementation and speeds vary in the most difficult way. Thus, if we prove that the system has a certain performance for a worst case oracle, then we have upper bounds on the performance of any set of director subprocesses. More general adverse oracles (which may alter later speeds of processes according to past success or failure of communication) are defined in Hart, Sharir and Pnueli [1982]. Our implementations may fail to meet some of their requirements if these more general oracles are allowed. We feel, however, that our notion of adversary is adequate for practical applications.

The following communication primitives can be implemented by the poller subsystem of each process: (In practice, the director may not get an immediate answer but may proceed to some other instruction and later a time slot for communication will be arranged by the poller subsystem. Of course, if successful two-way communication is achieved, we assume both processes are aware of the success of communication.)

ATTEMPT-COM $_i(j)$: indicates that the director of i wishes to communicate with the director of process j .

CANCEL-COM $_i(j)$: indicates that the director of i wishes no longer to communicate with j .

The precise semantics of ATTEMPT-COM and CANCEL-COM are given by the willingness relation $\rightarrow_t \subseteq \pi \times \pi$.

Note that assumption A2 implies that the oracle \mathcal{A} can withdraw willingness to communicate only after communication has been established. Thus, if ATTEMPT-COM $_i(j)$ is called at time t_1 and CANCEL-COM $_i(j)$ is called at time t_2 ($t_2 > t_1$) then communication was established for some t on $[t_1, t_2)$. (In fact, our implementations do not really require this assumption but only require that the willingness to communicate will not be cancelled before some constant number of steps. However, the assumption A2 given here, considerably simplifies our analysis.)

2.2. Complexity of VS-DCS. We assume here a global constant α . We say \mathcal{A} is *tame* for i, j on time interval Δ if the pairs $\{(i, j)\} \cup \{(i, k) | k \text{ is a neighbour of } i\} \cup \{(j, k) | k$

is a neighbour of j are each relative acceleration bounded by α on the time interval Δ .

For every ε on $(0, 1)$ let the ε -response $S(\varepsilon)$ be the minimum integer >0 such that for every pair of neighbours i, j and each time interval Δ and for every oracle \mathcal{A} which is tame for i, j on Δ , if $i \leftrightarrow_{\Delta} j$ and the number of steps of the slowest of i, j within Δ is $\geq S(\varepsilon)$ then there exists a subinterval $\Delta' \subseteq \Delta$ containing at least one step of the slowest of i, j such that

$$i \leftrightarrow_{\Delta'} j \text{ with probability } \geq 1 - \varepsilon.$$

Intuitively $1 - \varepsilon$ gives a lower bound on the probability of establishing communication in the case process i issues an ATTEMPT-COM (j) at the beginning of Δ , and after $S(\varepsilon)$ steps it calls CANCEL-COM (j). (Note that we presume here that i and j and their neighbours have relative acceleration bound α , only during the interval Δ ; at other times this acceleration bound may be violated, and furthermore the acceleration bound α need not hold for other processes even during the interval Δ .)

We consider an implementation to be *relative real time* if for all constants ε on $(0, 1)$, the relative ε -response $S(\varepsilon)$ is upper bounded by a constant, independent of any global measure of the willingness digraph G_t (such as $|\pi|$ or any function of it) and dependent only on the constant maximum valence v of the vertices of G_t , and on the bound α on the relative acceleration. Note that relative real time response *does not imply* that communication is guaranteed within any time interval but instead it is guaranteed within a bounded number of steps of the processes with high likelihood (this is because processes can slow down arbitrarily). In § 4 we show how to implement the VS-DCS so that relative real time response is achieved.

3. Applications. The primitives ATTEMPT-COM, CANCEL-COM are powerful enough to supply real time implementations of synchronization constructs of high level parallel languages such as CSP and ADA.

The following proposition will be useful in the applications.

PROPOSITION 3.1. *If the oracle \mathcal{A} is tame for processes i, j on an interval Δ which includes at least $\delta_0 X + \alpha X^2/2$ steps of either i or j , (where δ_0 is the speed ratio of processes i, j in the beginning of Δ), then Δ includes at least X local rounds.*

Proof. Consider the number of local rounds to be the “time” during which a fictitious moving object with initial speed δ_0 and acceleration α moves a distance equal to the maximum number of steps done by either i or j or Δ . \square

3.1. Real time resource granting systems with process failure. Previously, in Reif and Spirakis [1982a], [1984], [1984a] we utilized the more restricted DCS system which does not allow process failures to implement a real time resource granting system. In this paper, we can cope with sudden process failures (zero speeds). In this case, the process governing a resource will first get an estimate δ_0 on the speed of a process granted the resource and then it will attempt to communicate for $\delta_0 S(\varepsilon) + \alpha S^2(\varepsilon)/2$ of its steps with the process granted the resource. (Note that δ_0 will be 1 if the process governing the resource is the fastest.) By Proposition 3.1 the above interval should be enough for the process which has been granted the resource to respond. If that process does not respond, the resource governing process may reclaim the resource. If a resource allocator dies, then other processes can play its role.

3.2. Relative real time implementation of CSP and ADA’s synchronization constructs. In a typical stage during execution, the processes comprising a CSP program may be divided into two classes: those busy with local computations and those waiting

for a partner to communicate with. A distributed guard scheduler can be implemented by using the poller subprocesses of the relative real time VS-DCS system.

The *Communicating Sequential Processes* (CSP) language was defined by Hoare [1978] for concurrent programming. The language has elegant synchronization constructs:

(1) An *output command* of the form $i!u$ where i is a process and u is a value which i receives.

(2) An *input command* of the form $i?x$ where i is a process which sends a value which is assigned to variable x .

CSP also allows *alternative statements* which consist of a sequence of guarded commands of the form $G \rightarrow C$ where the *guard* G is a list of booleans followed by at most one input command and C is a command list. We assume here the extension of CSP given in Bernstein [1981], which allows G to be a list of booleans followed by at most one input or output command. An alternative statement is executed by indeterminately choosing a guard which is satisfied (by executing its elements from left to right) and then executing the corresponding command list. If no guard is satisfied, the alternative statement *fails*. CSP also allows a *repetitive statement* allowing repeated execution of an alternative statement until it fails.

Thus, the essential problem in implementing CSP is to synchronize execution of input and output commands. Let v be the maximum number of guards appearing in any alternative or repetitive statement; we assume that v is constant relative to the total number n of processes. Let J be the event: For a given alternative statement, the execution either determines a satisfied guard and executes the corresponding command list, or determines that no guard is satisfied and makes a failure exit from the statement. A CSP implementation is *relative real time* if there exists a positive integer l (which is independent of the number of processes n) such that J takes at most l steps of all processes associated with the guards of the alternative statement.

For a process i to execute an output command $j!u$, process i must execute the communication command $\text{ATTEMPT-COM}_i(j)$. Also, to execute an input command $i?x$, process j must execute the communication command $\text{ATTEMPT-COM}_j(i)$. If successful communication is established between i and j , then during that time process j transmits value u to variable x in process i . Processes i, j then execute $\text{CANCEL-COM}_i(j)$ and $\text{CANCEL-COM}_j(i)$, respectively. (Note that if processes i or j happen to die at this point, before cancelling communication, then successful communication cannot be made with them during the time speed is 0, so it is not essential for the communication request $i \leftrightarrow j$ to be cancelled.) We assume here an underlying relative real time VS-DCS implementation, with relative ε -response $S(\varepsilon)$, where ε is a system-wide constant which may be fixed to any arbitrarily small constant on the interval $(0, 1)$.

Let S be an alternative statement with guarded input and output commands, say G_1, \dots, G_s with $s \leq v$. To execute the statement S , process i first executes the booleans appearing in each guard. If no guard is satisfied, process i must then exit the statement S with failure. Otherwise, let R be the set of processes appearing in those guards of S all of whose booleans evaluate to true. Process i must then execute $\text{ATTEMPT-COM}_i(j)$ for each process $j \in R$. At the first time a communication is established between i and some willing process $j \in R$, process i must immediately execute $\text{CANCEL-COM}_i(j')$ for each $j' \in R$ and then execute the command list associated with the now satisfied guard in the statement S . Note that the above will take at most $l = \alpha S^2(\varepsilon)/2$ steps of i , with probability at least $1 - \varepsilon$, by Proposition 3.1 and the definition of $S(\varepsilon)$.

Also, in ADA, two-way communication between pairs of tasks is allowed to synchronized time instances called *rendezvous*. An *accept* statement of the form *accept f(-)* appearing in task T_1 indicates that T_1 is willing to rendezvous at f with any task of similar argument type. The task T_2 may execute a *call* statement of the form *f(-)* indicating that T_2 is willing to rendezvous with T_1 at the accept statement containing f . ADA also allows for *selective accept statements* containing multiple accept statements, one of which must be nondeterministically chosen to execute. (This is similar to the *select* statement of CSP.)

ADA's tasks may be implemented by processes whose speeds vary dynamically. (Processes may even fail for various time intervals.) The key implementation problem is to synchronize task rendezvous within relative real time, in spite of the dynamic speed variations. These processes may be connected within a distributed network whose transmission channels may also have variable speeds or fail. Unreliable transmission channels can be viewed as processes which are connected with the processes of the network via reliable communication channels.

We assume that it is possible to analyze (perhaps by data flow analysis) an ADA program to determine an undirected (possibly infinite) connections graph whose nodes are all the tasks possibly created by the ADA program and edges are the possible task communication pairs. Since an actual implementation will have in its hands at any time only a finite set of processes we assume that only the currently active tasks have an associated implementing process and a scheduler devotes a currently free process to a given newly created task. A garbage collector removes the implementing process from the deleted task and places it back to the free list of processes. These implementation techniques were developed by Dennis and Misunas [1974] for real time implementation of data flow machines. They correspond to the *initiate* and *abort* statements, which appeared in old ADA versions.

The synchronization facilities of our VS-DCS system provide (by the use of the ATTEMPT-COM and CANCEL-COM primitives) a real time implementation of the *accept* and *call* statements. A version of the *active* statement can be implemented so that deleted tasks and tasks implemented by nontame processes can be detected by their neighbours in real time with some (arbitrarily small) error probability. This can be done in our VS-DCS system by repeatedly attempting communication with neighbouring processes. Finally, the *symmetry* and *locality* of the VS-DCS implementation (due to its probabilistic nature) may help in eliminating the tradeoff between generality of expression and ease of implementation in ADA.

The *probabilistic fairness* guaranteed by the algorithms of the pollers eliminates the danger of bottlenecks which could be created if conventional techniques were used (a new task which centralizes requests and keeps track of busy server tasks is one of the conventional proposed solutions). Most of the problems which VS-DCS could cure are discussed in Mahjoub [1981], and Francez and Rodeh [1980]. A probabilistic solution to some of the discussed problems was given also in Francez and Rodeh [1980], but no discussion about real-time properties was done and neither the problem of speed variations nor that of dying processes was addressed.

4. Relative real time implementation of VS-DCS.

4.1. Intuitive description of the algorithm. We utilize $7v + 1$ synchronized parallel processes to implement the poller subprocess of each process i . These are the *communicators* $cp_1^i, cp_2^i, \dots, cp_{2v}^i$, the *speed estimators* ep_1^i, \dots, ep_v^i and the *judge* subprocesses $jp_0^i, jp_1^i, \dots, jp_{4v}^i$ of process i . Each pair of the communicators $cp_{k'}^i, cp_{k''}^i$ where $k' \bmod v = k'' \bmod v = k$, is devoted to communication with the k th neighbour. Each

estimator is used to continuously update an estimation of the speed of a particular neighbour process. The collection of judges has the task to select under certain conditions one communicator and to give to him the right to open the communication channel of process i to its corresponding neighbour.

We frequently use the technique of *handshake* by which we mean that each subprocess modifies a flag variable observed by the corresponding neighbour subprocess. Process contention between synchronized subprocesses is easy to implement (we can allow each to take a separate step in a small round).

Our algorithm for the k th communicator subprocess cp_k^i ($1 \leq k \leq 2v$) of the poller of process i proceeds as follows:

Let $k' = k \bmod v$. At every time $t \geq 0$, $E^i(1), \dots, E^i(D^i)$ is the list of targets of edges of G_i departing from $i \in \pi$, and $D^i \leq v$ is the current number of targets. Those variables are dynamically set by the oracle \mathcal{A} and they are the neighbours to which process i is willing to open communication at time t . Note that, by our assumption A2, oracle \mathcal{A} can remove an element from the list of targets (of edges of G_i departing from i), only immediately after communication with this target has been established. We furthermore assume that \mathcal{A} modifies the list of targets (and D_i) instantaneously. As a consequence, our algorithms are not confused by dynamic removal of edges of G_i , (in particular, improper communication can never happen). The subprocess cp_k^i deals with the $E^i(k')$ neighbour. If $k \leq v$, then cp_k^i is an *asker* subprocess, else it is a *responder* process. cp_k^i must first handshake with the corresponding subprocess of process $E^i(k')$ to which node i wishes to communicate. We need two handshake subprocesses (ask, respond respectively) per neighbour because of a certain asymmetry in the handshake (some subprocess has to be the first to modify a flag). In particular the asker procedure initiates the handshake and the responder answers to it.

Next, we wish to find a time slot in which the two neighbours may communicate. Because there may be contention among other processes j which also wish to communicate with i (and consequently, other askers or responders of node i also will handshake) we must resolve the contention by a fair judge. To do this, we add the process cp_k^i to a queue and the collection of judge synchronous subprocesses of poller i takes a random process from this queue and allocates time slots for communication attempts. To ensure that slower neighbours do not utilize any more total time on the average than faster neighbours during communication attempts, we weight the probabilities of subprocesses to be chosen from the queue by the factor

$$\frac{1}{\Delta_{ik} \cdot \omega(\Delta_i)},$$

where Δ_{ik} is the current estimation of the steps of process i per step of process k , supplied by the estimator ep_k^i , and $\omega(\Delta_i) = \sum_j (1/\Delta_{ij})$.

The judge subprocesses are organized in a balanced binary tree of height $\log(2v) + 1$. Any time a random process is to be selected from the queue, the *supreme judge subprocess* jp_0^i enables the tree of the rest of the judges to conduct a tournament between the waiting processes in the queue and to select a winner with the above stated probability. In that way, the total number of steps needed for a winner to be selected is $O(\log(2v))$. (Note that less efficient ways of using a random number generator to choose one waiting process from the queue could take $O(v)$ steps of process i , because of the form of the weight factor in the probabilities.)

Our technique of weighing the probability that subprocesses are chosen from the queue, has the effect that each subprocess in the queue attempts to communicate on the average $1/(2v + v^2/\alpha)$ of the total time. (This is proved in our analysis following

the formal description of the algorithms.) If a process is chosen by the judges but the communication is not established, the algorithm requires that subprocess to initiate another handshake with its partner (to check if they are still mutually willing to communicate and to synchronize steps). Then, it is again added to the queue to be given another chance to establish communication. This process proceeds until either the director of i withdraws its willingness to communicate with $E^i(k')$ or until it establishes communication.

Note that the time slots for communication attempts, allocated by the supreme judge jp_0^i to each selected communicator, take into account the current speed ratio of the process i and its neighbour corresponding to that communicator, adjusted by a factor related to the worst-case acceleration and the $\log_2 2v$ delay in the process of choosing a winner, to give the opportunity of at least one step overlap in time of process i and its neighbour, if their corresponding channels are both open.

We introduce random waits which help subprocess cp_k^i to eliminate the possibility of schedules set-up by the adverse oracle \mathcal{A} to have always a particular subprocess arrive first in the queue and win the contest. This possibility is eliminated since we have assumed that the oracle sets the speeds at time 0 and cannot affect the independent random choices done by the processes. Also, we assume that the random number generator RANDOM(0, 1) of each subprocess yields truly random real numbers, uniform on the interval [0, 1], and independent of the random numbers generated by any other subprocess.

Note that we trade computation effort (parallelism) in a node to achieve reliable communication. This parallelism is limited because of the bounded valence v of the graph G_t . We can always simulate these synchronous techniques. This will reduce the effective speed of each subprocess by only a factor of $7v+1$.

4.2. The algorithms of the poller subprocesses. In each process $i \in \pi$, we assume synchronous subprocesses

askers: $cp_1^i, cp_2^i, \dots, cp_v^i$,
 responders: $cp_{v+1}^i, cp_{v+2}^i, \dots, cp_{2v}^i$,
 estimators: ep_1^i, \dots, ep_v^i ,
 judges: $jp_0^i, jp_1^i, \dots, jp_{4v}^i$.

The askers and the responders are the communicators.

In the following algorithm, executed by each of the communicator subprocesses cp_k^i , $1 \leq k \leq 2v$, we implement the queue of process i by an array $Q^i(k)$, $1 \leq k \leq 2v$. $Q^i(k) = 1$ holds just if cp_k^i waits in the queue. Another array of binary values, marriage $^i(k)$, $1 \leq k \leq 2v$, is used to indicate which communicator subprocess currently holds process's i channel and attempts communication. When the predicate marriage $^i(k) = 1$ is true, then cp_k^i attempts communication at that time. The algorithms have designed so that at most one of the marriage $^i(j)$, $1 \leq j \leq 2v$, is set at any time. We now present the algorithm for the communicator subprocesses:

```

process  $cp_k^i$ 
  WHILE true DO
    IF  $D^i \geq k$  THEN
      BEGIN
         $W \leftarrow c_1 \cdot \text{RANDOM}(0, 1)$ 
        DO [  $W$  ] noop
        IF  $k \leq v$  THEN ASK $_i(E^i(k))$  ELSE RESPOND $_i(E^i(k))$ 
        COMMENT: Add  $k$  to queue
      
```

```

     $Q^i(k) \leftarrow 1$ 
    WHILE marriagei(k) = 0 DO a noop
    ESTABLISH-COM ( $E^i(k)$ ,  $c_2\Delta_{ik}$ )
    marriagei(k)  $\leftarrow$  0
  END
OD

```

A *noop* is a single unit step in which no operation is executed. The constants c_1 and c_2 are as follows:

$$c_1 = 2(2v+1)c_2, \quad c_2 = 4(\alpha\beta + 1), \quad \beta = 6 \log(2v).$$

The speed estimators ep_k^i , $1 \leq k \leq 2v$, execute the following algorithm. The algorithm continuously does a handshake in order to estimate the speed ratio between the process k to which the handshake is attempted, and the process i , of which the speed estimator is a subprocess.

```

process  $ep_k^i$ 
  DO FOREVER
     $F_{ik} \leftarrow 1$ 
    LOOP UNTIL  $F_{ki}$  is 1
  A:  $F_{ik} \leftarrow 0$ ;  $s \leftarrow$  CURSTEP
    LOOP UNTIL  $F_{ki} = 0$ 
  B:  $\Delta_{ik} \leftarrow \frac{\text{CURSTEP}-s}{2}$ 
OD

```

Note that F_{ik} is a flag set by i , read by k . We assume at time 0, F_{ik} initialized to 0. The special register CURSTEP gives the current step of process i . We assume that a step consists of an elementary statement of the algorithms; ep_k^i 's execution assures that Δ_{ik} is (within a factor of 2) the actual speed ratio of processes i and k , since from step A to step B the fastest of the partners does CURSTEP- s steps and the slowest does 2 steps.

4.3. The algorithms of the judge subprocesses. The algorithm of the supreme judge is

```

process  $jp_0^i$ 
  WHILE true DO
    IF queue  $Q^i$  not empty THEN
      BEGIN
        Use the tree of judges to select a random element  $k$  of the queue  $Q^i$  with
        probability
        
$$\frac{1}{\Delta_{ik} \cdot \omega(\Delta_i)}$$

        COMMENT: delete  $k$  from  $Q^i$ 
         $Q^i(k) \leftarrow 0$ 
        marriagei(k)  $\leftarrow$  1
        WHILE marriagei(k) = 1 DO noop
      END
    OD
  OD

```


Note that the supreme judge triggers the operation of the tree of judges. In each level, the winners of the previous level are paired up and half of them are selected. The judge subprocesses of each level execute in parallel synchronously. Finally, the jp_0^i accepts the choice of the root of the tree of judge subprocesses to be the communicator which is going to attempt communication. The supreme judge removes this winner subprocess from the queue Q^i by setting $Q^i(k)$ to 0 and allows the winner to attempt communication (so as to use process's i channel) by setting $\text{marriage}^i(k)$ to 1. Note that we can test if Q^i is empty by keeping a counter of the number of elements in Q^i .

We now give the algorithms of the judges.

```

process  $jp_0^i$ 
  WHILE true DO
    IF  $Q^i$  is not empty THEN
      BEGIN
        FOR level = 1,  $\dots$ ,  $\log(2v) + 1$  DO
          BEGIN
             $L^i \leftarrow$  level; do 6 noops
          END
           $k \leftarrow$  choice $i$ ( $4v$ )
           $Q_k^i \leftarrow 0$ 
           $\text{marriage}^i(k) \leftarrow 1$ 
          WHILE  $\text{marriage}^i(k) = 1$  DO noop
        END
      END
    OD
  
```

The rest of the judges are organized in a full binary tree of $4v$ nodes. The leaves are the processes jp_1^i, \dots, jp_{2v}^i . Each internal node $m \in \{2v+1, \dots, 4v\}$ has two children LCHILD(m), RCHILD(m). The root is the process jp_{4v}^i . Each jp_m^i has its level stored in MYLEVEL(m).

```

process  $jp_m^i$ 
  IF MYLEVEL( $m$ ) =  $L^i$  THEN
    BEGIN
      IF  $L^i = 1$  THEN
        BEGIN
          choice $i$ ( $m$ )  $\leftarrow m$ 
           $\text{marriage}^i(m) \leftarrow 0$ 
          IF  $Q^i(m) = 0$  THEN  $\text{sum}^i(m) \leftarrow 0$  ELSE  $\text{sum}^i(m) \leftarrow 1/\Delta_{im}$ 
        END
      ELSE
        BEGIN
           $r \leftarrow$  RANDOM(0, 1)
           $m_1 \leftarrow$  LCHILD( $m$ )
           $m_2 \leftarrow$  RCHILD( $m$ )
           $\text{sum}^i(m) \leftarrow \text{sum}^i(m_1) + \text{sum}^i(m_2)$ 
          IF  $r < \text{sum}^i(m_1)/\text{sum}^i(m)$ 
            THEN choice $i$ ( $m$ )  $\leftarrow$  choice $i$ ( $m_1$ )
            ELSE choice $i$ ( $m$ )  $\leftarrow$  choice $i$ ( $m_2$ )
        END
      END
    END
  
```

RANDOM (0, 1) is a uniform random number generator which returns a random real between 0 and 1. We assume each of the outputs of RANDOM (0, 1) is independent of any other output. Note that each judge subprocess which is not a leaf uses RANDOM (0, 1) to select one of the choices of its children with conditional probability $\text{sum}^i(m)/(\text{sum}^i(m_1)+\text{sum}^i(m_2))$ where m_1, m_2 are the children of m .

LEMMA 4.1. *It takes $\beta = 6 \log(2v)$ steps for the tree of judges to select a winner from the queue Q^i .*

Proof. The judges of each level work synchronously in parallel and each does at most 6 steps, per iteration of their loop. Since the tree has a height of $\log(2v)$, the total number of steps required is $\beta = 6 \log(2v)$. \square

LEMMA 4.2. *If $Q^i(k) = 1$ then the probability that the winner is communicator cp_k^i is at least $1/(\Delta_{ik} \cdot \omega(\Delta_i))$.*

Proof. The probability that cp_k^i will be selected is the product of the conditional probabilities that cp_k^i will be selected in each node of the path from k to $4v$, which is the root. We shall follow an inductive argument on the level of nodes in the tree. Observe that after execution of the program of process jp_m^i , the variable $\text{sum}^i(m)$ is the sum of $\text{sum}^i(j)$ over all j 's which are leaves of the subtree rooted at m .

Claim. Let k be any of the leaves of the subtree rooted at m . Let $\omega(m)$ be the sum of $\text{sum}^i(j)$ over all j 's which are leaves of the subtree rooted at m , given that $Q^i(k) = 1$. The probability that choice ^{i} (m) is k is equal to $1/(\Delta_{ik} \cdot \omega(m))$.

Proof of Claim. Let m be a node whose children are leaves. In the program that the process jp_m^i executes, the probability that the left child is selected is

$$\frac{\text{sum}^i(\text{LCHILD}(m))}{\text{sum}^i(\text{LCHILD}(m)) + \text{sum}^i(\text{RCHILD}(m))}$$

The probability that the right child is selected is

$$\frac{\text{sum}^i(\text{RCHILD}(m))}{\text{sum}^i(\text{LCHILD}(m)) + \text{sum}^i(\text{RCHILD}(m))}$$

Let the *level* of a node be its distance from the leaves plus 1 (the leaves are at level 1). Let us now assume the claim true for any node of the level $l-1$. Let m be any node of level l , with children m_1 and m_2 at level $l-1$. Let k be any of the leaves of the subtree rooted at m . The probability that choice ^{i} (m) is k , is equal to the probability that either choice ^{i} (m_1) or choice ^{i} (m_2) is equal to k and that k is furthermore selected by process jp_m^i . Since k is a descendant of m , it can be a descendant of only one of m_1, m_2 . Without loss of generality, let us assume that k is a descendant of m_1 . By the induction hypothesis, the probability that choice ^{i} (m_1) = k , is

$$\frac{1}{\Delta_{ik} \cdot \text{sum}^i(m_1)}$$

By the code of jp_m^i the probability that choice ^{i} (m) = choice ^{i} (m_1) is

$$\frac{\text{sum}^i(m_1)}{\text{sum}^i(m_1) + \text{sum}^i(m_2)}$$

The probability that choice ^{i} (m) = k is equal to

$$\frac{1}{\Delta_{ik} \cdot (\text{sum}^i(m_1) + \text{sum}^i(m_2))}$$

which is the product of the above two probabilities. But $\text{sum}^i(m_1) + \text{sum}^i(m_2) = \text{sum}^i(m) = \omega(m)$. \square

To finish the proof of the lemma, we apply the result of the claim for m being the root of the tree and note that

$$\omega(m) \leq \omega(\Delta_i). \quad \square$$

4.4. Low level synchronization procedures. The following are the low level synchronization procedures used by the poller algorithms:

```

procedure aski (target)
  BEGIN
    Qi,target ← 1;
    WHILE Atarget,i = 0 DO noop
    Qi,target ← 0;
    WHILE Atarget,i = 1 DO noop
  END

```

The setting of the flag $Q_{i,target}$ to 1 means that i asks the target. If the target detects $Q_{i,target} = 1$ then it answers positively by setting $A_{target,i} = 1$. Both partners reset these flags to 0 at the end of procedure ask and respond. We assume at time $t = 0$ these flags are initially 0.

```

procedure respondi (asker)
  BEGIN
    LOOP UNTIL Qasker,i = 1
    BEGIN
      Ai,asker ← 1;
      WHILE Qasker,i = 1 DO noop
      Ai,asker ← 0;
    END
  END

```

Let s be the maximum number of steps we are allowed to keep the channel open before we fail. We finally present the code for the procedure ESTABLISH-COM_i (target, s). During its execution process i opens its channel to process target.

```

procedure ESTABLISH-COMi (target, $s$ )
  BEGIN
    OPEN CHANNELi,target
    DO  $s-2$  noops
    CLOSE CHANNELi,target
  END

```

The procedure OPEN CHANNEL _{$i,target$} results in the appearance of $i \rightsquigarrow target$ at the time of its execution, and CLOSE CHANNEL _{$i,target$} sets $i \not\rightsquigarrow target$.

5. Correctness properties of our proposed implementation and time analysis.

LEMMA 5.1. *The implementation guarantees a partial matching with respect to the relation \rightsquigarrow .*

Proof. For the sake of contradiction, assume that there is a $t > 0$ and processes i, j, k such that $i \rightsquigarrow j$ and $i \rightsquigarrow k$. This implies that at time t both poller subprocesses cp_j^i and cp_k^i have the corresponding marriage variables set and the channel open. But this is impossible, because the supreme judge subprocess does not allow more than one marriage variable to be on at the same time. \square

A subprocess cp_k^i gets the channel when it executes ESTABLISH-COM _{i} (k).

LEMMA 5.2. *Death of a process does not affect the communication of other processes.*

Proof. Death of process “target” at any time will only cause blocking of only the subprocess cp_{target}^i per neighbour i of target. This does not disrupt the other subprocesses of the neighbours. \square

LEMMA 5.3. *Suppose that i, j start to be mutually willing to communicate at some time and continue to be willing for 5 local rounds. Then all four subprocesses $cp_{j_1}^i, cp_{j_2}^i$ and $cp_{i_1}^j, cp_{i_2}^j$ (with $j_1 \bmod v = j_2 \bmod v = j$ and $i_1 \bmod v = i_2 \bmod v = i$) will arrive in the queues of i and j in 5 local rounds.*

Proof. Note that at each time the slower of i, j will do only one step in the busy waits of procedures ask or respond. The result follows simply by counting the steps to be executed in each of the procedures. \square

Let Δ_{ij} be the current estimation by i of the ratio of steps of i per step of j as provided by the process ep_j^i . As we noted in § 4.2, execution of ep_j^i assures that Δ_{ij} is (within a factor of two) the actual speed ratio of processes i and j . Let ρ_{ij} be the ratio $1/(\omega(\Delta_i)\Delta_{ij})$. In the following we assume that the oracle \mathcal{A} is tame with respect to processes i, j in the time interval T they attempt communication.

Let S_{ij} be the average number of steps that cp_j^i makes before it is selected to attempt communication, measured from the time it enters the queue.

LEMMA 5.4. *Let Δ_{ij} be the most current estimation used in the last competition in which cp_j^i was the winner. Then $S_{ij} \leq 2vc_2\Delta_{ij}$, (where $c_2 = 4v(\alpha\beta + 1)$ as defined previously).*

Proof. The probability that cp_j^i is the winner, each time it participates in the contest, is at least ρ_{ij} by Lemma 4.2. Let x be a random variable which counts the number of selections done before cp_j^i is selected. Then, $\text{prob}\{x = t\} \leq (1 - \rho_{ij})^t$. The mean value of x is

$$\sum_{t=0}^{\infty} t \cdot \text{prob}\{x = t\} \leq \frac{1}{\rho_{ij}}.$$

Each time cp_j^i is not chosen, it waits in the queue Q^i for an average number of steps bounded above by

$$\sum_{k=1}^{2v} c_2\Delta_{ik}\rho_{ik} \leq \frac{2vc_2}{\omega(\Delta_i)}.$$

So

$$S_{ij} \leq \frac{1}{\rho_{ij}} \left(\frac{2vc_2}{\omega(\Delta_i)} \right) = 2vc_2\Delta_{ij}. \quad \square$$

LEMMA 5.5. *The relative position of the time intervals during which the channels of two neighbour processes are open, is a uniform independent random position and is not affected by the oracle \mathcal{A} .*

Proof. By assumption, the oracle sets the speeds of processes at time 0 and cannot affect their independent probabilistic choices. Furthermore, in our algorithms, each subprocess cp_k^i of a process i goes through a random wait of sufficient length before each return to the queue of process i . In particular, the mean length in steps of waiting interval is the mean number of local rounds to attempt communication. As a consequence, random waits are uniformly distributed in this interval. \square

Let process cp_k^i be in the queue Q^i if $Q^i(k) = 1$. Let $\lambda = 2v(1 + v/2\alpha)$.

THEOREM 5.1. *Each subprocess expects to get the channel at least $1/\lambda$ of the time.*

Proof. Let $S_{Q,k}^i$ be the average number of steps cp_k^i attempts to communicate by executing ESTABLISH-COM. Let Δ_{ik} be the estimation used in the last competition

in which the judges selected cp_k^i to be the winning process. By Lemma 5.4 and since

$$\omega(\Delta_i) = \sum_{k=1}^{2v} \frac{1}{\Delta_{ik}} \leq 2v,$$

$$S_{Q,k}^i = \left(\frac{1}{\Delta_{ik}\omega(\Delta_i)} \right) c_2 \Delta_{ik} = \frac{c_2}{\omega(\Delta_i)} \geq \frac{c_2}{2v} = \frac{4(\alpha\beta + 1)}{2} \geq \frac{2\alpha\beta}{v}.$$

For any time interval T , let $T_{Q,k}^i$ be the subinterval of T in which cp_k^i get the channel. Let δ be the mean speed of process i during T . Let $\mu = \text{mean}(\text{length}(T_{Q,k}^i)/\text{length}(T))$. By Lemma 5.5, our algorithm introduces uniform random translations in time. Then

$$\frac{\text{length}(T_{Q,k}^i)}{\text{length}(T)} \delta = S_{Q,k}^i \quad \text{and} \quad \text{length}(T)\delta = \sum_{k \in F} (S_{Q,k}^i + \beta),$$

where F is the set of indices of subprocesses contending in the queue. Clearly, the worst case contention happens when all $2v$ subprocesses are in the queue. Also, the length of T is the sum of time in channel for each process plus the time of competition for each process. In the case where process i is the fastest and all neighbours slow down with the same worst case acceleration α , $S_{Q,k}^i$ is the same for all subprocesses in the queue. Hence,

$$\text{length}(T)\delta \leq \sum_{k=1}^{2v} (S_{Q,k}^i + \beta)$$

implying

$$\mu \geq \frac{S_{Q,k}^i}{\sum_{k=1}^{2v} (S_{Q,k}^i + \beta)} \geq \frac{1}{\sum_{k=1}^{2v} (1 + \beta/S_{Q,k}^i)} \geq \frac{1}{2v(1 + v/2\alpha)} = \frac{1}{\lambda}$$

since $\beta/S_{Q,k}^i \leq v/2\alpha$. \square

Note that the above theorem justifies the use of the estimate $(\Delta_{ik} \cdot \omega(\Delta_i))^{-1}$ as the probability to select the subprocess cp_k^i from the queue.

In the following we assume $1 \leq i, k' \leq v$ and $k = k' + v$. Thus cp_k^i is the asker and $cp_{k'}^{k'}$ is the responder.

LEMMA 5.6. *The probability of instantaneous overlap of open channels of subprocesses cp_k^i and $cp_{k'}^{k'}$ is at least $1/\lambda^2$.*

Proof. By Theorem 5.1 and Lemma 5.5. \square

Let *success in communication* between i and k' be an overlap of open channels of i and k' for at least one step of both processes i, k' . A *phase* of subprocess cp_k^i is a random wait, a handshake with $cp_{k'}^{k'}$ a wait in queue and a communication attempt.

Let $\gamma_{\min} = 1/2\lambda^2$.

THEOREM 5.2. *The probability of success in communication in a phase of subprocess cp_k^i is at least γ_{\min} .*

Proof. When the subprocess cp_k^i opens its channel, the number of steps done from the time of the estimation of Δ_{ik} used in the selection process of the judges, is $\beta = 6 \log(2v)$ and hence, since $\Delta_{ik} \geq 1$, the new speed ratio can be $\Delta_{ik} + \alpha\beta \leq (\alpha\beta + 1)\Delta_{ik}$ in the worst case.

In the case where process i is the fastest and process k' slows down continuously with the maximum acceleration, process i will do more and more steps per step of k' . This case is the worst case, since it gives an upper bound to the number of steps during which cp_k^i has its channel open, in order to guarantee that $cp_{k'}^{k'}$ will make at least 2 steps. In this case, a communication attempt of $c_2\Delta_{ik}$ time slots where $c_2 = 4(\alpha\beta + 1)$ guarantees that $cp_{k'}^{k'}$ will make at least 2 steps during the time cp_k^i has its channel open.

Note the independent random relative position to these steps with respect to cp_k^i 's steps (due to independent random waits in the poller subprocesses' algorithms). Thus, given that there is an overlap, the probability is at least $1/2$ that the length of the overlap is at least 1 step.

Hence, by Lemma 5.6, there is an overlap and its length is at least one step of both processes with probability at least $\frac{1}{2} \cdot 1/\lambda^2$. \square

Note that the above theorem justifies the selection of the constant $c_2 = 4(\alpha\beta + 1)$ in the communicators' algorithm.

Let \mathcal{C} be the class of oracles for which the out-valence of each node of G_t is v for all t . This class of oracles creates the maximum contention and gives the worst relative response time.

Let $q_{ik}(h/\mathcal{A})$ be the probability that it takes exactly h phases for subprocess cp_k^i to communicate with $cp_i^{k'}$.

Let $\gamma_{\max} = 1/(2v)^2$.

LEMMA 5.7. For any oracle \mathcal{A} ,

$$q_{ik}(h/\mathcal{A}) \leq (1 - \gamma_{\min})^{h-1}.$$

For oracles $\mathcal{A} \in \mathcal{C}$,

$$q_{ik}(h/\mathcal{A}) \leq (1 - \gamma_{\min})^{h-1} \gamma_{\max}.$$

Proof. Since in each phase, the probability of successful communication is at least γ_{\min} and at most 1, we get

$$q_{ik}(h/\mathcal{A}) \leq (1 - \gamma_{\min})^{h-1}$$

by an application of Bayes' theorem of conditional probabilities.

For oracles in class \mathcal{C} , by Theorem 5.2 the probability of successful communication in a phase is at least γ_{\min} . Furthermore, this probability is at most γ_{\max} , due to the contention of all $2v$ processes in any communication attempt.

Let E_m be the event " cp_k^i does not succeed in communicating with $cp_i^{k'}$ at phase m ".

Let H_m be the intersection event $\bigcap_{i=1}^m E_i$.

Let \bar{E}_m be the complement of E_m .

Let H_0 be the empty event.

Then for $\mathcal{A} \in \mathcal{C}$

$$q_{ik}(h/\mathcal{A}) = \left(\prod_{m=1}^{k-1} \text{Prob}(E_m/H_{m-1}) \right) \cdot \text{Prob}(\bar{E}_h/H_{k-1}).$$

But $\text{Prob}(E_m/H_{m-1}) \leq 1 - \gamma_{\min}$ for any m and $\text{Prob}(\bar{E}_h/H_{k-1}) \leq \gamma_{\max}$. Hence, $q_{ik}(h/\mathcal{A}) \leq (1 - \gamma_{\min})^{h-1} \gamma_{\max}$. \square

By using the above lemma and known expressions (see Feller [1966]) for the mean and the tail of a geometric we get

LEMMA 5.8. For oracles in \mathcal{C}

$$\text{mean}(h) \leq \frac{\gamma_{\max}}{(\gamma_{\min})^2}.$$

LEMMA 5.9. For oracles in \mathcal{C}

$$\forall \varepsilon, 0 < \varepsilon < 1, \quad \text{Prob}\{h > h_{\max}(\varepsilon)\} \leq \varepsilon$$

where

$$h_{\max}(\varepsilon) = \frac{\log(\gamma_{\min} \varepsilon) - \log \gamma_{\max}}{\log(1 - \gamma_{\min})}.$$

Note that, by Lemma 5.1 and Theorem 5.1 in the worst case relation of speeds of processes i, k , the total length in steps of a phase of subprocesses cp_k^i is the number of local rounds in the random wait plus the number of local rounds up to the end of the communication attempt. This sum is $c_1 = 2(2v+1)c_2$.

This justifies the use of the constant c_1 in our algorithms for the poller subprocesses.

THEOREM 5.3. *For the worst case of any "adverse" oracle \mathcal{A} , the mean number of local rounds to achieve communication is*

$$\leq c_1 \cdot \frac{\gamma_{\max}}{\gamma_{\min}^2} = 4c_1\lambda^2 \left(1 + \frac{v}{2\alpha}\right) = O(v^6\alpha \log v)$$

and the ε -response of the presented implementation of VS-DCS is

$$S(\varepsilon) \leq c_1 h_{\max}(\varepsilon) = O\left(v^5\alpha \log v \log \frac{v}{\varepsilon}\right).$$

Proof. By the previous remark and the fact that

$$\begin{aligned} h_{\max}(\varepsilon) &= \frac{\log(\gamma_{\min} \varepsilon / \gamma_{\max})}{\log(1 - 1/\lambda^2)} \leq \lambda^2 \log\left(\varepsilon^{-1} \left(1 + \frac{v}{2\alpha}\right)\right) \\ &= \left(2v + \frac{v^2}{\alpha}\right)^2 \log\left(\varepsilon^{-1} \left(1 + \frac{v}{2\alpha}\right)\right). \quad \square \end{aligned}$$

6. Conclusion. We have utilized new adaptive techniques to deal with arbitrary speed variability. Since we have assumed global parameters α and v to be constant, by Theorem 5.3 our VS-DCS system has relative real time response. Our restrictions on processors rates are much less than in our previous work described in Reif and Spirakis [1984], [1984a]. Furthermore, the algorithms given in this paper are more modular and simple in design.

REFERENCES

- D. ANGLUIN, *Local and global properties in networks of processors*, 12th Annual Symposium on Theory of Computing, Los Angeles, CA, April 1980, pp. 82-93.
- E. ARJOMANDI, M. FISCHER AND N. LYNCH, *A difference in efficiency between synchronous and asynchronous systems*, 13th Annual Symposium on Theory of Computing, April 1981.
- A. J. BERNSTEIN, *Output guards and nondeterminism in communicating sequential processes*, ACM Trans. Prog. Lang. and Systems, 2 (1980), pp. 234-238.
- J. B. DENNIS AND D. P. MISUNAS, *A preliminary architecture for a basic data-flow processor*, Proc. 2nd Annual Symposium on Computer Architecture, ACM, IEEE, 1974, pp. 126-132.
- W. FELLER, *An Introduction to Probability Theory and Its Applications*, Vol. I, John Wiley, New York, 1966.
- N. FRANCEZ AND RODEH, *A distributed data type implemented by a probabilistic communication scheme*, 21st Annual Symposium on Foundations of Computer Science, Syracuse, NY, Oct. 1980, pp. 373-379.
- S. HART, M. SHARIR AND A. PNUELI, *Termination of probabilistic concurrent programs*, 9th ACM Symposium on Principles of Programming Languages, Albuquerque, NM, January 1982.
- C. A. R. HOARE, *Communicating sequential processes*, Comm. ACM, 21 (1978), pp. 666-677.
- D. LEHMANN AND M. RABIN, *On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers' problem*, 8th ACM Symposium on Principles of Programming Languages, January 1981.
- R. LIPTON AND F. G. SAYWARD, *Response time of parallel programs*, Research Report 108, Dept. Computer Science, Yale Univ., New Haven, CT, June 1977.
- N. A. LYNCH, *Fast allocation of nearby resources in a distributed system*, 12th Annual Symposium on Theory of Computing, Los Angeles, CA, April 1980, pp. 70-81.
- A. MAHJOUR, *Some comments on Ada as a real-time programming language*, to appear.

- M. RABIN, *N-process synchronization by a $4 \log_2 N$ -valued shared variable*, 21st Annual Symposium on Foundations of Computer Science, Syracuse, NY, October, 1980, pp. 407-410.
- , *The choice coordination problem*, Mem. No. UCB/ERL M80/38, Electronics Research Lab., Univ. California, Berkeley, August 1980.
- J. H. REIF AND P. SPIRAKIS, *Unbounded speed variability in distributed communication systems*, 9th ACM Symposium on Principles of Programming Languages, Albuquerque, NM, Jan. 1982. [1982]
- , *Real time synchronization of interprocess communications*, ACM Trans. Prog. Lang. and Systems, 6 (1984), pp. 215-238. [1984]
- , *Real time resource allocation in a distributed system*, 1st ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Ottawa, Canada, August 1982. [1982a]
- , *Probabilistic bidding gives optimal distributed resource allocation*, 11th International Colloquium on Automata, Languages and Programming, Antwerp, Belgium, Springer-Verlag, New York, 1984. [1984a]
- J. SCHWARZ, *Distributed synchronization of communicating sequential processes*, DAI Research Report 56, Univ. Edinburgh, 1980.
- S. TONAG, *Deadlock and livelock-free packet switching networks*, 12th Annual Symposium on Theory of Computing, Los Angeles, CA, April 1980, pp. 82-93.
- L. G. VALIANT, *A scheme for fast parallel communication*, Technical Report, Computer Science Dept., Edinburgh, Scotland, July 1980.

VORONOI DIAGRAM IN THE LAGUERRE GEOMETRY AND ITS APPLICATIONS*

HIROSHI IMAI†, MASAO IRI† AND KAZUO MUROTA†

Abstract. We extend the concept of Voronoi diagram in the ordinary Euclidean geometry for n points to the one in the Laguerre geometry for n circles in the plane, where the distance between a circle and a point is defined by the length of the tangent line, and show that there is an $O(n \log n)$ algorithm for this extended case. The Voronoi diagram in the Laguerre geometry may be applied to solving effectively a number of geometrical problems such as those of determining whether or not a point belongs to the union of n circles, of finding the connected components of n circles, and of finding the contour of the union of n circles. As in the case with ordinary Voronoi diagrams, the algorithms proposed here for those problems are optimal to within a constant factor. Some extensions of the problem and the algorithm from different viewpoints are also suggested.

Key words. Voronoi diagram, computational geometry, Laguerre geometry, computational complexity, divide-and-conquer, Gershgorin's theorem

Introduction. The Voronoi diagram for a set of n points in the Euclidean plane is one of the most interesting and useful subjects in computational geometry. Shamos and Hoey [15] presented an algorithm which constructs the Voronoi diagram in the Euclidean plane in $O(n \log n)$ time by using the divide-and-conquer technique, and showed many useful applications. Since then, various generalizations of the Voronoi diagram have been considered. Hwang [6] and Lee and Wong [10] considered the Voronoi diagrams for a set of n points under the L_1 -metric, and the L_1 - and L_∞ -metrics, respectively, and gave $O(n \log n)$ algorithms to compute them. Lee and Drysdale [9] studied the Voronoi diagrams for a set of n objects such as line segments or circles, where the distance between a point and an object is defined as the least Euclidean distance from the point to any point of the object, and therefore the edges of these Voronoi diagrams are no longer simple straight line segments but may contain fragments of parabolic or hyperbolic curves. They gave an $O(n(\log n)^2)$ algorithm to construct these diagrams, and Kirkpatrick [7] reduced its complexity to $O(n \log n)$.

Here we extend the concept of usual Voronoi diagram in the Euclidean geometry for n points to the one in the Laguerre geometry for n circles in the plane, where the distance from a point to a circle is defined by the length of the tangent line. Then the edges of these extended diagrams are simple straight line segments which are easy to manipulate. We show that there is an $O(n \log n)$ algorithm for this extended case.

In spite of the unusual distance employed here, the Voronoi diagram in the Laguerre geometry can be applied to solving efficiently a number of geometric problems concerning circles. By using this extended Voronoi diagram, the problem of determining whether or not a point belongs to the union of given n circles can be solved in $O(\log n)$ time and $O(n)$ space with $O(n \log n)$ preprocessing. We can also solve the problem of finding the connected components of given n circles in $O(n \log n)$ time, which can be applied to a problem in numerical analysis, namely, estimating the region where the eigenvalues of a given matrix lie [4]. The problem of finding the contour of the union of n circles can also be solved in $O(n \log n)$ time, which can be applied to image processing and computer graphics. As in the case of the problems connected with the

* Received by the editors December 15, 1981, and in final revised form August 20, 1983.

† Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo, Tokyo, Japan 113.

ordinary Voronoi diagram, the methods proposed here are optimal to within a constant factor.

Some further generalizations of the problems and the algorithms from different viewpoints are also suggested.

1. Laguerre geometry. Consider the three-dimensional real vector space \mathbf{R}^3 where the distance $d(P, Q)$ between two points $P = (x_1, y_1, z_1)$ and $Q = (x_2, y_2, z_2)$ is defined by $d^2(P, Q) = (x_1 - x_2)^2 + (y_1 - y_2)^2 - (z_1 - z_2)^2$. In the Laguerre geometry [1], a point (x, y, z) in this space \mathbf{R}^3 is made to correspond to a directed circle in the Euclidean plane with center (x, y) and radius $|z|$, the circle being endowed with the direction of revolution corresponding to the sign of z . Then the distance between two points in \mathbf{R}^3 corresponds to the length of the common tangent of the corresponding two circles. Hereafter we consider the plane with distance so defined. Note here that, so long as the distance $d_L(C_i, P)$ between a circle $C_i = C_i(Q_i; r_i)$ with center $Q_i = (x_i, y_i)$ and radius r_i and a point $P = (x, y)$ is concerned, the direction of the circle has no meaning since the distance $d_L(C_i, P)$ is expressed as

$$(1) \quad d_L^2(C_i, P) = (x - x_i)^2 + (y - y_i)^2 - r_i^2,$$

$d_L(C_i, P)$ being the length of the tangent segment from P to C_i if P is outside of C_i . Note that, according as a point P lies in the interior of, on the periphery of, or in the exterior of circle C_i , $d_L^2(C_i, P)$ is negative, zero, or positive, respectively. The locus of the points equidistant from two circles C_i and C_j is a straight line, called the *radical axis* of C_i and C_j , which is perpendicular to the line connecting the two centers of C_i and C_j . If two circles intersect, their radical axis is the line connecting the two points of intersection. Typical types of radical axes are illustrated in Fig. 1. If the three centers

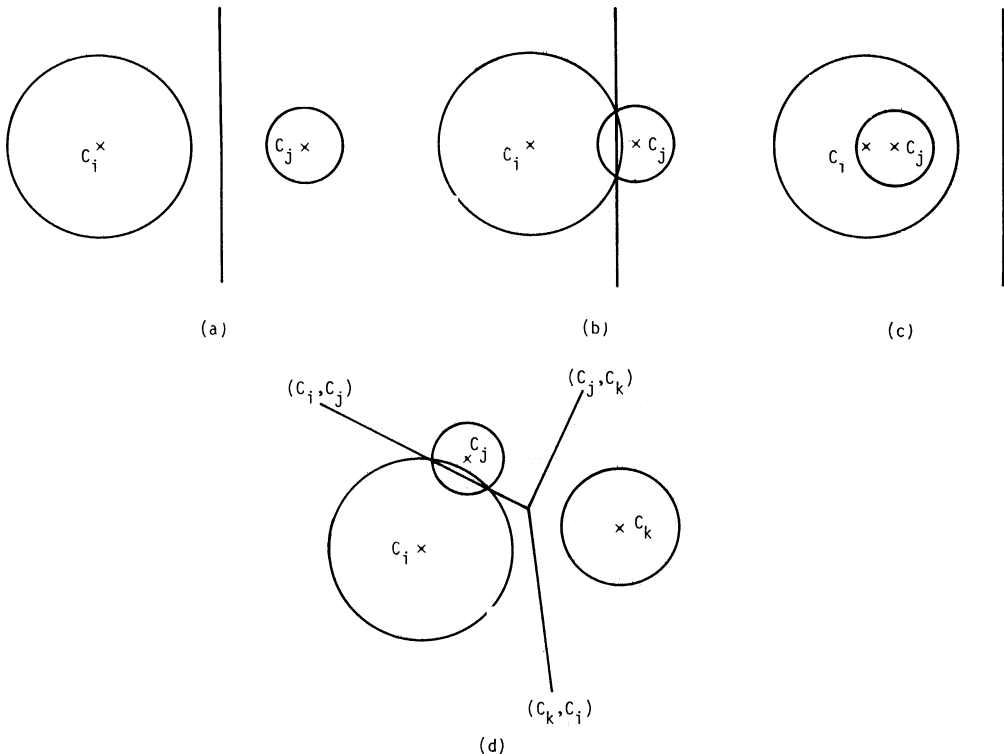


FIG. 1. Radical axes and radical centers.

of three circles C_i, C_j and C_k are not on a line, the three radical axes among C_i, C_j and C_k meet at a point, which is called the *radical center* of C_i, C_j and C_k (see Fig. 1(d)).

2. Definition of the Voronoi diagram in the Laguerre geometry. Suppose n circles $C_i = C_i(Q_i; r_i)$ ($Q_i = (x_i, y_i)$) are given in the plane, where the distance between a circle C_i and a point P is defined by $d_L(C_i, P)$ as in § 1. Then the *Voronoi polygon* $V(C_i)$ for circle C_i is defined by

$$(2) \quad V(C_i) = \bigcap_j \{P \in \mathbf{R}^2 \mid d_L^2(C_i, P) \leq d_L^2(C_j, P)\}.$$

Note that the inequality $d_L^2(C_i, P) \leq d_L^2(C_j, P)$ determines a half-plane so that $V(C_i)$ is convex. However, note also that $V(C_i)$ may be empty and that C_i may not intersect its polygon $V(C_i)$ when circle C_i is contained in the union of the other circles. The Voronoi polygons for n circles C_i ($i = 1, \dots, n$) partition the whole plane, which we shall refer to as the *Voronoi diagram in the Laguerre geometry* (see Fig. 2). A corner of a Voronoi polygon is called a *Voronoi point*, and a boundary edge of the Voronoi polygon is called a *Voronoi edge*. Furthermore, a circle whose corresponding Voronoi polygon is nonempty (empty) is referred to as a *substantial* (*trivial*) *circle*. In Fig. 2, circle C_3 is trivial and all the others are substantial. It is also seen that, in Fig. 2, circle C_2 has no intersection with $V(C_2)$. A circle that intersects the corresponding Voronoi polygon is said to be *proper*, and a circle which is not proper is called *improper*. The following is immediate from the above definitions.

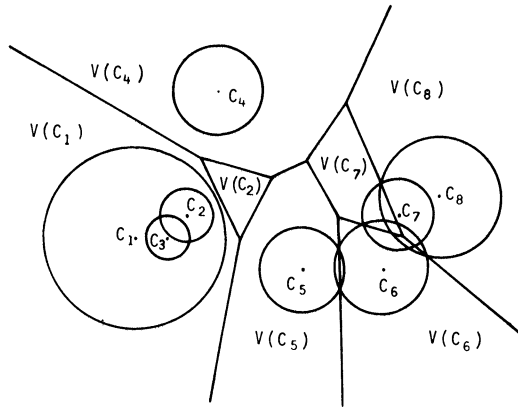


FIG. 2. Voronoi diagram in the Laguerre geometry.

LEMMA 1. (i) A trivial circle is necessarily improper.

(ii) An improper circle is contained in the union of the proper circles.

Obviously, if $r_i = 0$ for all i , the Voronoi diagram in the Laguerre geometry reduces to that in the ordinary Euclidean geometry.

In a Voronoi diagram in the Laguerre geometry, a Voronoi edge is (part of) a radical axis and a Voronoi point is a radical center. Since the diagram is planar, and Euler's formula [5] still holds, we have

LEMMA 2. There are $O(n)$ Voronoi edges and points in the Voronoi diagram in the Laguerre geometry for n circles.

In the case of the Voronoi diagram in the ordinary Euclidean geometry for n points P_i ($i = 1, \dots, n$), the Voronoi polygon $V(P_j)$ is unbounded iff point P_j is on the boundary of the convex hull of the n points P_i , but, for the Voronoi diagram in

the Laguerre geometry for n circles C_i with center Q_i , this statement needs some modification, as in Lemma 3 below. In Fig. 3, the center Q_2 of C_2 lies on the boundary of the convex hull of the centers, but $V(C_2)$ is empty.

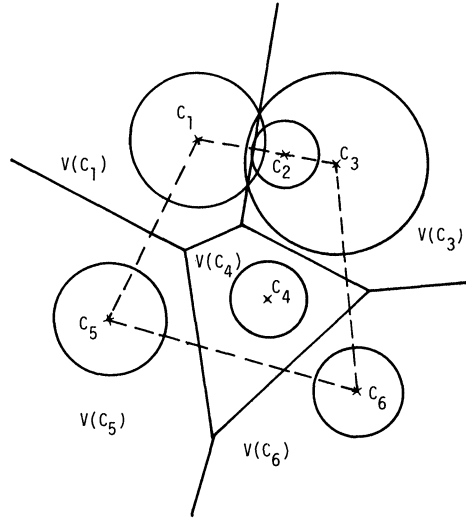


FIG. 3. Relations between the convex hull and Voronoi polygons.

LEMMA 3. *In the Voronoi diagram in the Laguerre geometry, the Voronoi polygon $V(C_i)$ is nonempty and unbounded if the center Q_i of the circle C_i is at a corner of the convex hull of the centers Q_1, \dots, Q_n . Furthermore, if the center Q_j of a circle C_j is on the boundary of this convex hull but not at a corner, its Voronoi polygon $V(C_j)$ is either unbounded or empty. If the center Q_k of a circle C_k is not on the boundary of this convex hull, its Voronoi polygon $V(C_k)$ is either bounded or empty.*

Proof. Consider the Voronoi diagram in the Laguerre geometry for n circles $C_i(Q_i; r_i)$ ($Q_i = (x_i, y_i)$; $i = 1, \dots, n$), where we can assume $y_i \neq y_j$ ($i \neq j$) without loss of generality. First recall (cf. (1), (2)) that a point $P = (x, y)$ belongs to $V(C_1)$ iff

$$d_L^2(C_1, P) \leq d_L^2(C_i, P), \quad i = 1, \dots, n,$$

i.e.,

$$(3) \quad (x_i - x_1)x + (y_i - y_1)y \leq R_i, \quad i = 1, \dots, n,$$

where

$$R_i = (x_i^2 + y_i^2 - r_i^2 - x_1^2 - y_1^2 + r_1^2)/2.$$

Next, note that the center Q_1 of C_1 lies on the boundary (including the corners) of the convex hull of $\{Q_i | i = 1, \dots, n\}$ iff

$$(4) \quad \exists(\alpha, \beta) (\neq (0, 0)): \quad \alpha(x_i - x_1) + \beta(y_i - y_1) \leq 0, \quad i = 1, \dots, n,$$

since all the centers Q_i ($i = 2, \dots, n$) lie on one side with respect to a line passing through (x_1, y_1) .

Suppose that $V(C_1) \neq \emptyset$ and $(x_0, y_0) \in V(C_1)$. Then, $V(C_1)$ ($\neq \emptyset$) is unbounded iff a half line starting from (x_0, y_0) is contained in $V(C_1)$, i.e.,

$$\exists(\alpha, \beta) (\neq (0, 0)), \forall M (> 0): \quad (x, y) = (x_0 + M\alpha, y_0 + M\beta) \text{ satisfies (3),}$$

which is easily seen to be equivalent to (4) above, so that $V(C_1) (\neq \emptyset)$ is unbounded iff the center Q_1 of C_1 lies on the boundary of the convex hull.

When the center Q_1 lies at a corner of the convex hull, there exist two distinct pairs of (α, β) , say, (α_1, β_1) and (α_2, β_2) such that (4) holds and that the vectors $(x_i - x_1, y_i - y_1)$ ($i = 2, \dots, n$) can be represented as linear combinations of (α_1, β_1) and (α_2, β_2) with nonpositive coefficients one of which is strictly negative. The assertion that $V(C_1) \neq \emptyset$ easily follows from the fact that (3) holds for $(x, y) = (M\alpha, M\beta)$ with a sufficiently large $M(>0)$, where $(\alpha, \beta) = (\alpha_1 + \alpha_2, \beta_1 + \beta_2)$. \square

3. Construction of the Voronoi diagram in the Laguerre geometry. We shall show that the Voronoi diagram in the Laguerre geometry can be constructed in $O(n \log n)$ time. The algorithm is based on the divide-and-conquer technique, which is very much like the one proposed initially by Shamos and Hoey [15] in constructing the Voronoi diagram in the ordinary Euclidean geometry for n points, but which is different in some essential points. We shall briefly review Shamos and Hoey's algorithm first, and then explain the difference.

Shamos and Hoey's algorithm works as follows. For a given set $S = \{P_1, P_2, \dots, P_n\}$ of n distinct points, we sort them lexicographically by their (x, y) -coordinates with the x -coordinate as the first key. Then, renumbering the indices of the points in that order, we divide S into two subsets $L = \{P_1, P_2, \dots, P_{\lfloor n/2 \rfloor}\}$ and $R = \{P_{\lfloor n/2 \rfloor + 1}, \dots, P_n\}$. We recursively construct the Voronoi diagrams $V(L)$ and $V(R)$ for points in L and R , respectively, and merge $V(L)$ and $V(R)$. If we can merge $V(L)$ and $V(R)$ in $O(n)$ time, the Voronoi diagram $V(S)$ can be computed in $O(n \log n)$ time.

By virtue of the manner of partitioning S into L and R , there exists a unique unicursal polygonal line, called the *dividing (polygonal) line*, such that every point to the left [right] of it is closer to some point in L [R] than to any point in R [L]. Once this dividing line is found, we can obtain the diagram $V(S)$ in $O(n)$ time simply by discarding that part of Voronoi edges in $V(L)$ and $V(R)$ which lies, respectively, to the right and to the left of the dividing line.

Hence, the main problem in merging $V(L)$ and $V(R)$ is to find the dividing polygonal line in $O(n)$ time, which is actually possible by virtue of the following properties (Lemmas 4 and 5) of the dividing line.

LEMMA 4. *The dividing line is composed of two rays extending to infinity and some finite line segments. Each element (a ray or a segment) is contained in the intersection of $V(P_i)$ in $V(L)$ and $V(P_j)$ in $V(R)$ for some pair of $P_i \in L$ and $P_j \in R$ and is the perpendicular bisector of P_i and P_j .* \square

LEMMA 5. *Each of the two rays is the perpendicular bisector of a pair of consecutive points on the boundary of $CH(S)$, the convex hull of points of S , such that one is in L and the other in R .* \square

Lemma 4 implies that, given a ray, we can find the dividing line in $O(n)$ time by tracing it from the ray to the other by means of a special scanning scheme, i.e., by the clockwise and counterclockwise scanning scheme [9]. Lemma 5, on the other hand, enables us to find a ray in $O(n)$ time from $CH(S)$, which, in turn, can be found in $O(n)$ time from $CH(L)$ and $CH(R)$ [14], [15].

Most of the above ideas for the Euclidean Voronoi diagram, with suitable modifications, can be carried over to obtain an efficient algorithm for constructing the Voronoi diagram in the Laguerre geometry for n circles $C_i(Q_i; r_i)$ as follows.

The first problem is how to partition the set of given circles into two subsets. We partition the set S of n circles C_i into two sets L and R with respect to the coordinates of the centers Q_i of C_i . That is, we sort centers Q_i ($i = 1, \dots, n$) lexicographically by

their (x, y) -coordinates with the x -coordinate as the first key and divide them into two subsets. Then, the locus of points equidistant (in the Laguerre geometry) from L and R , which we call the dividing line (see Fig. 4), enjoys the same property as in the Euclidean case, as stated below.

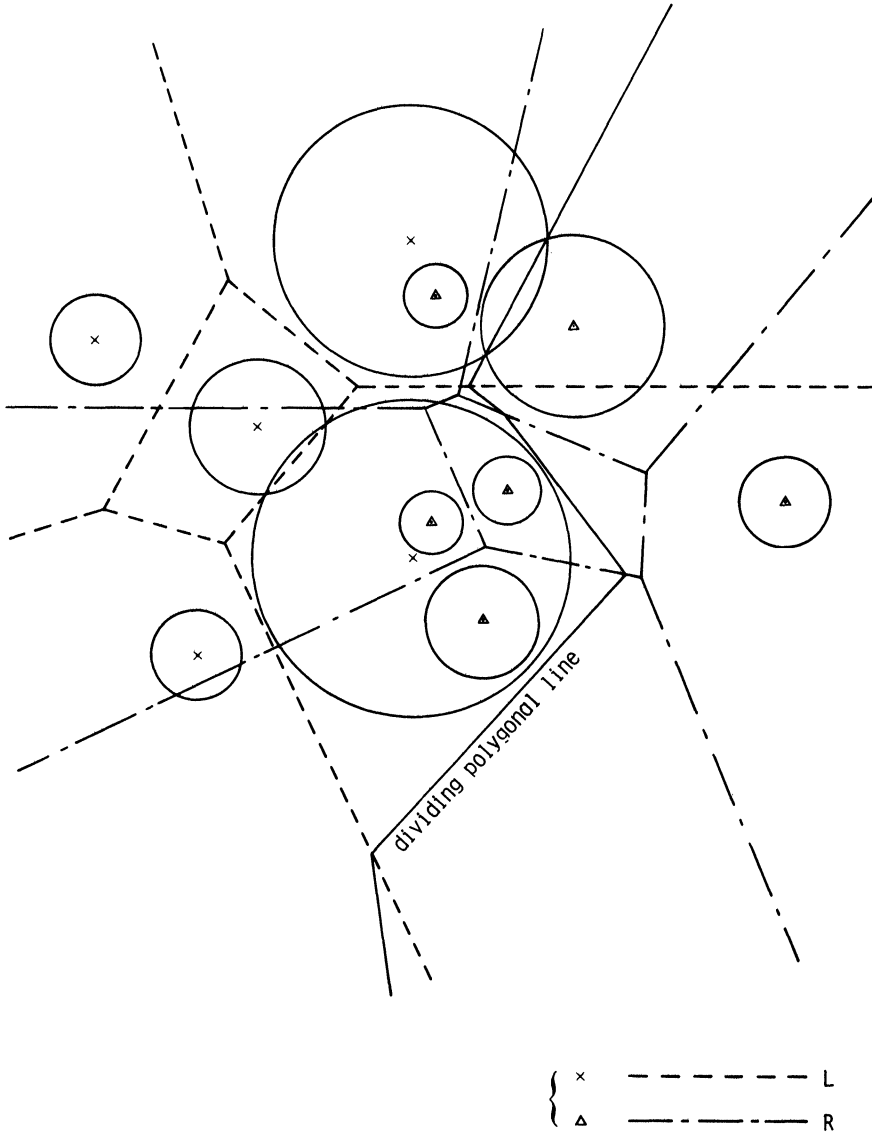


FIG. 4. Merging the Voronoi diagrams in the Laguerre geometry.

LEMMA 6. *The dividing polygonal line is unicursal, consisting of two rays and several finite line segments. Every point to the left [right] of this polygonal line is closer (in the sense of the Laguerre geometry) to some circle in L [R] than to any circle in R [L].*

Proof. By rotating clockwise the axes, if necessary, by a sufficiently small angle, we can assume that $x_i \neq x_j$ ($i \neq j$). Then there exists no Voronoi edge parallel to the x -axis.

It suffices to prove that, for any t , there exists one and only one intersection point $P = (s, t)$ of the dividing line with the line $y = t$, i.e., the dividing line is monotone and hence unicursal. By the assumption that $x_i \neq x_j$ ($i \neq j$), there exists at least one such point $P = (s, t)$, since the point $(-\infty, t)$ is nearer to L than to R whereas the point $(+\infty, t)$ is nearer to R than to L .

For such a point $P = (s, t)$ let $C_i(Q_i; r_i)$ be the circle in L that is nearest to the point P and $C_j(Q_j; r_j)$ the circle in R that is nearest to P . Since $x_i < x_j$, we see by elementary calculation that, for some $\varepsilon > 0$,

$$(5) \quad (s + \varepsilon, t) \in V(C_j) \text{ and } (s - \varepsilon, t) \in V(C_i).$$

Suppose that there were more than one intersection point, say, $P_1 = (s_1, t)$, $P_2 = (s_2, t)$, \dots , $P_k = (s_k, t)$ ($s_1 < s_2 < \dots < s_k$; $k \geq 2$). It follows from (5) that the points (s, t) with $s = s_1 + \varepsilon$ ($< s_2$) are nearer to R than to L , whereas the points with $s = s_2 - \varepsilon$ ($> s_1$) are nearer to L than to R . Therefore, there exists one and only one intersection point $P = (s, t)$ of the dividing line with the line $y = t$. The Lemma then follows by the continuity arguments. \square

It should be noted that the property of the above Lemma 6 does not hold for the Voronoi diagram for line segments, i.e., that there may appear an “ L -island” in the R -region and vice versa, which makes the problem quite complicated [9].

The second problem is to trace the dividing line from a given ray to the other ray in linear time. Since a statement similar to Lemma 4 holds for the Voronoi diagram in the Laguerre geometry, we can simply utilize the ordinary clockwise and counter-clockwise scanning scheme by taking advantage of the fact that the Voronoi edges are straight lines.

The last problem is to find a ray in $O(n)$ time. The ray is found just as in the ordinary Voronoi diagram from the convex hull of the centers (cf. Lemma 5), provided that the new hull edge is not degenerate (i.e., not collinear). In the degenerate case, however, the property of Lemma 5, as it stands, does not necessarily hold, and something more is needed. For example, consider the case shown in Fig. 5(i), where one of the new hull edges is degenerate. Let l be the line of the new degenerate hull edge of the convex hull of the centers. Even if Q_4 and Q_5 are the closest pair of centers on l such that $C_4 \in L$ and $C_5 \in R$, the radical axis of C_4 and C_5 does not appear in the Voronoi diagram (Fig. 5(ii)). In place of Lemma 5, we have the following Lemma 7 in the Laguerre geometry.

LEMMA 7. *Consider the line l of the new hull edge (in the degenerate case, edges) of the convex hull of the centers Q_1, \dots, Q_n . Let L_l and R_l be sets of circles in L and R , respectively, with their centers on l . Let $C_{i*} \in L_l \subseteq L$ and $C_{j*} \in R_l \subseteq R$ be two circles which have the corresponding Voronoi edge e^* in the Voronoi diagram $V(L_l \cup R_l)$ in the Laguerre geometry for the subset $L_l \cup R_l$ of circles. Then, e^* , which is the radical axis of C_{i*} and C_{j*} , is a ray of the dividing line in merging $V(L)$ and $V(R)$.*

Proof. From the Lemma 3, it is obvious that the two circles corresponding to a ray of $V(L \cup R)$ have their centers on the boundary of the convex hull of Q_1, \dots, Q_n . Therefore, the edge e^* is the only candidate for the ray of the dividing line. \square

In order to find the ray of the dividing line in $O(n)$ time, we find the Voronoi edge e^* in the diagram $V(L_l \cup R_l)$ for circles in $L_l \cup R_l$ in linear time in the following way. Note that the Voronoi edges of $V(L_l \cup R_l)$ are all parallel.

First, we construct the diagrams $V(L_l)$ and $V(R_l)$ for circles in L_l and in R_l , respectively, from the diagrams $V(L)$ and $V(R)$, which can be done in linear time as follows. Considering a part of the diagram $V(L)$ far from the line l , we see that two circles in L_l share a Voronoi edge in $V(L_l)$ iff they share a Voronoi edge in $V(L)$.

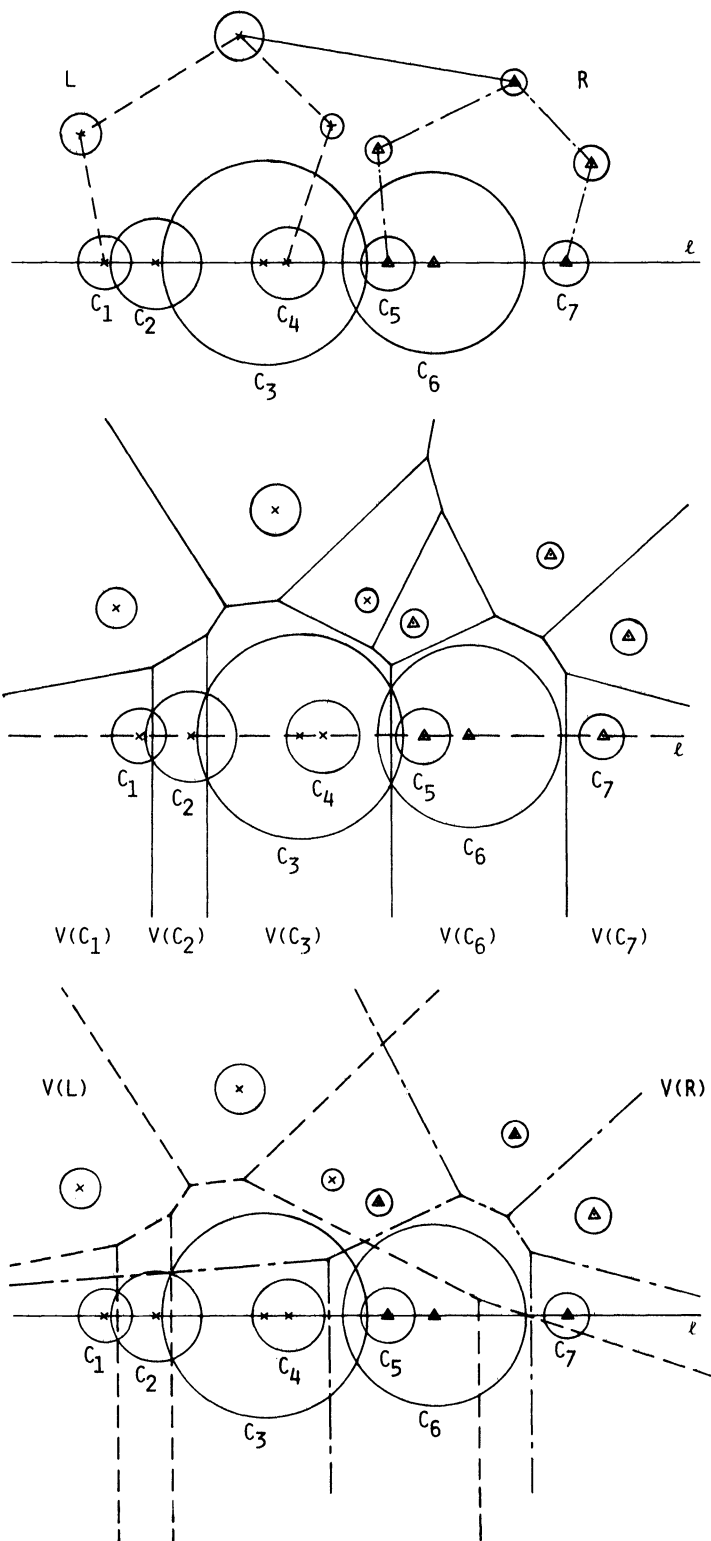


FIG. 5. Finding a ray in a degenerate case. (i) Degenerate new hull edge l ($L_l = \{C_1, C_2, C_3, C_4\}$, $R_l = \{C_5, C_6, C_7\}$). (ii) $V(LUR)$. (iii) $V(L)$ and $V(R)$. (iv) $V(L_l)$ and $V(R_l)$.

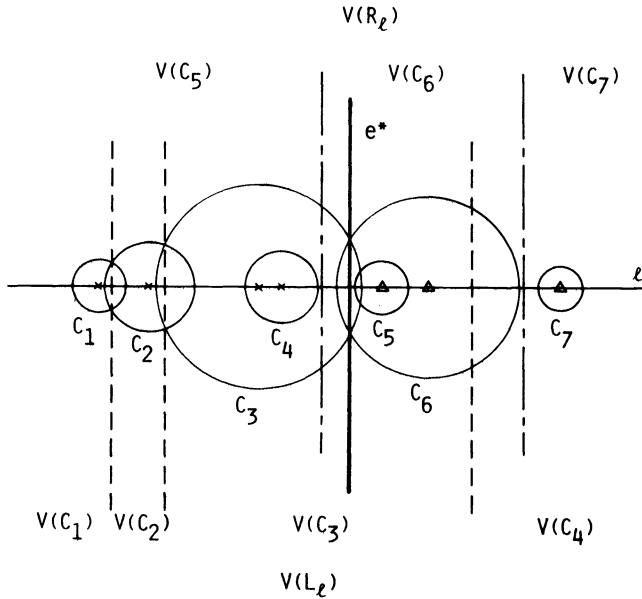


FIG. 5. (cont.)

Hence, the diagram $V(L_l)$ can be constructed simply by picking out the Voronoi edges (rays) of pairs of circles in L_l in the diagram $V(L)$. (In the example of Fig. 5, the $V(L_l)$ shown in Fig. 5(iv) by broken lines can be obtained by extending that part (consisting of parallel lines) of $V(L)$ which is far down to the bottom in Fig. 5(iii).) A similar construction is valid for the diagram $V(R_l)$.

Next, we can find e^* from $V(L_l)$ and $V(R_l)$ in linear time as follows. Since all the Voronoi edges in both diagrams $V(L_l)$ and $V(R_l)$ are perpendicular to l , we can merge the diagrams $V(L_l)$ and $V(R_l)$ to obtain $V(L_l \cup R_l)$ in linear time in a way similar to that in which we merge two sorted lists into a single sorted list. In the merged diagram of $V(L_l)$, and $V(R_l)$, each region between two neighbouring edges is the intersection of two Voronoi regions, one in $V(L_l)$ and the other in $V(R_l)$. For each region of the merged diagram, with which is associated a pair $(C_i \in L_l, C_j \in R_l)$ of circles, we examine whether or not there exists a point equidistant (in the Laguerre geometry) from C_i and C_j within the region; if there exists one, the radical axis of C_i and C_j is the ray e^* . (In the example of Fig. 5(iv), the ray e^* , lying in the intersection of $V(C_3)$ in $V(L_l)$ and $V(C_6)$ in $V(R_l)$, is equidistant from C_3 and C_6 .) Since the number of those regions in that diagram is $O(n)$, we can find e^* , which is the ray of the dividing line, in $O(n)$ time. $V(L_l \cup R_l)$ is ready to obtain from $V(L_l)$, $V(R_l)$ and e^* .

Thus, it has been shown that the Voronoi diagram in the Laguerre geometry for n circles can be constructed in $O(n \log n)$ time.

4. Applications.

Problem 1. Given n circles in the plane, determine whether a given point P is contained in their union or not.

Once we have constructed the Voronoi diagram in the Laguerre geometry for the given n circles $C_i (i = 1, \dots, n)$, we have only to find the Voronoi polygon $V(C_j)$ containing P and check if P lies in C_j . If P is not in C_j , then for any circle C_i , $d_L^2(C_i, P) \cong d_L^2(C_j, P) > 0$, and therefore P is not in any circle. Since we can construct

the Voronoi diagram in the Laguerre geometry in $O(n \log n)$ time, and locate a point in a polygonal subdivision of the plane in $O(\log n)$ time and $O(n)$ storage, using $O(n \log n)$ preprocessing [8], [12], we can solve this problem completely in $O(\log n)$ time and $O(n)$ storage with $O(n \log n)$ preprocessing.

Problem 2. Partition the set of n circles into the connected components. That is, find the connected components of the intersection graph of the n circles, i.e. the graph whose vertices are the circles and which has an edge between two vertices iff the circles corresponding to them intersect in the plane.

This problem arises in numerical analysis when we estimate the eigenvalues of a matrix by means of Gershgorin's theorem [4]. Though the intersection graph can have $O(n^2)$ edges, we can solve this problem in $O(n \log n)$ time as follows with the help of the Voronoi diagram in the Laguerre geometry.

Since an improper circle is contained in the union of the proper circles (Lemma 1) and does not affect the connectedness of the other circles, we first consider only proper circles. For the connectedness of proper circles, we have:

LEMMA 8. *For any pair of proper circles C and C' in the same connected component, there exists a sequence $C = C_1, C_2, \dots, C_k = C'$ of proper circles such that every pair of consecutive circles intersect each other so that they have the corresponding Voronoi edge.*

Proof. Consider the connected component S_I which consists of proper circles and contains C and C' . Since the union of circles in S_I is a connected region and is partitioned into $C_i \cap V(C_i)$ ($C_i \in S_I$) [i.e., $\cup \{C_i \mid C_i \in S_I\} = \cup \{C_i \cap V(C_i) \mid C_i \in S_I\}$], we can take a path within this connected region from a point in $C \cap V(C)$ to a point in $C' \cap V(C')$. Considering a sequence $C = C_1, C_2, \dots, C_k = C'$ of circles in the order in which this path passes through $C_i \cap V(C_i)$ ($C_i \in S_I$), we can see that every pair of consecutive circles in this sequence intersect each other so that they have the corresponding Voronoi edge. \square

We construct a subgraph G of the intersection graph of the n circles which is guaranteed by Lemma 8 to carry the same information as the intersection graph so far as the connected components of the proper circles are concerned. For each pair of proper circles (C_i, C_j) having a common Voronoi edge, we put an edge connecting C_i and C_j in G if the two circles C_i and C_j have a nonempty intersection in the plane. The graph G can be constructed in $O(n)$ time since there exist only $O(n)$ Voronoi edges. Furthermore, the connected components of G can easily be found in $O(n)$ time.

In order to find which components the improper circles belong to, we first make a list of all the improper circles, among which the trivial circles are found in the course of the construction of the diagram and the substantial but improper circles are found by scanning all the Voronoi edges. Next, for each improper circle C_b , we find a proper circle that intersects C_b by locating the center Q_i of C_i in the diagram; if $Q_i \in V(C_b)$, then C_i is a proper circle that contains Q_b , i.e., intersects C_b . The set of centers of improper circles can be located in the diagram in $O(n \log n)$ time by means of the simple algorithm which makes use of a balanced tree [11]. Thus, the total time to find the partition of n circles into the connected components is $O(n \log n)$.

This algorithm is optimal to within a constant factor. In fact, we have

LEMMA 9. *Any algorithm which finds the partition of n circles into the connected components makes at least $\Omega(n \log n)$ comparisons under the linear decision tree model.¹*

¹ A referee has kindly informed the authors that this lemma holds true not only under the linear decision tree model but also under the more precise algebraic computation tree model, based on the recent result by Ben-Or (see M. Ben-Or, *Lower bounds for algebraic computation trees*, Proc. 15th ACM Symposium on Theory of Computing, Boston, 1983, pp. 80-86).

Proof. This follows immediately from the fact that the element-uniqueness problem, i.e., to determine whether given n real numbers are distinct, reduces in linear time to the connected-component problem, where the lower bound of $\Omega(n \log n)$ is known for the element-uniqueness problem under the above model of computation [3]. \square

Problem 3. Find the contour of the union of n given circles in the plane.

This kind of problem is sometimes encountered in image processing and computer graphics. First, we construct the Voronoi diagram in the Laguerre geometry for n circles and then collect that part of the periphery of each circle C_i which lies in the Voronoi polygon $V(C_i)$ for $i = 1, \dots, n$. The validity of this algorithm is obvious. Concerning the number of circular arcs on the contour, we have the following.

LEMMA 10. *The number of circular arcs on the contour is $O(n)$.*

Proof. To distinct pairs of consecutive arcs of the contour, there correspond distinct Voronoi edges (i.e., radical axes), the number of which is $O(n)$. \square

This algorithm is optimal for the contour problem. In fact, we have

LEMMA 11. *The complexity of finding the contour of the n circles in the plane is $\Omega(n \log n)$ under the decision tree model.*

Proof. We show that sorting n real numbers x_1, x_2, \dots, x_n reduces to this problem in $O(n)$ time. First, find $x_* = \min(x_i)$ and $x^* = \max(x_i)$, and let $R = x^* - x_* \geq 0$. Then, consider n circles with centers $(x_i, 0)$ and radii R (see Fig. 6). The contour of the union of these circles consists of circular arcs, and the order of arcs, according to which the contour can be traced unicusally, gives us the sorted list of n numbers. \square

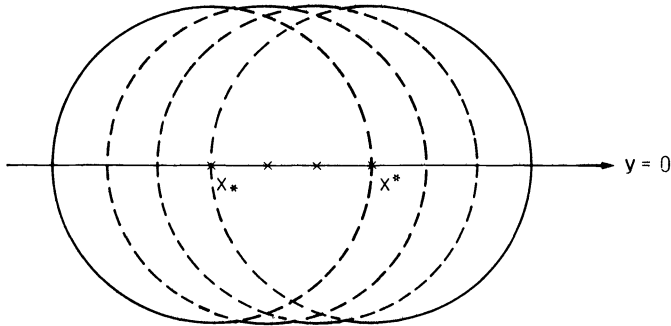


FIG. 6. Reduction of sorting to finding the contour of the union of circles.

5. Discussion. Consider the Voronoi diagram in the Laguerre geometry for n circles $C_i(Q_i; r_i)$ ($Q_i = (x_i, y_i)$; $i = 1, \dots, n$). This diagram will remain invariant if $r_i^2 (i = 1, \dots, n)$ are replaced simultaneously by $r_i^2 - R$ with some constant R ; in other words, this diagram can be regarded as the Voronoi diagram for n points $Q_i = (x_i, y_i)$ in the plane where, with some constant R , a distance $d(Q_i, P)$ between Q_i and a point $P = (x, y)$ is defined by

$$d^2(Q_i, P) = (x - x_i)^2 + (y - y_i)^2 - r_i^2 + R.$$

On the other hand, the two-dimensional section (with $z = 0$) of the Voronoi diagram in the three-dimensional Euclidean space for n points $P_i = (x_i, y_i, z_i)$ ($i = 1, \dots, n$) is a kind of Voronoi diagram for n points $Q_i = (x_i, y_i)$ ($i = 1, \dots, n$), which we will call the *section diagram* (or, the generalized Dirichlet tessellation [13]), with the distance $d(Q_i, P)$ between Q_i and a point $P = (x, y)$ defined by

$$d^2(Q_i, P) = (x - x_i)^2 + (y - y_i)^2 + z_i^2.$$

Hence, by setting $r_i^2 = R - z_i^2$ with sufficiently large constant R , the algorithm we presented here can be applied to the construction in $O(n \log n)$ time of the section with the plane $z = 0$ of the Voronoi diagram for n points in the three-dimensional Euclidean space.

More generally, we can consider the section of the Voronoi diagram in the k -dimensional space with the distance $d_G(P_i, P_j)$ between two points, $P_i = x_i$ and $P_j = x_j \in \mathbf{R}^k$, defined by

$$d_G^2(P_i, P_j) = (x_i - x_j)' G (x_i - x_j),$$

where G is a $k \times k$ symmetric matrix [13], [16]. We can apply the algorithm presented here to such section diagrams even if G is not positive definite (for example, $G = \text{diag}[1, -1, -1]$). Here, it should be noted that the Voronoi diagram in the Laguerre geometry itself is the section with the plane $z = 0$ of the Voronoi diagram for n points $P_i = (x_i, y_i, z_i)$ in three-dimensional space where the square of distance between two points (x_1, y_1, z_1) and (x_2, y_2, z_2) is defined by $(x_1 - x_2)^2 + (y_1 - y_2)^2 - (z_1 - z_2)^2$. Nevertheless, it would be worth while to consider the Voronoi diagram in the Laguerre geometry in connection with the circles since, then, the Voronoi edges and the Voronoi points have the geometrical and physical meanings of radical axes and radical centers, respectively.

Concluding remarks. We have shown that the Voronoi diagram in the Laguerre geometry can be constructed in $O(n \log n)$ time, and is useful for geometric problems concerning circles. Brown [2] considered a technique of inversion which is also useful for geometrical problems for circles. In fact, it can be applied to the problems treated in the present paper. However, our approach is intrinsic in the plane and would be of interest in itself. We have also discussed the relation between the Voronoi diagram in the Laguerre geometry and the two-dimensional section of the Voronoi diagram in the three-dimensional Euclidean space.

Acknowledgment. The authors thank the referees for many helpful comments, without which the paper might have been less readable.

REFERENCES

- [1] W. BLASCHKE, *Vorlesungen über Differentialgeometrie III*, Springer, Berlin, 1929.
- [2] K. Q. BROWN, *Geometric transforms for fast geometric algorithms*, Ph.D. thesis, Computer Science Department, Carnegie-Mellon Univ., Pittsburgh, PA, 1979.
- [3] D. P. DOBKIN AND R. J. LIPTON, *On the complexity of computations under varying sets of primitives*, J. Comput. System Sci., 18 (1979), pp. 86–91.
- [4] D. K. FADDEEV AND V. N. FADDEEVA, *Computational Methods of Linear Algebra*, Fizmatgiz, Moscow, 1960; translated by R. C. Williams, Freeman, San Francisco, 1963.
- [5] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.
- [6] F. K. HWANG, *An $O(n \log n)$ algorithm for rectilinear minimal spanning trees*, J. Assoc. Comput. Mach., 26 (1979), pp. 177–182.
- [7] D. G. KIRKPATRICK, *Efficient computation of continuous skeletons*, Proc. 20th IEEE Symposium on Foundations of Computer Science, San Juan, 1979, pp. 18–27.
- [8] ———, *Optimal search in planar subdivision*, this Journal, 12 (1983), pp. 28–35.
- [9] D. T. LEE AND R. L. DRYSDALE, III, *Generalization of Voronoi diagrams in the plane*, this Journal, 10 (1981), pp. 73–87.
- [10] D. T. LEE AND C. K. WONG, *Voronoi diagrams in $L_1(L_\infty)$ metrics with 2-dimensional storage applications*, this Journal, 9 (1980), pp. 200–211.
- [11] D. T. LEE AND C. C. YANG, *Location of multiple points in a planar subdivision*, Inform. Proc. Letters, 9 (1979), pp. 190–193.

- [12] R. J. LIPTON AND R. E. TARJAN, *Applications of planar separator theorem*, Proc. 18th IEEE Symposium on Foundations of Computer Science, Providence, RI, 1977, pp. 162–170.
- [13] R. E. MILES, *The random division of space*, Suppl. Adv. Appl. Prob. (1972), pp. 243–266.
- [14] F. P. PREPARATA AND S. J. HONG, *Convex hulls of finite sets of points in two or three dimensions*, Comm. ACM, 20 (1977), pp. 87–93.
- [15] M. I. SHAMOS AND D. HOEY, *Closest-point problems*, Proc. 16th IEEE Symposium on Foundations of Computer Science, Berkeley, CA, 1975, pp. 151–162.
- [16] R. SIBSON, *A vector identity for the Dirichlet tessellation*, Math. Proc. Camb. Phil. Soc., 87 (1980), pp. 151–155.

AN EFFICIENT FORMULA FOR LINEAR RECURRENCES*

CHARLES M. FIDUCCIA†

Abstract. The solutions to a scalar, homogeneous, constant-coefficient, linear recurrence are expressible in terms of the powers of a companion matrix. We show how to compute these powers efficiently via polynomial multiplication. The result is a simple expression for the solution, which does not involve the characteristic roots and which is valid for any module over any commutative ring. The formula yields the n th term of the solution to a k th order recurrence with $O(\mu(k) \cdot \log n)$ arithmetic operations, where $\mu(k)$ is the total number of arithmetic operations required to multiply two polynomials of degree $k-1$. Thus if the ring supports a fast Fourier transform, then $O(k \cdot \log k \cdot \log n)$ operations are sufficient to compute the n th term.

Key words. linear recurrences, difference equations, companion matrices, algebraic complexity, polynomial arithmetic, fast algorithms, parallel algorithms

1. Introduction. Let the infinite sequence F_0, F_1, F_2, \dots be a solution to the k th order linear recurrence

$$(1.1) \quad F_{n+k} = c_0 F_n + c_1 F_{n+1} + \dots + c_{k-1} F_{n+k-1}.$$

Given the coefficients, the initial values and an arbitrary natural number n , we wish to compute efficiently the n th term F_n without computing all terms which precede it. Toward this end, let $F[i:j]$ denote the row vector $[F_i, F_{i+1}, \dots, F_j]$. Equation (1.1) can then be written in the vector-matrix form

$$(1.2) \quad F[n+1:n+k] = F[n:n+k-1] \cdot C,$$

where C is the $k \times k$ companion matrix

$$C = \begin{bmatrix} 0 & & & & c_0 \\ 1 & & & & c_1 \\ & 1 & & & c_2 \\ & & \ddots & & \vdots \\ & & & 1 & c_{k-1} \end{bmatrix}.$$

From (1.2) we see that, in terms of the initial values $F[0:k-1]$, we have

$$(1.3) \quad F[n:n+k-1] = F[0:k-1] \cdot C^n.$$

If in (1.3) we let n range over all multiples of k , the entire solution $F[0:\infty]$ can be written as the infinite matrix equation

$$(1.4) \quad F[0:\infty] = F[0:k-1] \cdot C^*,$$

where, by definition, C^* is the $k \times \infty$ matrix

$$(1.5) \quad C^* = [I \quad C^k \quad C^{2k} \quad C^{3k} \quad \dots].$$

Equation (1.4) shows that any solution to (1.1) is a linear combination

$$(1.6) \quad F_0 \beta_1 + \dots + F_{k-1} \beta_k,$$

* Received by the editors February 3, 1983, and in revised form September 19, 1983. A preliminary version of this paper was presented at the Twentieth Annual Allerton Conference on Communication, Control and Computing, Monticello, Illinois, October 1982.

† General Electric R & D Center, Schenectady, New York 12345.

where β_i , henceforth called the i th *basic solution*, is the i th infinite row of C^* . This terminology is suggested by the fact, evident from (1.6), that β_i is the solution to (1.1), when $F[0:k-1]$ is taken as the i th vector of the standard basis.

We also see from (1.3) that the n th term F_n of a solution is the inner product of the vector of its initial values times the first column of C^n . Because the inner product is easy, we will confine our attention to computing the first column of the matrix power C^n .

In any semigroup, the n th power x^n of an element x can be computed with at most $2 \cdot \log n$ semigroup multiplications by the well-known method of repeated squaring

$$x^{2^n} = (x^n)^2, \quad x^{2^{n+1}} = x(x^n)^2, \quad x^1 = x.$$

Urbanek [4] suggests this approach for computing F_n with $O(k^3 \cdot \log n)$ arithmetic operations, by using the classical $O(k^3)$ algorithm to multiply $k \times k$ matrices. Gries and Levin [3] give more efficient recursive formulas for computing the required entries on C^n with $O(k^2 \cdot \log n)$ operations; unfortunately, their method sheds no light on what these entries are.

We show that, for the purpose of computing the n th power of a companion matrix C , indeed any polynomial $p(C)$, matrix multiplication can be replaced with modular polynomial multiplication. This not only reveals what the entries of $p(C)$ are, but also yields a simple expression for the n th term F_n . Unlike existing formulas, such as the well-known one for the n th Fibonacci number, no characteristic roots are required. The new expression immediately explains the result of Gries and Levin, gives a more efficient $O(k^{1.59} \log n)$ algorithm, and yields an $O(k \cdot \log k \cdot \log n)$ algorithm over rings that support an FFT (fast Fourier transform).

2. Arithmetic with a companion matrix. In the sequel, the underlying scalar domain, from which the coefficients of the recurrence are taken, is assumed to be an arbitrary commutative ring K with 1. As usual, $K[X]$ denotes the ring of polynomials over K , while $K[X]/(f(X))$ denotes the ring of polynomials modulo the monic polynomial

$$(2.1) \quad f(X) = X^k - (c_0 + c_1X + \cdots + c_{k-1}X^{k-1}).$$

For concreteness, we view $K[X]/(f(X))$ as the set of all polynomials of degree less than k in which arithmetic is done modulo $f(X)$. Doing arithmetic in $K[X]/(f(X))$ is then equivalent to doing it in $K[X]$ and taking the result modulo $f(X)$. To avoid the repeated use of the operator $\text{mod } f(X)$, the congruence class in $K[X]/(f(X))$ containing X will be denoted by ξ . This means that for any $p(X)$ in $K[X]$, the element $p(\xi)$ in $K[X]/(f(X))$ “is” $p(X) \text{ mod } f(X)$. In particular, $f(\xi) = 0$; so that

$$(2.2) \quad \xi^k = c_0 + c_1\xi + \cdots + c_{k-1}\xi^{k-1}.$$

It is well known that $f(X)$ is the characteristic polynomial of the companion matrix C ; so that $f(C) = 0$. Since the characteristic polynomial of a companion matrix “comes free”, from its last column, one obvious way to compute C^n is first to use the method of repeated squaring to compute $X^n \text{ mod } f(X) = \xi^n = r(\xi)$, say, and then to compute $r(C) = C^n$.

The problem thus reduces to doing fast multiplication in $K[X]/(f(X))$. This can always be done with $O(k^2)$ operations, by doing one multiplication and one division. Modular multiplication can, in fact, be done much faster. Indeed, it has the same complexity as polynomial multiplication, because division is reducible to multiplication [1]. We will henceforth let $\mu(k) = \mu(k, K)$ denote the total number of arithmetic operations required to multiply two polynomials of degree $k-1$ in $K[X]$. We note

[1], [2] that $\mu(k) = O(k^{1.59})$ for any ring K , and that $\mu(k) = O(k \cdot \log k)$ if K supports an FFT or any Vandermonde transform [2]. For our purposes, the most relevant result is the following:

Fact. $X^n \bmod f(X) = \xi^n$ can be computed with $O(\mu(k) \cdot \log n)$ operations.

Returning to the computation of C^n , we see that once $r(\xi)$ has been computed, we can then compute $C^n = r(C)$, with an additional $O(k^3)$ operations, by using Horner's rule and the fact that, owing to its sparseness, C can be multiplied by any $k \times k$ matrix with $O(k^2)$ operations. This yields an $O(\mu(k) \cdot \log n + k^3)$ algorithm for computing C^n . A slight improvement is possible because polynomials $p(C)$ of low degree are sparse; we need not pursue this, however, for we will show that C^n can be computed with only $O(\mu(k) \cdot \log n + k^2)$ operations. In fact, if our only interest is to compute the n th term F_n , we need not compute C^n at all, as we will shortly derive an efficient polynomial expression for any desired column of C^n .

This more efficient method is based on a simple lemma from Fiduccia [2]. Consider the equivalence between $K[X]/(f(X))$ and K^k (obtained by choosing the basis $1, \xi, \xi^2, \dots, \xi^{k-1}$ for the former and the standard basis for the latter) that identifies $p(\xi) = p_0 + p_1\xi + \dots + p_{k-1}\xi^{k-1}$ with its column vector of coefficients $\tilde{p} = [p_0 \ p_1 \ \dots \ p_{k-1}]$. Formally, this equivalence is given by

$$(2.3) \quad p(\xi) = [1 \ \xi \ \xi^2 \ \dots \ \xi^{k-1}] \cdot \tilde{p}.$$

We shall henceforth make no distinction between $p(\xi)$ and \tilde{p} , calling them equal. This gives us the benefits of both matrix notation and polynomial notation.

We are immediately confronted with the question of which polynomial in $K[X]/(f(X))$ is equal to the vector $C \cdot \tilde{p}$ in K^k . The following lemma shows that pre-multiplication by the matrix C is equivalent to multiplication by $X \bmod f(X)$.

LEMMA 2.1 [2]. *For any $p(\xi)$ in $K[X]/(f(X))$, $C \cdot \tilde{p} = \xi p(\xi)$.*

Proof. The proof is by direct computation of $C \cdot \tilde{p}$ and $\xi p(\xi) = Xp(X) \bmod f(X)$. Both computations are trivial, since C is sparse and since the second computation requires only one step of the long division process. Alternatively, we may derive it formally from (2.3), using the identity $f(\xi) = 0$. More elegantly, and a hint of things to come, note that the i th column of the companion matrix C is the element $p(\xi) = \xi^i$; so that $C \cdot \tilde{p} = p_0\xi + \dots + p_{k-1}\xi^k = \xi p(\xi)$. \square

COROLLARY 2.2. *For any $q(X)$ in $K[X]$ and any $p(\xi)$ in $K[X]/(f(X))$, $q(C) \cdot \tilde{p} = q(\xi)p(\xi)$.*

Proof. We use induction on the degree of $q(X)$. Say $q(X) = s(X)X + q_0$; so that $q(C) \cdot \tilde{p} = s(C)(C \cdot \tilde{p}) + q_0\tilde{p}$. Using Lemma 2.1 and induction, this is equal to $s(\xi)(\xi p(\xi)) + q_0p(\xi) = (s(\xi)\xi + q_0)p(\xi) = q(\xi)p(\xi)$. \square

The reader may have noticed that we appear to be working on the wrong problem! For if we were to take $q(X) = X^n$ in Corollary 2.2, we would obtain an efficient method for pre-multiplying by C^n , not post-multiplying by it as required by (1.3). We resolve this problem by using Corollary 2.2 to compute the columns of C^n rather than the product itself.

COROLLARY 2.3. *For any $q(X)$ in $K[X]$, the i th column of $q(C)$ is $q(\xi)\xi^{i-1}$.*

Proof. Choose $p(\xi) = \xi^{i-1}$ in Corollary 2.2; so that \tilde{p} is the i th column of the $k \times k$ identity matrix I , i.e., consider the matrix identity $q(C) = q(C)I$. Since the i th column of I is $p(\xi) = \xi^{i-1}$, the i th column of $q(C)$, on the left, is $q(C) \cdot \tilde{p} = q(\xi)p(\xi) = q(\xi)\xi^{i-1}$. \square

PROPOSITION 2.4. *For any $q(X)$ in $K[X]$, $q(C)$ can be computed with $N(q) + (k-1)(2k-1)$ arithmetic operations, where $N(q)$ is the number of operations needed to compute $q(\xi)$. In particular, C^n can be computed with $O(\mu(k) \cdot \log n + k^2)$ operations.*

Proof. By Corollary 2.3, the first column of $q(C)$ is $q(\xi)$; this can be done with $N(q)$ operations by hypothesis. Using Corollary 2.3 again, along with Lemma 2.1, the $(i+1)$ th column of $q(C)$ is C times its i th column. Each of these $k-1$ multiplications by C takes at most k multiplications and $k-1$ additions; hence, all the remaining entries of $q(C)$ can be done with $(k-1)(2k-1) = O(k^2)$ operations. We know that the first column ξ^n of C^n can be computed with $N(q) = O(\mu(k) \cdot \log n)$ operations; so the total number of operations to compute all the entries of C^n is $O(\mu(k) \cdot \log n + k^2)$. \square

3. An expression for F_n . As previously noted, (1.3) shows that F_n is the inner product of the vector $F[0:k-1]$ of initial values times the first column of C^n . By Corollary 2.3, this column is $X^n \bmod f(X) = \xi^n$. Thus if $[\gamma_0, \dots, \gamma_{k-1}]$, say, is the coefficient vector of ξ^n , then

$$(3.1) \quad F_n = \gamma_0 F_0 + \dots + \gamma_{k-1} F_{k-1}.$$

Using the inner product operator $\langle \cdot, \cdot \rangle$ we obtain the simple expression:

THEOREM 3.1.

$$F_n = \langle [F_0 \ \dots \ F_{k-1}], \xi^n \rangle = \langle [F_0 \ \dots \ F_{k-1}], X^n \bmod f(X) \rangle.$$

Since the inner product can be computed with $2k-1 = O(\mu(k))$ operations, we obtain, as a corollary, our main complexity result:

PROPOSITION 3.2. *The n th term F_n of a k th order linear recurrence can be computed with $O(\mu(k) \cdot \log n)$ arithmetic operations, where $\mu(k) = \mu(k, K)$ is the total number of operations required to multiply two polynomials of degree $k-1$ in $K[X]$.*

The bulk of the work is the computation of $\xi^n = X^n \bmod f(X)$. We may of course use any method at our disposal for this computation, making use of any special knowledge about the polynomial $f(X)$ and the base ring K . It is well-known that solutions exist based on the roots of $f(X)$; those solutions are, of course, equivalent to the expression given by Theorem 3.1 after an appropriate change of basis for $K[X]/(f(X))$ based on the factorization properties of $f(X)$ over K or its extensions.

Equivalence, however, has nothing to do with complexity. Consider for example the case when K is a field and $f(X)$ is irreducible; so that $K[X]/(f(X))$ is a field of degree k over K . The classical expression for F_n is a linear combination of the n th power of each of the k roots of $f(X)$. Since these roots are in $K[X]/(f(X))$, the n th power of each root can be computed with $O(\mu(k) \cdot \log n)$ operations; however, as this must be done for each of the k roots, the total cost will be $O(k \cdot \mu(k) \cdot \log n)$. So, even if we ignore the cost of finding the roots of $f(X)$, the classical method is k times less efficient than our solution; this is because we compute the n th power of only one root of $f(X)$ —the “symbolic” root ξ .

4. The universal solution. Let us delve further into other possible solutions to (1.1) by noting that its right-hand side is simply a K -linear combination valid over any K -module M (essentially any set closed under linear combinations). Thus, given initial values F_0, F_1, \dots, F_{k-1} in M , the recurrence will generate a solution F_0, F_1, F_2, \dots in M . This suggests that Theorem 3.1 has a more general abstract form:

THEOREM 4.1. *For any K -linear mapping L , from $K[X]/(f(X))$ into any K -module M , the infinite sequence $L(1), L(\xi), L(\xi^2), L(\xi^3), \dots$ is a solution to (1.1) in M . All solutions to (1.1) in all K -modules M are of this form.*

Proof. It is clear from (2.2) that the infinite sequence of powers

$$(4.1) \quad 1, \xi, \xi^2, \xi^3, \dots$$

is a solution to (1.1) in $K[X]/(f(X))$, since for all n , we have

$$(4.2) \quad \xi^{n+k} = c_0 \xi^n + c_1 \xi^{n+1} + \dots + c_{k-1} \xi^{n+k-1}.$$

Apply L to both sides and invoke its linearity to establish the first part of Theorem 4.1. To establish that all solutions are as claimed, let F_0, F_1, F_2, \dots be a solution to (1.1) in M . Since $1, \xi, \xi^2, \dots, \xi^{k-1}$ is a basis for $K[X]/(f(X))$, there is always a linear mapping L such that

$$(4.3) \quad L(1) = F_0, \quad L(\xi) = F_1, \dots, L(\xi^{k-1}) = F_{k-1}.$$

For any element $p(\xi) = p_0 + \dots + p_{k-1} \xi^{k-1}$ in $K[X]$, we then have $L(p(\xi)) = p_0 F_0 + \dots + p_{k-1} F_{k-1}$. We establish that $F_n = L(\xi^n)$ for all n by induction. It is true for $n = 0, \dots, k-1$ by definition of L . Assume that $F_i = L(\xi^i)$ for all $i < n+k$. Since, by hypothesis, F_0, F_1, F_2, \dots is a solution to (1.1), we can substitute (4.3) into (1.1) to get

$$(4.4) \quad F_{n+k} = c_0 L(\xi^n) + \dots + c_{k-1} L(\xi^{n+k-1}).$$

The linearity of L and (4.2) then yield $F_{n+k} = L(\xi^{n+k})$ for all n . \square

We again see that the n th term F_n of any solution in any K -module M is

$$(4.5) \quad F_n = L(\xi^n) = \gamma_0 F_0 + \gamma_1 F_1 + \dots + \gamma_{k-1} F_{k-1},$$

where $\xi^n = \gamma_0 + \gamma_1 \xi + \dots + \gamma_{k-1} \xi^{k-1}$, say. This expression is identical to (3.1), except that the F_i are now arbitrarily chosen initial values in M . Note that Theorem 4.1 is an independent abstract reaffirmation of our previous results. No appeal was made to (basis dependent) companion matrices.

Theorem 4.1 shows that the sequence (4.1) in $K(X)/(f(X))$ is, in some sense, the *universal solution* to (1.1), since all other solutions in all other K -modules are linear images of it. It is interesting to pursue the reason for this universality to get an intuitive feeling for it.

For notational simplicity, we confine our attention to solutions in K and consider the infinite matrix equation $F[0:\infty] = F[0:k-1] \cdot C^*$ given by (1.4), where by definition

$$C^* = [I \quad C^k \quad C^{2k} \quad C^{3k} \quad \dots].$$

As previously noted in (1.6), $F[0, \infty]$ is the linear combination

$$F_0 \beta_1 + \dots + F_{k-1} \beta_k,$$

where β_i is the i th infinite row of C^* . This row is the i th basic solution to (1.1) generated by choosing the i th element of the standard basis as the vector $F[0:k-1]$ of initial values. Hence, the rows of C^* are precisely the k basic solutions from which all other solutions can be obtained. The universality of solution (4.1) is now evident from Corollary 2.3; for the powers of ξ are precisely the column vectors of C^* , i.e.,

$$(4.6) \quad C^* = [1 \quad \xi \quad \xi^2 \quad \xi^3 \quad \dots].$$

Consequently, as we generate the next element in the power sequence (4.1), in $K[X]/(f(X))$, we are computing the next column of C^* , and thus *simultaneously* computing the next term of every one of the k basic solutions β_1, \dots, β_k .

As an interesting application of these observations, consider the obvious shift-register implementation of (1.1). This is shown in Fig. 1, and is valid over any K -module M . The boxes represent delay elements (to store the current state), whose internal labels comprise the initial state-vector of the shift-register. In this implementation, the

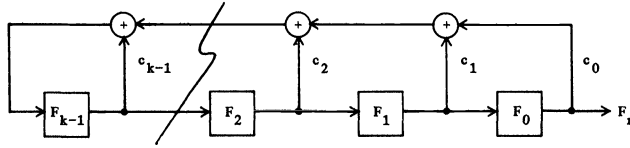


FIG. 1. The obvious implementation of the recurrence (1.1).

initial state consists of the initial values of the recurrence. At each step, the circuit performs k scalar multiplications and $k - 1$ additions in M to compute the next term of the single *specific solution* generated by the given initial values F_0, \dots, F_{k-1} .

Now consider the shift-register shown in Fig. 2, which operates strictly over the base ring K . If its current state is a column k -vector $\tilde{p} = [p_0, \dots, p_{k-1}]$, say, then its next state will clearly be $C \cdot \tilde{p}$. Hence, by Lemma 2.1, the shift-register's next-state function is multiplication by ξ . It follows that if its initial state is any $p(\xi)$ in $K[X]/(f(X))$, its state after the n th iteration will be $\xi^n p(\xi)$. If we start it with the initial state $p(\xi) = [1, 0, \dots, 0] = 1$, in $K[X]/(f(X))$, it will generate the power sequence (4.1). In particular, the infinite sequence produced at the output of the i th delay element will be precisely the basic solution β_i .

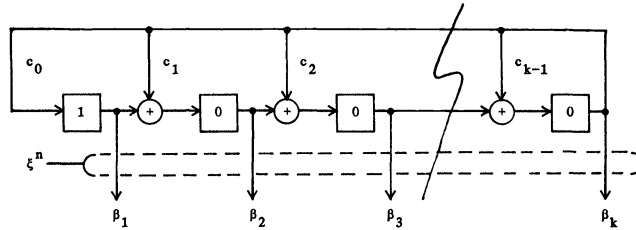


FIG. 2. The generator of the powers $1, \xi, \xi^2, \xi^3, \dots$.

This is interesting from a complexity viewpoint; for this “power generator” simultaneously computes all k basic solutions to (1.1) using only k multiplications and $k - 1$ additions (both in K) per iteration. Moreover, this shift-register is inherently faster than the one in Fig. 1, since that one requires at least $\log k$ time, per iteration, to compute the term F_{n+k} . This speed-up is accomplished at the expense of a high “fan-out” from the last stage of the shift-register in Fig. 2.

Considering the fact that (1.1) uses $2k - 1$ operations to generate each term of a solution, the above observations establish the following rather surprising complexity result:

PROPOSITION 4.2. *The first n terms of all basic solutions β_1, \dots, β_k to the k th order linear recurrence (1.1) can be computed with $(2k - 1)(n - k - 1) = O(k \cdot n)$ arithmetic operations.*

Proof. Follows directly from (1.5) and (1.6), because these $k \cdot n$ terms comprise the first n columns of the matrix C^* . Since its first k columns form the identity matrix, start with column $k + 1$ (it comes free from column k of C). To generate each of the remaining $n - (k + 1)$ columns, pre-multiply the most current column by the matrix C ; this uses at most $2k - 1$ operations per column. \square

Since the first n columns of C^* contain $O(k \cdot n)$ entries, this is (within a factor of 2) an optimal algorithm for computing these terms. Note that each new term of each basic solution β_i is being computed with only one multiplication and one addition per term. By contrast, (1.1) always uses $2k - 1$ operations per term, even for a basic solution.

Proposition 4.2 remains valid over any algebraic structure in which equations (1.1) and (1.5) make sense; for example, the coefficient ring K may be replaced by an arbitrary semiring. An even more general valid setting is obtained by replacing the K -module structure by an (additive) monoid with endomorphisms.

Another interesting property of the second shift-register is that it operates independently of the initial values F_0, \dots, F_{k-1} of the recurrence. These values may be introduced, at any time, by performing a K -linear transformation on the current state-vector ξ^n of the shift-register. One potentially practical benefit of this is that at any stage, having done the bulk of the computation, we can experiment with various initial values, of the recurrence, without having to restart the computation. Indeed, as a consequence of the universality of the power sequence $1, \xi, \xi^2, \xi^3, \dots$ we may “fan-out” the state-vector to as many linear transformations as desired and *simultaneously* compute as many solutions, each with its own initial values, in as many different K -modules, as desired.

5. Conclusions. We have shown that the power sequence $X^n \bmod f(X)$, $n \geq 0$, plays a fundamental role in the efficient solution to a linear recurrence. This was done both concretely, using companion matrices, and abstractly, using K -modules. In the process, we learned how to do efficient arithmetic with a companion matrix and derived a simple generic formula for the solution. The formula does not involve characteristic roots and yields the n th term of a k th order recurrence with $O(k \cdot \log k \cdot \log n)$ arithmetic operations over any ring which supports an FFT. This is a considerable improvement over the $O(k^3 \cdot \log n)$ method and the previously best known $O(k^2 \cdot \log n)$ algorithm.

We have also shown that, unlike the obvious method suggested by (1.1), which uses $2k - 1$ operations to compute each term of a single particular solution, all k basic solutions can be simultaneously computed with no more than two operations per term. This k -fold speed-up is not merely an asymptotic improvement, but an honest-to-goodness gain valid for any k .

This speed-up begs the issue of whether an algorithm exists to generate the first n terms of a single solution with fewer than the obvious $O(k \cdot n)$ operations. The author has recently established that this computation can be done with only $O(n \cdot \log k)$ operations, via the FFT, even for the nonhomogeneous case. Because the methods are substantially different from those presented here, we leave it for a future paper.

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] C. M. FIDUCCIA, *On the algebraic complexity of matrix multiplication*, Ph.D. Thesis, Brown University, Providence, RI, 1973.
- [3] D. GRIES AND G. LEVIN, *Computing Fibonacci numbers (and similarly defined functions) in log time*, Inform. Proc. Lett., 11 (1980), pp. 68–69.
- [4] F. J. URBANEK, *An $O(\log n)$ algorithm for computing the n th element of the solution of a difference equation*, Inform. Proc. Lett., 11 (1980), pp. 66–67.

SPARSE SETS IN $\mathbf{NP-P}$: RELATIVIZATIONS*

STUART A. KURTZ†

Abstract. We construct an oracle relative to which $\mathbf{P} \neq \mathbf{NP}$ and there are no sparse sets in $\mathbf{NP-P}$. The well-known construction of Baker, Gill and Solovay [SIAM J. Comput., 4 (1975), pp. 431-442] gives an oracle relative to which there is a sparse set in $\mathbf{NP-P}$. Together, these results show that simple modifications of conventional proof techniques cannot establish whether or not sparse sets exist in $\mathbf{NP-P}$, even if one assumes $\mathbf{P} \neq \mathbf{NP}$.

Key words. polynomial time (P), nondeterministic polynomial time (NP), sparse sets, relativizations, oracles, forcing

1. Introduction. Relativizations in complexity theory have recently come to play an unexpectedly important role. These results appear to be somewhat problematic to our community: first, because the methods of proof are often highly technical and somewhat foreign; and second, because it is not immediately obvious how these results should be interpreted. The existence of an oracle relative to which some complexity theoretic statement S holds is at best weak evidence for the truth of S (as an unrelativized statement), as frequently another oracle will exist relative to which S fails. Further, the existence of two such oracles is not credible evidence that the statement in question is independent of the usual formal systems of arithmetic, as many provable nonrelativizing statements, both trivial and deep, attest.

The existence of an oracle relative to which S holds does tell us something (beyond the expertise of its creator at oracle constructions). Simply stated, it tells us that any proof of $\neg S$ cannot be based on *relativizing* notions, e.g. *closure conditions* or *uniform enumerability*.

In this paper, we construct an oracle A relative to which there are no sparse sets in $\mathbf{NP-P}$, while guaranteeing that $\mathbf{P}^A \neq \mathbf{NP}^A$. The well-known construction of Baker, Gill and Solovay of an oracle B relative to which $\mathbf{P} \neq \mathbf{NP}$ explicitly yields a sparse (tally) set in $\mathbf{NP}^B - \mathbf{P}^B$. Together, in light of the preceding paragraph, these two oracles demonstrate that no proof based solely on the assumption that $\mathbf{P} \neq \mathbf{NP}$ and basic recursion theoretic properties of \mathbf{P} and \mathbf{NP} can establish whether or not sparse sets are present in $\mathbf{NP-P}$. In particular, the existence of sparse sets in $\mathbf{NP-P}$ cannot be deduced from the hypothesis $\mathbf{P} \neq \mathbf{NP}$ by clever modification of Ladner's density theorem [La]. (The principal technique of Ladner's proof was first used by Borodin, Constable, and Hopcroft [BCH].) Our result shows that such techniques alone cannot suffice.

Hartmanis, Immerman, and Sewelson [HIS] have extended this work to show that the existence of sparse sets in $\mathbf{NP-P}$ is equivalent to the separation of deterministic and nondeterministic exponential time. Thus, the principal theorem of this paper is seen, in retrospect, to be equivalent to a theorem of Book, Wilson, and Xu [BWX] which provides an example of an oracle relative to which exponential deterministic and nondeterministic time collapse, but \mathbf{P} and \mathbf{NP} are separated. Nevertheless, the proof we use is novel, and can be applied in a variety of situations where priority arguments have been used in the past.

We assume familiarity with the paper of Baker, Gill, and Solovay [BGS], and a general acquaintance with Baker and Hartmanis [BH]. Hopcroft and Ullman [HU] serves as a general reference.

* Received by the editors September 21, 1982, and in final form October 17, 1983.

† Department of Mathematics, The University of Chicago, Chicago, Illinois 60637.

In the remainder of this paper, the [BGS] *strategy* will refer to the diagonalization technique employed by Baker, Gill, and Solovay to construct an oracle relative to which $\mathbf{P} \neq \mathbf{NP}$.

2. An oracle.

THEOREM. *There is an oracle A such that $\mathbf{P}^A \neq \mathbf{NP}^A$ and there are no sparse sets in $\mathbf{NP}^A - \mathbf{P}^A$.*

Let L^A denote the language consisting of all strings σ for which there exists a τ of the same length such that $\sigma\tau \in A$. In notation, $L^A = \{\sigma : (\exists \tau)[|\sigma| = |\tau| \ \& \ \sigma\tau \in A]\}$.

To ensure $\mathbf{NP} \neq \mathbf{P}$ we will construct A so that L^A is not in \mathbf{P}^A . (It can be easily seen that L^A is in \mathbf{NP}^A for all oracles A .) As is often the case, a complex goal is more easily realized if it is broken into smaller pieces. Thus, we will ensure that L^A is not in \mathbf{P}^A by satisfying each of the following requirements:

$$R_s: L^A \neq P_s^A$$

where P_s^A is the s th set computable in polynomial time from the oracle A in some fixed enumeration. These requirements have the advantage that the satisfaction of each such requirement can be verified on the basis of only finitely much information about A .

Our other goal is to ensure that there are no sparse sets in $\mathbf{NP}^A - \mathbf{P}^A$. To do this, we will attempt to code the sparse sets in \mathbf{NP}^A into A so that they can be recovered in polynomial time. Let NP_e^A denote the e th \mathbf{NP}^A language in some standard enumeration. Let p_e denote the e th polynomial in some standard enumeration of the polynomials. We can assume without loss of generality that NP_e^A is computable in nondeterministic time p_e ; however, we cannot assume that we have any idea in advance as to how sparse NP_e^A is. Because of this, we will need to make infinitely many coding attempts for every language NP_e^A , one for every polynomial density. Of course, once we see that NP_e^A does not have a certain density (by dint of this density having been exceeded), we need feel no compunction to continue with that particular coding attempt.

In order to describe the coding strategy, we first describe a tripling function with certain technically important properties. Each triple e, i, n will determine an odd integer $k_{e,i,n}$ unique to it. We will use the strings of length $k_{e,i,n}$ to code elements of NP_e^A of length n whenever there are fewer than $p_i(n)$ such elements. We may assume that $k_{e,i,n}$ is computable (in unary representation) in polynomial time from n (also in unary representation) for fixed e and i , and furthermore that $k_{e,i,n}$ is greater than $p_e(n)$, n , and $p_i(n)$. (Such a tripling function can be implemented by a simple modification of Rogers' pairing function [Ro, p. 64]— $\tau(n, m) = \frac{1}{2}(x^2 + 2xy + y^2 + 3x + y)$). We simply define $k_{e,i,n}$ to be $2\tau(\tau(e, i), \tau(\tau(p_e(n), p_i(m)), n)) + 1$.)

Now, if NP_e^A has fewer than $p_i(n)$ members, we will include the string $\sigma 0^{k_{e,i,n}-n}$ in A if and only if σ is in NP_e^A . Notice that because this coding string is of length $k_{e,i,n}$, it can be uniquely parsed. Also notice that the inclusion or exclusion of the coding string depends only on A restricted to strings of strictly lesser length than σ —thus a coding string cannot have been queried by the computation it encodes.

The construction itself is carried out in stages $s \in \omega$. The purpose of the s th stage is to satisfy the s th diagonalization requirement, i.e. to ensure that $L^A \neq P_s^A$.

Each stage is a complex entity unto itself. This complexity is due to possible interactions between the coding and diagonalization strategies. To accomplish the diagonalization, we will want to use essentially the [BGS] strategy. The difficulty occurs

when we attempt to set up a computation $P_s^A(n)$ to diagonalize against. This computation may attempt to query strings not of the form $\sigma\tau$ (τ of the same length as σ). In [BGS] such queries were harmless, conceptually because they had nothing to do with σ 's possible membership in L^A , and technically because any queries made about undetermined strings could be immediately resolved by excluding such strings from A .

In this computation we cannot arbitrarily include or exclude such strings, as they may be intended for encoding computations. Attempts to anticipate the coding may fail, for the computation being encoded may be the very one we want to set up for σ 's membership in L^A —thereby thwarting our attempted diagonalization.

At this point things seem fairly grim. Nevertheless there is a solution. The critical observation is that while we do need to do considerable coding for the coding strategy to work, the amount of coding at each level is only linear, (in that no more than k coding bits are required of length k) whereas the number of possible points of diagonalization is exponential. Thus, while we are not able to choose precisely which string σ will be used for diagonalization purposes, if we are careful, there will be such a string.

A key idea is drawn from forcing in arithmetic. (The notion of forcing was introduced by Cohen [Co] to establish the independence of the continuum hypothesis. It was subsequently modified by Feferman [Fe] and others for use in arithmetic. Jockusch [Jo] is a particularly attractive representative of current work in recursive function theory on forcing.) We will justify our use of this terminology later. We are working in a very simplified setting, and the sentences we force are themselves simple. No formal acquaintance with these notions is presumed.

The construction builds A in levels. At the end of each stage s we will have determined precisely those membership questions about A for strings of length less than some integer m_s . By convention, and to keep notation uniform, m_{-1} is 0.

Each stage is divided into three phases, each with its own strategies and goals. During the first phase, we act only to satisfy coding requirements, while extending A to all strings of length less than some even integer m . It is our intention to arrange that $L^A(\sigma) \neq P_e^A(\sigma)$. During the second phase, we endeavor to extend A to strings of length greater than m and less than or equal to $p_s(m)$. This is the heart of the construction. While we extend, we need to make sure that the coding requirements are satisfied. This is harder than it seems, because we will not have determined A for certain strings which are shorter than those being used for coding. Somewhat remarkably, we can do this, at the cost of determining A for a small number of strings of length m . (Phase I guarantees that “small” is “small enough.”) During the third and final phase, we complete the determination of A for strings of length m . It is during this phase that we are able to apply the [BGS] strategy and guarantee that the diagonalization requirement will be satisfied.

The construction—Stage e .

Phase I (idling). The purpose of the first phase is to gain sufficient room for the following phases. During this phase, no attempt is made to satisfy the diagonalization requirement. Rather, we only worry about satisfying the coding requirement. For this reason, we refer to this as the “idling” phase of this stage. Inductively, we assume that membership questions about A have been decided for all strings of length less than some integer m_{s-1} . This phase consists of substages k , $m_{s-1} \leq k < m$, where m is the least even integer greater than or equal to m_{s-1} such that $2^{m/2} > p_s(m)$.

Substage k . Include a string τ of length k in A if and only if this is necessary to meet the coding strategy, i.e. if $k = k_{e,i,n}$ and there is a σ of length n in NP_e^A such that $\tau = \sigma 0^{k_{e,i,n}-n}$.

Phase II (forcing). This phase consists of substages, one for each k from $m+1$ to $p_s(m)$. During each substage, we completely determine all membership questions about strings of length k in A , while endeavoring to decide as few questions about strings of length m as possible. (Recall, we intend to win our diagonalization requirement at a string of length $m/2$.) During each substage, we will determine membership questions in A for all strings of length k , as well as for at most k^2 strings of length m . The purpose of each substage is to guarantee that the strings of length k perform whatever coding is required, while leaving A as undetermined as possible for strings of length m .

If $k = k_{e,i,n}$, then substage k will consist of two subphases. Readers familiar with forcing will recognize the purpose of the first subphase is to force either the statement “ NP_e^A contains no more than $p_i(n)$ elements of length n ” or its negation, where our forcing conditions are extensions of the determined portion of A to include new strings of length m .

Substage k . If k does not equal $k_{e,i,n}$ for some choice of e , i , and n , we decide that no strings of length k are in A , and proceed to the next substage (or to phase III if k is $p_s(m)$). Otherwise, we proceed by endeavoring to include as many strings of length n in NP_e^A , either until no more can be added, or until NP_e^A contains at least $p_i(n)$ many such strings. To add a new element of length n to NP_e^A , we decide some undecided membership questions about A for strings of length m . It is easily seen that if we can add a string of length n to NP_e^A , we can do it by deciding no more than $p_i(m) < k$ many strings of length m in A . Thus, we see that no more than $p_s^2(m)$ many strings of length m were decided about A during this substage, as we added no more than $p_i(n) (< k \leq p_s(m))$ many strings of length n to NP_e^A each of which required adding no more than $p_e(n) (< k \leq p_s(m))$ many strings of length m to A .

At this point, how we satisfy the coding requirement depends on the outcome of the previous paragraph. If we arranged that NP_e^A contains at least $p_i(n)$ many strings of length n , then the coding requirement is trivially satisfied, as NP_e^A fails to meet the sparseness condition. If, on the other hand, we did not arrange this, then no matter how A is further extended to strings of length m , no new strings of length n will appear in NP_e^A . Thus, we include a string τ of length k in A if and only if $\tau = \sigma 0^{k-n}$ for some σ of length m , such that the portion of A already determined is sufficient to ensure that $\sigma \in NP_e^A$. Because no new strings of length m can appear in NP_e^A as the result of any further action we can take, the coding requirement will be satisfied.

Phase III (diagonalization). As a result of Phase II, we easily see that at most $p_s^3(m) (< 2^{m/2})$ many strings of length m of A were decided. Thus there must be a string σ of length $m/2$ such that no string of the form $\sigma\tau$, τ also of length $m/2$, has been decided about A . At this point we easily utilize the [BGS] diagonalization strategy to arrange that $L^A(\sigma) \neq P_s^A(\sigma)$. At this point, we can set $m_s = p_s(m)$, and continue to stage $s+1$.

We leave to the reader the routine task of verifying that the construction achieves what we purport it to.

3. A brief reflection. We would like to spend a moment reflecting on this construction.

There appears to be considerable confusion as to what constitutes a priority argument, a forcing argument, and a diagonalization argument. Often times, the boundaries between these notions are imprecise, nevertheless there are certain features of an argument that might lead one to call it one type of argument, rather than another.

All of these constructions have the property that they are driven by attempts to satisfy infinitely many requirements. In the cases of interest here, there are only countably many requirements, and the satisfaction of each requirement ultimately depends on only finitely much information about the set being constructed. All of these constructions proceed in countably many stages.

The simplest of these constructions is the diagonalization. In a diagonalization, the s th requirement is *effectively* satisfied during the s th stage. Furthermore, it is possible to see effectively how each requirement is ultimately satisfied.

The next most complex type of argument is forcing. In a typical forcing construction, the s th requirement is satisfied during the s th stage of the construction, but *ineffectively*. In a forcing construction, there is usually no effective way to determine precisely how a requirement was satisfied. Forcing arguments (at least in arithmetic) are finite extension arguments. Because of this, the boundary between forcing and diagonalization is fuzzy indeed. Nevertheless, there is a different “feel” to simple diagonalization requirement as opposed to a forcing construction. Because of the way forcing the negation of a sentence is defined, a universal characteristic of forcing arguments is that one performs an action repetitively (and often ineffectively) until one has either performed it enough times (in which case one has forced a sentence), or until one is no longer able to perform it (in which case one has forced the negation of some sentence).

Priority arguments are the most complex of these types of arguments. In a priority argument, one is attempting to construct an object with certain effectiveness properties. This is typically not a concern in a forcing argument. This presents a real difficulty, in as much as one cannot usually determine effectively how a certain requirement is to be satisfied. (In a typical case, a requirement requires one sort of action if a certain computation converges, and another altogether different sort if it does not. Of course, there is no effective way to ascertain whether or not a given computation will converge.) The solution is to place a priority on the requirements so that for each requirement there are only finitely many other requirements with higher priority. During each stage, one endeavors to satisfy that requirement of highest priority for which decisive action can be taken. In this attempt to take decisive action, it is important that no action taken to satisfy requirements of higher priority be undone. (Usually one has no recourse but to undo work done for the sake of requirements of lesser priority.) Thus, the feel of a priority argument is completely different from either forcing or diagonalization. Because of the dynamic nature of the construction, one cannot argue that any specific requirement has been finally and forevermore dealt with at any point *during* the construction. Only after the construction can one argue that such a point must have arisen.

In classifying the argument of the theorem above, there is some difficulty. Certainly, from a “stage to stage” perspective, this is nothing more than a diagonalization proof—and in this sense nothing out of the ordinary is claimed of it. However, within a stage, the construction is very much a forcing construction. The objection can be raised that what we are doing could be done effectively—but only at the cost of a rather generous notion of what is and what is not computable. Analyzing the timing of this construction is difficult, and certain optimizations are likely possible, but we doubt that this construction can be carried out in time less than 2^{2^2} . This strains our notion of computable. Furthermore, this construction does possess (during the substages of phase II) that “do it until you’re done or can’t do it any more” aspect characteristic of forcing constructions.

Finally, we have found that the distinction between polynomial and exponential is analogous to that between finite and infinite. This analogy has been extremely helpful to us in discovering methods of proof—or exporting known methods from recursion theory. In all respects, our nomenclature is derived by analogy to recursion theoretic arguments, not from literal lemma by lemma and definition by definition translation.

4. Epilogue. The work reported in this paper set off a flurry of activity, both by us and by Juris Hartmanis and his students at Cornell.

The oracle of the previous section came as quite a surprise, as it suggested a “computational universe” quite different from the one we believe we live in. Hartmanis has often described his interest in sparse sets by asking people to consider why problems in **NP** are hard: is it because they contain difficult instances, or are they difficult only in the aggregate. In the universe of this oracle, there are no difficult individual instances.

In an effort to incorporate this oracle into his world view, Hartmanis [HIS] showed that the existence of sparse sets in **NP – P** was equivalent to the separation of deterministic and nondeterministic exponential time. This is a truly remarkable result. In some sense, it can be viewed as having made the current work superfluous, as Wilson [Wi] (see also Book, Wilson and Xu [BWX]) provides an example of an oracle relative to which deterministic and nondeterministic exponential time collapse, while separating **NP** and **P**. Certainly, their proof is simpler than the one presented here.

On the other hand, the method contained herein is powerful, and can be used to prove significantly more than we stated. In particular, at Hartmanis’s urging, we modified the proof to demonstrate the existence of oracles for which “pseudo-sparse” sets of arbitrary sub-exponential density failed to exist in **NP – P**. (The modification is technically a bit tricky, but not particularly deep.) This provides us, via a simple padding observation of Hartmanis, with oracles relative to which **NP**-complete sets have comparatively fast ($2^{n^{1/k}}$) algorithms. Vivian Sewelson has also provided examples of oracles relative to which such fast algorithms exist, without eliminating the sparse sets. Sewelson also constructed a remarkable oracle relative to which deterministic and nondeterministic exponential time collapse, but there remain *co-sparse* sets in **NP – P**.

We would also like to point out that the technique of our theorem can be used to obtain nonpriority constructions of oracles relative to which (most of) the Homer–Maass [HM] results hold. For example, a slight modification of the construction of the preceding section yields an oracle relative to which **P** and **NP** are different, but every infinite **NP** set contains an infinite **P** subset. Many other people have discovered nonpriority proofs of some of the Homer–Maass results, but the original priority constructions remain the most elegant.

Acknowledgments. We would like to thank many people for their role in this paper, and its development. First, we want to thank the people at or visiting Cornell during the AMS Summer Institute in Recursive Function Theory: Juris Hartmanis, Vivian Sewelson, Richard Shore, Peter Fejer, Paul Young, and Deborah Joseph—both for helping us to understand the role of our work in the greater context of computer science, and for valuable aid in its development. We also would like to thank Michael Sipser, who discovered a nonpriority proof of the Homer–Maass results in June ’82 for valuable discussions on this subject.

The question of the role of relativizations in computer science has been the focus of considerable research and discussion recently, as the power and variety of techniques for relativization has increased dramatically. The comments at the beginning of this paper on the role of relativizations are my own, nevertheless, they have been influenced

by discussions with numerous people—especially Alan Selman, Steven Homer, and Juris Hartmanis. While we sincerely doubt that any of these people would completely agree with our assessments, we hope they would not disagree too strenuously!

REFERENCES

- [BCH] A. B. BORODIN, R. L. CONSTABLE AND J. E. HOPCROFT, *Dense and nondense families of complexity classes*, IEEE 10th Annual Symp. on Switching and Automata Theory, 1969, pp. 7–19.
- [BGS] T. BAKER, J. GILL AND R. SOLOVAY, *Relativizations of the P = ?NP question*, SIAM J. Comput., 4 (1975), pp. 431–442.
- [BH] L. BERMAN AND J. HARTMANIS, *On isomorphisms and density of NP and other complete sets*, SIAM J. Comput., 6 (1977), pp. 305–322.
- [BWX] R. V. BOOK, C. B. WILSON AND M. XU, *Relativizing time, space, and time-space*, SIAM J. Comput., 11 (1982), pp. 571–581.
- [Co] P. J. COHEN, *The independence of the continuum hypothesis*, Proc. Natl. Acad. Sci. USA, 50 (1963), pp. 1143–1148; 51 (1964), pp. 105–110.
- [Fe] S. FEFERMAN, *Some applications of the notions of forcing and generic sets*, Fund. Math., 56 (1965), pp. 325–345.
- [HIS] S. HARTMANIS, N. IMMERMANN AND V. SEWELSON, *Sparse sets in NP-P: Exptime versus Nexptime*, Proc. 15th Annual ACM STOC, 1983, pp. 382–391.
- [HM] S. HOMER AND W. MAASS, *Oracle dependent properties of the lattice of NP sets*, Theor. Comp. Sci., 24 (1983), pp. 279–289.
- [HU] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA, 1979.
- [Jo] C. G. JOCKUSCH, *Degrees of generic sets*, in *Recursion Theory: Its Generalizations and Applications*, Drake and Wainer, eds., Cambridge Univ. Press, Cambridge, 1980, pp. 110–139.
- [La] R. LADNER, *On the structure of polynomial time reducibility*, J. Assoc. Comput. Mach., 22 (1975), pp. 155–171.
- [Ro] H. ROGERS, *The Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.
- [Wi] C. WILSON, *Relativization, reducibilities, and the exponential hierarchy*, Technical Report 140/80, Department of Computer Science, Univ. Toronto, 1980.

ON FAULT-TOLERANT NETWORKS FOR SORTING*

ANDREW C. YAO† AND F. FRANCES YAO‡

Abstract. The study of constructing reliable systems from unreliable components goes back to the work of von Neumann, and of Moore and Shannon. The present paper studies the use of redundancy to enhance reliability for sorting and related networks built from unreliable comparators. Two models of fault-tolerant networks are discussed. The first model patterns after the concept of error-correcting codes in information theory, and the other follows the stochastic criterion used by von Neumann and Moore-Shannon. It is shown, for example, that an additional $k(2n-3)$ comparators are sufficient to render a sorting network reliable, provided that no more than k of its comparators may be faulty.

Key words. Batcher's network, comparators, fault-tolerant, Hamming distance, merging, networks, sorting, stochastic

1. Introduction. Consider sorting networks that are built from comparators, where each comparator is a 2 input–2 output device capable of sorting two numbers (Fig. 1). It is of interest to construct sorting networks for n inputs using a minimum number of comparators (see Knuth [5]). It was well-known (see [5]) that at least $\Omega(m \log n)$ comparators are needed, and an upper bound was provided by Batcher's sorting network [2] which used $O(n(\log n)^2)$ comparators. For a long time it remained an open problem to determine the order of magnitude of the true minimum number of comparators needed. Recently, Ajtai, Komlos, and Szemerédi [1] settled this problem by giving an ingenious construction of an n -input sorting network that uses $O(n \log n)$ comparators. In this paper we look into this problem in a new setting. Suppose that some of the comparators are potentially faulty; how can one still design economic networks that will sort properly? We shall assume that, for a faulty comparator, the inputs are directly output without a comparison (Fig. 2).

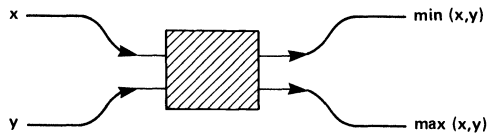


FIG. 1

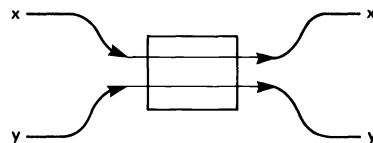


FIG. 2

The study of constructing reliable systems from unreliable components goes back to the work of von Neumann [8], and Moore and Shannon [6]. Currently, the subject of fault-tolerant computing is an active area of research (see, e.g. [7]). The present paper studies the use of redundancy to enhance reliability for a particular problem, similar in spirit to the work on switching networks by Moore and Shannon [6].

* Received by the editors February 26, 1979, and in final revised form September 22, 1983. This research was supported in part by the National Science Foundation under grant MCS-77-05313.

† Computer Science Department, Stanford University, Stanford, California 94305.

‡ Present address, Xerox Palo Alto Research Center, 3333 Coyote Hill Rd, Palo Alto, California 94304.

From the standpoint of analysis of algorithms, our models resemble the problem of sorting with unreliable comparisons. In that direction, a study of binary search with allowance for unreliable comparisons was done in [3].

2. Definitions and notation. An n -network α is a finite sequence of the form $[i_1 : j_1][i_2 : j_2] \cdots [i_r : j_r]$, where each pair $[i_l : j_l]$, with $1 \leq i_l < j_l \leq n$, is called a *comparator*. Any input vector $\mathbf{x} = \langle x_1, x_2, \dots, x_n \rangle \in R^n$ of n real numbers is transformed into an output vector $\mathbf{y} \in R^n$ by the network α , as described below. Associate with a comparator $[i, j]$ the mapping from R^n to R^n defined by

$$\langle x_1, x_2, \dots, x_n \rangle [i : j] = \langle x'_1, x'_2, \dots, x'_n \rangle,$$

where $x'_l = x_l$ if $l \notin \{i, j\}$, and $x'_i = \min \{x_i, x_j\}$, $x'_j = \max \{x_i, x_j\}$. The network α then defines a mapping from R^n into R^n by successively applying the mappings induced by $[i_1 : j_1]$, $[i_2 : j_2]$, \dots , and $[i_r : j_r]$. In other words, for any $\mathbf{x} \in R^n$, the output $\mathbf{y} = \mathbf{x}\alpha$ is defined by

$$\begin{aligned} \mathbf{x}^{(0)} &= \mathbf{x}, \\ \mathbf{x}^{(l)} &= \mathbf{x}^{(l-1)} [i_l : j_l] \quad \text{for } 1 \leq l \leq r, \end{aligned}$$

and

$$\mathbf{x}\alpha = \mathbf{x}^{(r)}.$$

We shall represent an n -network α as shown in Fig. 3, where from left to right each comparator $[i_l : j_l]$ is drawn as a vertical bar connecting the i th and the j th lines. We input $\mathbf{x} = \langle x_1, x_2, \dots, x_n \rangle$ from the left end, with line i carrying x_i . As a comparator $[i_l : j_l]$ is passed, the smaller of the two incoming numbers moves to the upper line i_l , and the larger to the lower line j_l (see Fig. 4 for an example). Thus, between the l th and the $(l+1)$ st comparators, the number carried by line i is the i th component of the vector $\mathbf{x}^{(l)}$. In particular, $(\mathbf{x}\alpha)_i$ is the number found on line i at the right end of α . We call $\mathbf{x}^{(l)}$ the l th state vector of input \mathbf{x} relative to α .

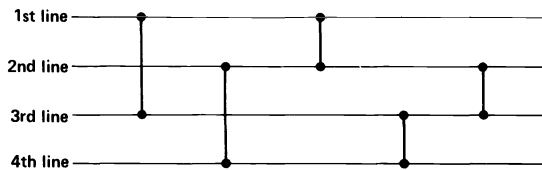


FIG. 3

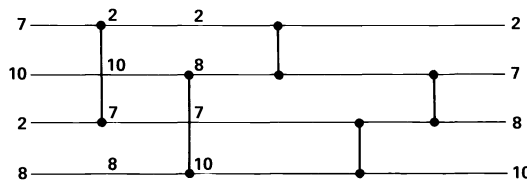


FIG. 4

A vector $\mathbf{x} = \langle x_1, x_2, \dots, x_n \rangle$ is *sorted* if $x_1 \leq x_2 \leq \dots \leq x_n$. A *sorting network* for n elements, or an n -*sorter*, is an n -network α such that, for any input $\mathbf{x} \in R^n$, the output vector $\mathbf{x}\alpha$ is sorted. For instance, the network in Fig. 3 is easily seen to be a 4-sorter. For each n , let $S(n)$ denote the minimum number of comparators required by any n -sorter. It is known [1] [5] that, for large n , $S(n)$ is of order $n \log n$.

Let us now consider the situation when “faulty comparators” may be present. As the effect of having faulty comparators is equivalent to deleting them from the network, an n -sorter may no longer be an n -sorter if there are faulty comparators. Indeed, since the usual emphasis in the design of sorting networks is to avoid redundant comparisons, it is expected that every comparator is crucial in an efficient sorter. It is, therefore, an interesting question whether economic sorting networks would have to look quite different when some fault-tolerant properties are required. We shall discuss two models, with different fault-tolerant criteria, in the following sections. The first model (§ 3) patterns after the concept of error-correcting codes in information theory, and the other (§ 5) follows the criterion used in von Neumann [8] and Moore–Shannon [6].

3. The k -fault model. Let $k \geq 0$ be an integer. We are interested in constructing n -sorters which can sort properly if no more than k of its comparators are faulty. Formally, a k -tolerant n -sorter is an n -sorter α such that, if any k (or fewer) of its comparators are removed, the resulting n -network is still an n -sorter. Let $S_k(n)$ be the minimum number of comparators needed in any k -tolerant n -sorter. Trivially $S_k(n) \leq (k+1)S(n)$, since we can obtain a k -tolerant n -sorter by replacing every comparator in an optimal n -sorter with $k+1$ copies. Our main result in this model is the following theorem, which states that any n -sorter can be made k -tolerant by appending to it a network with $O(kn)$ comparators. The rest of this section is devoted to a proof of Theorem 1.

THEOREM 1. *If α is an n -sorter, then there exists an n -network β with $k(2n-3)$ comparators, such that $\alpha\beta$ is a k -tolerant n -sorter.¹*

COROLLARY. $S_k(n) \leq S(n) + k(2n-3)$.

We need the following “zero-one principle” [5].

LEMMA 1. *Let ξ be an n -network. If $\mathbf{x}\xi$ is sorted for every $\mathbf{x} \in \{0, 1\}^n$, then ξ is an n -sorter.*

Proof. See Knuth [5, § 5.3.4, Thm. Z]. \square

Let θ denote the n -network $[1: 2][2: 3] \cdots [i: i+1] \cdots [n-2: n-1][n-1: n] \times [n-2: n-1] \cdots [i: i+1] \cdots [1: 2]$ (see Fig. 5), and $\beta = \theta^k$ the concatenation of k such networks. Clearly, β consists of $k(2n-3)$ comparators.

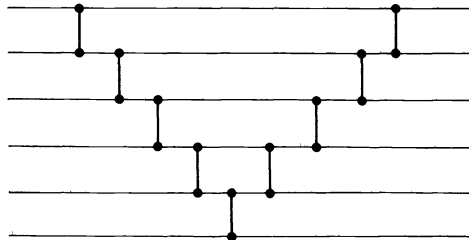


FIG. 5

PROPOSITION 1. *Let ξ be any network obtained from the n -network $\alpha\beta$ by deleting some k' comparators where $k' \leq k$. Then $\mathbf{x}\xi$ is sorted for any $\mathbf{x} \in \{0, 1\}^n$.*

We shall prove Proposition 1 below. Theorem 1 then follows immediately in view of Lemma 1.

Write $\xi = \alpha'\beta'$, where α' and β' are the networks resulting from α and β respectively when some a and b comparators have been removed, with $a + b \leq k$. In

¹ We use $\alpha\beta$ to denote the concatenation of α and β .

the remainder of this section, we will use \mathbf{x} , \mathbf{y} , etc. exclusively for vectors in $\{0, 1\}^n$. For any vector \mathbf{x} , we use \mathbf{x}_s to denote the sorted vector that has the same number of 0's as \mathbf{x} . We first show that the difference between $\mathbf{x}\alpha'$ and \mathbf{x}_s is at most $2a$ in terms of their Hamming distance. (The Hamming distance $D(\mathbf{x}, \mathbf{y})$ of \mathbf{x} and \mathbf{y} , for $\mathbf{x}, \mathbf{y} \in \{0, 1\}^n$, is the number of components where \mathbf{x} and \mathbf{y} differ.) We then show that the network β' , with at least $k - b \geq a$ "good" copies of θ , can reduce that distance to zero.

LEMMA 2. $D(\mathbf{x}[i:j], \mathbf{y}[i:j]) \leq D(\mathbf{x}, \mathbf{y})$ for any comparator $[i:j]$.

Proof. It suffices to show that

$$D(\langle x_i, x_j \rangle[1:2], \langle y_i, y_j \rangle[1:2]) \leq D(\langle x_i, x_j \rangle, \langle y_i, y_j \rangle).$$

This is clearly true if the right-hand side is either 0 or 2. Now, when the right-hand side is 1, that means one of $\{\langle x_i, x_j \rangle, \langle y_i, y_j \rangle\}$ has exactly one 0, and the other has either two or no 0. In either case, we have $D(\mathbf{x}[i:j], \mathbf{y}[i:j]) = 1$. \square

LEMMA 3. $D(\mathbf{x}[i:j], \mathbf{y}) \leq D(\mathbf{x}, \mathbf{y}) + 2$.

Proof. It suffices to prove that

$$D(\langle x_i, x_j \rangle[1:2], \langle y_i, y_j \rangle) \leq D(\langle x_i, x_j \rangle, \langle y_i, y_j \rangle) + 2,$$

which is obviously true. \square

LEMMA 4. Let α' be an n -network obtained from the n -sorter α by deleting some a comparators. Then for any \mathbf{x} ,

$$D(\mathbf{x}\alpha', \mathbf{x}_s) \leq 2a.$$

where \mathbf{x}_s is the sorted version of \mathbf{x} .

Proof. Let $\mathbf{x}^{(l)}$ denote the l th state vector of \mathbf{x} relative to α as defined in § 2, and $\mathbf{y}^{(l)}$ the state vector of \mathbf{x} relative to α' in the corresponding interval. Then, according to Lemmas 2 and 3,

$$D(\mathbf{x}^{(l)}, \mathbf{y}^{(l)}) \leq 2 \times (\text{the number of deleted comparators among the first } l \text{ of } \alpha)$$

by induction on l . Therefore, $D(\mathbf{x}\alpha', \mathbf{x}\alpha) \leq 2a$, and the lemma follows since $\mathbf{x}\alpha = \mathbf{x}_s$. \square

Now we consider the effect of β' on $\mathbf{x}\alpha'$. The network θ is designed so that if a vector \mathbf{z} differs from \mathbf{z}_s only by a transposition, i.e., $\mathbf{z} = \langle 0, 0, \dots, \bar{0}, \dots, 0, 1, 1, \dots, \bar{1}, \dots, 1, \dots, 1 \rangle$ (\bar{d} denotes the complement of d), then θ can carry out the desired swap for \mathbf{z} . In general, θ applied to an arbitrary vector \mathbf{z} which is not sorted reduces the Hamming distance of \mathbf{z} and \mathbf{z}_s by at least 2.

LEMMA 5. $D(\mathbf{z}\theta, \mathbf{z}_s) \leq D(\mathbf{z}, \mathbf{z}_s) - 2$ if $D(\mathbf{z}, \mathbf{z}_s) > 0$.

Proof. Let $\mathbf{z}^{(l)}$ denote the state vectors of \mathbf{z} relative to θ . Suppose there are m 0's in the components of \mathbf{z} ; the following facts can easily be checked.

Fact A. $D(\mathbf{z}^{(l)}, \mathbf{z}_s) = 2 \times (\text{the number of 1's in the first } m \text{ components of } \mathbf{z}^{(l)})$.

Fact B. $D(\mathbf{z}^{(l)}, \mathbf{z}_s)$ is nonincreasing as l increases.

Fact C. $(\mathbf{z}^{(m-1)})_m = 1$.

Proof of Fact C. By the construction of θ , $(\mathbf{z}^{(m-1)})_m = \max\{z_1, z_2, \dots, z_m\}$. Since $D(\mathbf{z}, \mathbf{z}_s) > 0$, z_1, z_2, \dots, z_m cannot be all 0.

We now prove Lemma 5.

Case 1. Suppose $z_{m+1} = 0$. Then $(\mathbf{z}^{(m-1)})_{m+1} = 0$ and $(\mathbf{z}^{(m-1)})_m = 1$ by Fact C. The m th comparator $[m:m+1]$ will swap the two components, and hence $\mathbf{z}^{(m)}$ has one fewer 1's in the first m components than $\mathbf{z}^{(m-1)}$. The lemma then follows from Facts A and B.

Case 2. Suppose $z_{m+1} = 1$. Then Fact C implies that $(\mathbf{z}^{(m)})_m = 1$. It is easy to see that $(\mathbf{z}^{(2n-m-3)})_m = 1$ and $(\mathbf{z}^{(2n-m-3)})_{m+1} = 0$. The $(2n-m-2)$ th comparator $[m:m+1]$ then swaps these two components in $\mathbf{z}^{(2n-m-3)}$, causing $\mathbf{z}^{(2n-m-2)}$ to have one fewer

1's in the first m components than $\mathbf{z}^{(2n-m-3)}$. The lemma again follows from Facts A and B \square

Fact D. For any n -network γ . $D(\mathbf{z}\gamma, \mathbf{z}_s) \leq D(\mathbf{z}, \mathbf{z}_s)$.

LEMMA 6. Assume $D(\mathbf{z}, \mathbf{z}_s) \leq 2a$, and let β' be a network obtained from β by deleting no more than $k - a$ comparators. Then $\mathbf{z}\beta' = \mathbf{z}_s$.

Proof. Write $\beta = \beta^{(1)}\beta^{(2)} \cdots \beta^{(k)}$, where each $\beta^{(i)}$ is a copy of θ . Let $\beta' = \gamma^{(1)}\gamma^{(2)} \cdots \gamma^{(k)}$ such that for some $1 \leq i_1 < i_2 < \cdots < i_a \leq k$, $\gamma^{(i_l)} = \beta^{(i_l)} = \theta$ for all l . If we write $w^{(j)} = \mathbf{z}\gamma^{(1)}\gamma^{(2)} \cdots \gamma^{(j)}$ and $w^{(0)} = \mathbf{z}$, then as j increases, $D(w^{(j)}, \mathbf{z}_s)$ does not increase by Fact D, and in fact decreases by at least 2 when $j = i_l$ and $D(w^{(j)}, \mathbf{z}_s) > 0$ by Lemma 5. Thus $D(w^{(k)}, \mathbf{z}_s) \leq 2a - 2a = 0$. As $w^{(k)} = \mathbf{z}\beta'$, this implies that $\mathbf{z}\beta' = \mathbf{z}_s$. \square

Proposition 1 is an immediate consequence of Lemma 4 and Lemma 6. This completes the proof of Theorem 1.

4. Networks related to sorting. The k -fault model of the previous section extends naturally to comparator networks for other tasks, such as merging and selection.

An (m, n) -merging network α is an $(m+n)$ -network such that, for any $\mathbf{x} \in R^{m+n}$ satisfying $x_1 \leq x_2 \leq \cdots \leq x_m$ and $x_{m+1} \leq x_{m+2} \leq \cdots \leq x_n$, the vector $\mathbf{x}\alpha$ is sorted. Let $M(m, n)$ denote the minimum number of comparators needed by α . An *mf-network* β (minimum-finding) for n inputs is an n -network such that, for any $\mathbf{x} \in R^n$, $(\mathbf{x}\beta)_1 = \min \{x_1, x_2, \dots, x_n\}$. Let $Y(n)$ denote the minimum number of comparators needed by β . It is known that $Y(n) = n - 1$ and

$$\frac{1}{2}n \log_2 (m + 1) \leq M(m, n) \leq (n + m) \lceil \log_2 m \rceil / 2 + m / 2^{\lceil \log_2 m \rceil}$$

(Batcher [2, § 5.3.4], Floyd [5, § 5.3.4, Thm. F], Yao and Yao [9]). The k -fault model for sorting networks can immediately be generalized to these networks. Let $M_k(m, n)$ and $Y_k(n)$ denote the corresponding minimum number of comparators for such networks with k -fault tolerance.

Theorem 1 implies immediately that

$$M_k(m, n) \leq M(m, n) + k(2(m + n) - 3).$$

For $Y_k(n)$, we have the following theorem.

THEOREM 2. $Y_k(n) = (k + 1)(n - 1)$ for $k \geq 0$.

Proof. Let α be any k -tolerant *mf-network* for n inputs. For each j , $1 < j \leq n$, there must be at least $k + 1$ comparators in α of the form $[*, j]$.² Otherwise, when all comparators of the form $[*, j]$ are faulty, the input $\langle x_1, x_2, \dots, x_n \rangle$ with $x_i = 1 - \delta_{ij}$ will not have the correct output under α . Thus, $Y_k(n) \geq (k + 1)(n - 1)$. The reverse inequality follows from the fact that $\alpha = \beta^{k+1}$, where $\beta = [n - 1 : n][n - 2 : n - 1] \cdots [i : i + 1] \cdots [1 : 2]$, is a k -tolerant *mf-network*. \square

5. The stochastic-fault model. In the preceding two sections, we have discussed fault-tolerant networks in a framework that allows at most k faulty comparators. For sorting and merging networks, the addition $O(kn)$ comparators needed is relatively small compared to the basic cost of $n \log n$; for minimum-finding, this extra kn cost is k times the original basic network.

For very large networks, the assumption of no more than k faulty comparators may be too restrictive. It is reasonable to expect that some fixed fraction, say 10^{-4} , of the basic units are faulty. A natural extension of the previous model then leads to the following question. How many comparators are needed to construct an n -sorter

² We use $[*, j]$ to denote any comparator of the form $[i : h]$ where $h = j$.

which remains reliable if any 10^{-4} of the comparators in it are faulty? Unfortunately, reliable networks in this case do not exist when n is large ($n > 10^4 + 1$). Indeed, we assert that if a fraction of $1/(n-1)$ of the comparators may be faulty, then there does not exist any reliable n -sorter in this sense. For any n -sorter α , let $j \in \{2, 3, \dots, n\}$ be such that at most $1/(n-1)$ of the comparators in α are of the form $[*: j]$; then α clearly will not sort all inputs properly if all such comparators $[*: j]$ are faulty (cf. the proof of Theorem 2). In view of this fact, we will define a more relaxed, stochastic model that is very similar to the models studied in von Neumann [8], Moore and Shannon [6].

A stochastic model. Let $0 < \epsilon, \delta < 1$ and n be an integer. An n -network α is an (ϵ, δ) -stochastic n -sorter if the random n -network α' , obtained from α by deleting independently each comparator with any fixed probability $\delta' \leq \delta$, is an n -sorter with probability at least $1 - \epsilon$.

In an (ϵ, δ) -stochastic n -sorter, we shall refer to δ as the *fault probability* (of the comparators), and ϵ as the *failure probability* (of the network). Let $S^{(\epsilon, \delta)}(n)$ be the minimum number of comparators required by any (ϵ, δ) -stochastic n -sorter. Similarly, we can define (ϵ, δ) -stochastic merging networks for $m + n$ inputs, (ϵ, δ) -stochastic mf -networks for n inputs, and the corresponding complexity $M^{(\epsilon, \delta)}(m, n)$, $Y^{(\epsilon, \delta)}(n)$.

A conventional method of achieving reliability is to replace a basic component by several unreliable components which simulate the basic component with high reliability [6], [8]. In our case, connecting in series m comparators, each with δ probability of fault, gives the effect of a single comparator with fault probability δ^m . If α is an n -network with N comparators (none are faulty), the network β obtained from α by replacing each comparator with m comparators in series is called the *canonical m -redundant network of α* . The probability for β to be a network performing the same mapping as α is at least $(1 - \delta^m)^N$, which is greater than $1 - \epsilon$ for large N if $m > (\log(N/\epsilon))/\log(1/\delta)$.

DEFINITION. For given ϵ, δ and network α , the canonical m -redundant network β of α with m chosen just large enough so that β becomes an (ϵ, δ) -stochastic network is called the *canonical (ϵ, δ) -stochastic network simulating α* .

It follows from the preceding discussion that, for fixed ϵ, δ , an arbitrary network α with N comparators may be simulated by its canonical (ϵ, δ) -stochastic network which is of size $O(N \log_2 N)$. It is of interest to study the optimality of this basic strategy for enhancing reliability. As this method exploits redundancy in a primitive way, it is also not surprising that more efficient constructions exist for many problems. We shall bear out these points in the following results. The first result illustrates the optimality of the canonical construction for minimum-finding.

Given $n > 1$ and $m > 0$, let $m_i = \lfloor (m + i - 1)/(n - 1) \rfloor$ for $1 \leq i < n$. The m_i 's form a partition of m into $n - 1$ almost equal parts in that $\sum_i m_i = m$ and $|m_i - m_j| \leq 1$ for all i, j ; they are also the unique set of $n - 1$ numbers satisfying these conditions (see [4, § 1.2.4, Example 38]). Define $g_{\delta, n}(m) = \prod_{1 \leq i < n} (1 - \delta^{m_i})$. It is easy to see that $g_{\delta, n}(m)$ is a nondecreasing function of m for fixed n and $\delta < 1$.

THEOREM 3. Let $0 < \epsilon, \delta < 1$. Then $Y^{(\epsilon, \delta)}(n) = m$ where m is the smallest positive integer satisfying $g_{\delta, n}(m) \geq 1 - \epsilon$.

COROLLARY. For any fixed $0 < \epsilon, \delta < 1$, $Y^{(\epsilon, \delta)}(n) = \Theta(n \log_2 n)$ as $n \rightarrow \infty$.³

Proof. The network $[n - 1 : n]^{m_1} [n - 2 : n - 1]^{m_2} \dots [2 : 3]^{m_{n-2}} [1 : 2]^{m_{n-1}}$ is easily seen to be a valid mf -network with probability at least $g_{\delta, n}(m)$, which is at least $1 - \epsilon$ by the definition of $g_{\delta, n}$. This proves that $Y^{(\epsilon, \delta)}(n) \leq m$.

³ The Θ notation means that there exist constants $a, b > 0$ such that $a(n \log_2 n) \leq Y^{(\epsilon, \delta)}(n) \leq b(n \log_2 n)$.

To prove the reverse inequality, we observe that, in any (ϵ, δ) -stochastic mf -network α for n inputs, we must have

$$(5.1) \quad \prod_{2 \leq j \leq n} (1 - \delta^{l_j}) \geq 1 - \epsilon,$$

where l_j is the number of comparators of the form $[* : j]$.

Fact E. Let $k > 0$ be an integer and $0 < \delta < 1$ a real number. The expression $(1 - \delta^{k_1})(1 - \delta^{k_2})$, where k_1 and k_2 are nonnegative integers satisfying $k_1 + k_2 = k$, is maximized when $|k_1 - k_2| \leq 1$.

Proof of Fact E. Otherwise, assume that the maximum is achieved at (k_1, k_2) with $k_1 > k_2 + 1$. Then

$$(1 - \delta^{k_1})(1 - \delta^{k_2}) > (1 - \delta^{k_1-1})(1 - \delta^{k_2+1}).$$

This implies

$$\delta^{k_1} + \delta^{k_2} < \delta^{k_1-1} + \delta^{k_2+1},$$

or

$$\delta^{k_2}(1 - \delta) < \delta^{k_1-1}(1 - \delta),$$

or

$$k_2 > k_1 - 1,$$

which is a contradiction. \square

In (5.1) let $l = \sum_{2 \leq j \leq n} l_j$. By repeated application of Fact E, the expression $\prod_{2 \leq j \leq n} (1 - \delta^{l_j})$ is maximized when $|l_i - l_j| \leq 1$ for all $2 \leq i, j \leq n$. Therefore

$$g_{\delta,n}(l) \geq \prod_{2 \leq j \leq n} (1 - \delta^{l_j}) \geq 1 - \epsilon.$$

This implies that $l \geq m$. We have proved Theorem 3. \square

To prove the corollary, let $t = \log_{\delta} (1 - (1 - \epsilon)^{1/(n-1)})$, $m' = \lceil t \rceil (n-1)$ and $m'' = (\lceil t \rceil - 1)(n-1)$. It is easy to check that $g_{\delta,n}(m') \geq 1 - \epsilon$ and $g_{\delta,n}(m'') < 1 - \epsilon$. The monotonicity of $g_{\delta,n}$ then implies that $m'' \leq Y^{(\epsilon,\delta)}(n) \leq m'$. It is easy to check that, for fixed $0 < \epsilon, \delta < 1$, we have $t = \Theta(\log n)$ as $n \rightarrow \infty$. This implies that $m' = \Theta(n \log n)$, $m'' = \Theta(n \log n)$, and hence $Y^{(\epsilon,\delta)}(n) = \Theta(n \log n)$.

The canonical (ϵ, δ) -stochastic network may not always be the best solution possible, as the following example shows.

Consider the 3-sorter $\alpha = [2 : 3][1 : 2][2 : 3]$, and its canonical $(\epsilon, \frac{1}{2})$ -stochastic sorter $\beta = [2 : 3]^m [1 : 2]^m [2 : 3]^m$. By definition, the value of m is the smallest positive integer such that $(1 - 1/2^m)^3 > 1 - \epsilon$. It follows that

$$m = \lceil -\log_2 (1 - (1 - \epsilon)^{1/3}) \rceil.$$

For $\epsilon \ll 1$, the total number of comparators in β is then

$$3m \sim 3(\log_2 (1/\epsilon) + \log_2 3) + O(\epsilon).$$

We shall now show that there exist $(\epsilon, \frac{1}{2})$ -stochastic 3-sorters using only $2 \log_2 (1/\epsilon) + O(\ln \ln (3/\epsilon))$ comparators. That is, the canonical construction uses nearly 50% more comparators than is necessary when $\epsilon \rightarrow 0$. The result follows from the next theorem.

THEOREM 4.

$$S^{(\epsilon,\delta)}(3) = 2 \frac{\log_2 (1/\epsilon) + O(\ln \ln (3/\epsilon))}{\log_2 (1/\delta)}$$

Proof. We first compute $Y^{(\epsilon, \delta)}(3)$, which according to Theorem 3 is the smallest m satisfying

$$(1 - \delta^{\lceil m/2 \rceil})(1 - \delta^{\lfloor m/2 \rfloor}) \geq 1 - \epsilon.$$

Writing $m' = \lceil m/2 \rceil$, we obtain

$$1 - \delta^{m'} \geq (1 - \epsilon)^{1/2} = 1 - \frac{1}{2}\epsilon + O(\epsilon^2).$$

This leads to

$$m \geq 2m' - 2 \geq 2 \frac{\log_2(1/\epsilon) + O(1)}{\log_2(1/\delta)}.$$

As $S^{(\epsilon, \delta)}(3) \geq Y^{(\epsilon, \delta)}(3)$, we have proved that

$$S^{(\epsilon, \delta)}(3) \geq 2 \frac{\log_2(1/\epsilon) + O(1)}{\log_2(1/\delta)}.$$

To prove the reverse inequality, we construct a 3-sorter $\alpha_l = [2:3]([1:2][2:3])^l$ (Fig. 6). We shall prove that, for some constant c , the network α_l with $l = (\log_2(1/\epsilon) + c \ln \ln(3/\epsilon))/\log_2(1/\delta)$ is an (ϵ, δ) -stochastic 3-sorter. This then proves the theorem.

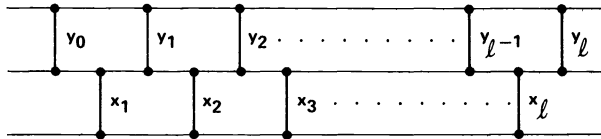


FIG. 6

Writing x for $[2:3]$ and y for $[1:2]$, we can denote α_l by the string $\alpha_l = yxyxy \cdots xy$. For added clarity, we also use the subscripted notation $\alpha_l = y_0x_1y_1x_2y_2 \cdots x_ly_l$ where x_i and y_i refer to the i th $[2:3]$ and $(1:2)$ comparators, respectively. It is easy to see that, when comparators are deleted, the resulting network α'_l fails to be a valid 3-sorter if and only if α'_l does not contain a substring which belongs to yx^+y or xy^+x , i.e., $\alpha'_l \in y^+x^* \cup x^*y^+ \cup x^*$. Thus the probability p_l that α'_l is less than $p_1 + p_2 + p_3$ where

1) $\alpha'_l \in y^+x^*$ with probability

$$(5.2) \quad p_1 \leq \sum_{1 \leq k \leq l+1} k \binom{l+1}{k} (1-\delta)^k \delta^{2l+1-k}$$

since we must have $\alpha'_l = y_{i_1}y_{i_2} \cdots y_{i_j}x_{i_{j+1}}x_{i_{j+2}} \cdots x_{i_k}$ where $1 \leq k \leq l+1$,

$$1 \leq j \leq k, \text{ and } 0 \leq i_1 < \cdots < i_j < i_{j+1} < \cdots < i_k \leq l.$$

After simplifications, (5.2) becomes

$$\begin{aligned} p_1 &\leq (1-\delta) \cdot (l+1) \cdot \delta^l \sum_{1 \leq k \leq l+1} \binom{l}{k-1} (1-\delta)^{k-1} \delta^{l-(k-1)} \\ &= (1-\delta) \cdot (l+1) \cdot \delta^l. \end{aligned}$$

2) $\alpha'_l \in x^*y^+$ with probability $p_2 = p_1 \leq (1-\delta) \cdot (l+1) \cdot \delta^l$, since network α_l is symmetric with respect to left-right reversal.

3) $\alpha'_l \in x^*$ with probability

$$\begin{aligned} p_3 &\leq \sum_{0 \leq k \leq l} \binom{l}{k} (1-\delta)^k \delta^{2l+1-k} \\ &= \delta^{l+1} \sum_{0 \leq k \leq l} \binom{l}{k} (1-\delta)^k \delta^{l-k} = \delta^{l+1}. \end{aligned}$$

Therefore $p_l = p_1 + p_2 + p_3 \leq 3(l+1)\delta^l$. It can be verified that by choosing

$$l = \left\lceil \frac{\ln(c/\varepsilon) + 2(\ln \ln(c/\varepsilon))}{\ln(1/\delta)} \right\rceil, \quad \text{where } \ln c \geq 3,$$

we will have $p_l \leq 3(l+1)\delta^l \leq \varepsilon$. This proves the theorem. \square

6. Concluding remarks. We have studied efficient ways to achieve fault-tolerant ability in some particular problems. The canonical redundancy method sometimes yields economic networks (as for minimum-finding in both models), but not always (it works poorly for sorting in both models). It would be of great interest to find other general principles besides the canonical method.

Some related open problems:

1. For fixed ε, δ , we know that $c_1 n \log n \leq M^{(\varepsilon, \delta)}(n) \leq c_2 n (\log n)^2$, and the same bounds hold for $S^{(\varepsilon, \delta)}(n)$. Question: Determine the order of $M^{(\varepsilon, \delta)}(n)$ and $S^{(\varepsilon, \delta)}(n)$. We conjecture that these functions grow faster than $O(n \log n)$, as the intuitively much simpler minimum-finding network has complexity $Y^{(\varepsilon, \delta)}(n) = \Theta(n \log n)$ already.

2. For fixed δ , determine $S^{(\varepsilon, \delta)}(3)$ as $\varepsilon \rightarrow 0$. In particular, is our construction optimal?

3. The interpretation of a network as a string, and the probability of fault being the probability of a random substring not containing some particular patterns give rise to questions in a more general setting, which may be of interest by themselves.

REFERENCES

- [1] M. AJTAI, J. KOMLOS AND E. SZEMEREDI, *An $O(n \log n)$ sorting network*, Proc. 15th Annual ACM Symposium on Theory of Computing, April 1983, Boston, pp. 1-9.
- [2] K. E. BATCHER, *Sorting networks and their application*, Proc AFIPS 1968 SJCC, Vol 32, AFIPS Press, Montvale, NJ, pp. 307-314.
- [3] D. J. KLEITMAN, A. R. MEYER, R. L. RIVEST, J. SPENCER AND K. WINKLEMANN, *Coping with errors in binary search procedures*, Proc. 10th Annual Symposium on Theory of Computing, San Diego, CA, 1978, pp. 227-232.
- [4] D. E. KNUTH, *The Art of Computer Programming* Vol. 1, 2nd edition, Addison-Wesley, Reading, MA.
- [5] ———, *The Art of Computer Programming* Vol. 3, 2nd edition, Addison-Wesley, Reading, MA, 1975.
- [6] E. F. MORE AND C. SHANNON, *Reliable circuits using less reliable relays I-II*, J. Franklin Inst., 262 (1956), 3, pp. 191-208, no. 4, pp. 281-297.
- [7] *Proceedings of the IEEE*, special issue on fault-tolerant digital systems, 66 (1978), pp. 1105-1300.
- [8] J. VON NEUMANN, *Probabilistic logics and the synthesis of reliable organisms from unreliable components*, in Automata Studies, Princeton Univ. Press, Princeton, NJ, 1956, pp. 43-98.
- [9] A. C. YAO AND F. F. YAO, *Lower bounds on merging networks*, J. Assoc. Comput. Mach., 23 (1976), pp. 566-571.

ON THE EXPECTED PERFORMANCE OF PATH COMPRESSION ALGORITHMS*

ANDREW C. YAO†

Abstract. We consider the expected running time of an equivalence algorithm using the path compression rule (but not the weighting rule). An $O(n)$ expected running time is proved for the execution of a random equivalence program in the Spanning Tree Model.

Key words. equivalence program, expected running time, path compression, set merging, spanning tree model

1. Introduction. Let S be a set of n elements. An *equivalence program* σ on S is a sequence of *equivalence instructions* ($x[1] \equiv y[1], x[2] \equiv y[2], \dots, x[m] \equiv y[m]$) with each $x[i], y[i] \in S$. Starting with n equivalent classes each containing one element, an equivalence instruction $x[i] \equiv y[i]$ asks whether $x[i]$ and $y[i]$ currently belong to different equivalent classes, and if so requests that the two classes be merged. Equivalence programs have many applications, such as the processing of EQUIVALENCE statements in FORTRAN [4]. A common method to implement an equivalence program is by using a *set merging scheme*. A set merging scheme (see AHU [1], Tarjan [7]) maintains the equivalence classes as sets and processes commands of the forms FIND (x) and UNION (A, B). The command FIND (x) requires that the name of the set containing x be returned, and the command UNION (A, B) asks that the two sets with names A and B be merged into one. To implement an equivalence program using a given set merging scheme, one need only replace each equivalence instruction $x[i] \equiv y[i]$ by the sequence FIND ($x[i]$), FIND ($y[i]$), UNION (A, B), where A, B are the names of the sets containing $x[i], y[i]$ (omit the UNION if $A = B$). In this paper, we are interested in the expected running time of a random equivalence program, when a particular set merging scheme is used. This set merging scheme uses a forest data structure, and employs a path compression rule [1], [7]; we will refer to this scheme as *quick merge with path compression* (or, QMP). The expected performance of equivalence algorithms using other set merging schemes has been extensively studied in Knuth and Schönage [5], Yao [9]. It seems reasonable to regard the expected performance on equivalence programs as a benchmark for the average-case behavior of a set merging scheme (Doyle and Rivest [2] discussed the expected cost of a set merging scheme by considering a sequence of random FINDs and UNIONs directly, however).

In the QMP set merging scheme, the family of subsets (equivalence classes) are represented by a forest of disjoint rooted trees. Each tree corresponds to a subset, with the name of that subset stored at the root. Command FIND (x) accesses the node v representing x and triggers a traversal up the tree to its root r . In addition to returning the name of the subset, FIND (x) also performs a *path compression* from v to r , i.e., connecting every node on the path directly to r . Command UNION(A, B) is implemented by attaching the root for subset A to that for subset B . For definiteness, we charge 1 time unit for UNION and l time units for FIND, where l is the number of

* Received by the editors July 31, 1981, and in final revised form November, 1983. This research was done while the author was visiting the Computer Science Department, IBM San Jose Research Center, 5600 Cottle Road, San Jose, California. This work was supported in part by the National Science Foundation under grant MCS-77-05313-A01.

† Computer Science Department, Stanford University, Stanford, California 94305.

nodes on the traversed path. It is known that, with QMP, the worst-case time for performing a sequence of $O(n)$ UNIONS and FINDs is $\Theta(n \log n)$, where the lower bound proof is due to Fischer [3] and the upper bound is due to Paterson [6].

We now describe the randomness assumption we will use on the equivalence program. Let \mathcal{T}_n be the set of all equivalence programs $\sigma = (x[1] \equiv y[1], x[2] \equiv y[2], \dots, x[n-1] \equiv y[n-1])$ such that the set of edges $\{x[i], y[i]\}$, $1 \leq i < n$, forms a spanning tree for the set S . Clearly, $|\mathcal{T}_n| = n^{n-2}(n-1)!2^{n-1}$, where the last factor 2^{n-1} accounts for the fact that each edge $\{a, b\}$ can appear as either $a \equiv b$ or $b \equiv a$. Let us consider the *spanning tree model* [5] [9], in which each equivalence program $\sigma \in \mathcal{T}_n$ is equally likely. Let C_n^{QMP} be the expected running time of a random σ when QMP is used.

Our main result is the following theorem.

THEOREM 1. $C_n^{\text{QMP}} = O(n)$.

As an intermediate step, we will prove a result of some independent interest (Theorem 2 below) that applies to the expected running time under any randomness assumption belonging to a general class.

Consider a model τ for random equivalence programs, specified by a probability distribution $p_\tau(\sigma)$. We call τ a *canonical model* if $p_\tau(\sigma) = 0$ for all $\sigma \notin \mathcal{T}_n$. For each σ , let C_σ^{QMP} be the running time of σ when QMP is used. The expected running time is then $C_\tau^{\text{QMP}} = \sum_\sigma p_\tau(\sigma) C_\sigma^{\text{QMP}}$. Note that the Spanning Tree Model is a canonical model $\tau_0^{(n)}$ such that $P_{\tau_0^{(n)}}(\sigma) = 1/|\mathcal{T}_n|$ for all $\sigma \in \mathcal{T}_n$.

For any σ , let $W_i(\sigma)$ be the new equivalence class obtained from the merge of the two equivalence classes containing $x[i]$ and $y[i]$, when the i th instruction $x[i] \equiv y[i]$ of σ is performed. Let $\alpha_\sigma = \sum_i \log_2 |W_i(\sigma)|$. For a model τ , define the *potential* of τ as $H_\tau = \sum_\sigma p_\tau(\sigma) \alpha_\sigma$.

THEOREM 2. For any canonical model τ ,

$$C_\tau^{\text{QMP}} \leq 2H_\tau + 5(n-1).$$

In § 2 we briefly review Paterson's proof [6] for the worst-case upper bound on the QMP running time. In § 3 we establish Theorem 2, which involves a refinement of Paterson's analysis. We then prove Theorem 1 in § 4 by using Theorem 2. Some remarks and open problems are given in § 5.

2. Paterson's entropy. In this section we review Paterson's proof [6] for the QMP worst-case upper bound.

Let T be a rooted forest with only internal nodes. For each $v \in T$, let $w_T(v)$ be the number of descendants of v (including itself). The *entropy* of T is defined to be $H_0(T) = \sum_{v \in T} \log_2 (w_T(v))$. Clearly $H_0(T) \leq n \log_2 n$, if T has n nodes.

LEMMA 1 (Paterson [6]). *Suppose a path compression of length $t+2$ is performed along a path $v_0, v_1, \dots, v_t, v_{t+1}$ in T . Let T' be the new resultant tree. Then there exists a $1 < \beta \leq (\omega_T(v_t))^{1/t}$ such that*

$$H(T) - H(T') \geq t \log_2 \frac{\beta}{\beta - 1}.$$

Proof (sketch). The expression

$$H(T) - H(T') = \sum_{1 \leq i \leq t} (\log_2 w_T(v_i) - \log_2 (w_T(v_i) - w_T(v_{i-1})))$$

(under the constraint $1 \leq w_T(v_0) < w_T(v_1) < \dots < w_T(v_t)$) is minimized when $\{w_T(v_i)\}$ form a geometric progression $\{\alpha\beta^i\}$ with $\alpha \geq 1$, $\alpha\beta^t = w_T(v_t)$. The lemma follows by an explicit evaluation. \square

For any sequence σ of $O(n)$ UNIONS and FINDs, one can equivalently first carry out all the UNIONS, followed by the FINDs (which are now “partial” FINDs as the path compressions may end at nodes other than the roots of the forest) [1]. Let T_σ be the forest obtained after all the UNIONS are performed, but before any FIND is. The subsequent (partial) FINDs will modify the forest and decrease its entropy. Each FIND with cost $t+2 > \log_2 n + 2$ will decrease the entropy by at least t , according to Lemma 1. Thus, the total cost for FINDs is bounded by $H_0(T_\sigma) + O(n)$, plus the costs due to the FINDs with individual cost $\leq \log_2 n + 2$. It follows that the total cost for the FINDs is $O(n \log n)$; the UNIONS, of course, only cost $O(n)$. This finishes Paterson’s proof of an $O(n \log n)$ upper bound for QMP.

3. Proof of Theorem 2. An equivalence program σ induces a sequence σ' of UNIONS and FINDs that the QMP algorithm actually executes. We will use T_σ to stand for $T_{\sigma'}$.

LEMMA 2. For any $\sigma \in \mathcal{T}_n$,

$$C_\sigma^{\text{QMP}} \leq H_0(T_\sigma) + \alpha_\sigma + 5(n-1).$$

Proof. Let $\sigma = (x[1] \equiv y[1], x[2] \equiv y[2], \dots, x[n-1] \equiv y[n-1])$, and $S_{2i-1} \subseteq S, S_{2i} \subseteq S$ be the components containing $x[i], y[i]$ just before the i th equivalence instruction is executed. Consider the sequence of path compressions $\xi_1, \xi_2, \dots, \xi_{2n-2}$ that QMP carries out on T_σ , where ξ_{2i-1} is induced by FIND ($x[i]$) and ξ_{2i} by FIND ($y[i]$). Let l_i be the length of ξ_i ; define $J_1 = \{i | l_i > \log_2 |S_i| + 2\}$ and $J_2 = \{i | l_i \leq \log_2 |S_i| + 2\}$. Let $A_1 = \sum_{i \in J_1} t_i$ and $A_2 = \sum_{i \in J_2} t_i$, where $t_i = l_i - 2$. As each UNION only costs one unit time, we have

$$C_\sigma^{\text{QMP}} = n - 1 + \sum_i l_i \leq 5(n-1) + A_1 + A_2.$$

We first prove $A_1 \leq H_0(T_\sigma)$. Each ξ_i will successively modify the forest T_σ and decrease the entropy of the forest by at least $t_i \log_2 (\beta / (\beta - 1))$ according to Lemma 1. It is easy to see that the quantity $w_T(v_i)$ in Lemma 1 is at most $|S_i|$. It follows that $1 < \beta \leq (w_T(v_i))^{1/t_i} \leq 2$ (since $t_i > \log_2 |S_i|$). The entropy decrease is thus at least t_i . This implies $H_0(T_\sigma) \geq \sum_{i \in J_1} t_i = A_1$.

To finish the proof of Lemma 2, we need only prove $A_2 \leq \alpha_\sigma$. By definition,

$$A_2 \leq \sum_{i \in J_2} \log_2 |S_i| \leq \sum_{1 \leq i \leq 2n-2} \log_2 |S_i|.$$

Observe that, except for the $n-1$ S_i with $|S_i| = 1$, the S_i are in one-to-one correspondence with the $W_j(\sigma)$ in the expression $\alpha_\sigma = \sum_{1 \leq j \leq n-1} \log_2 |W_j(\sigma)|$. This proves $A_2 \leq \alpha_\sigma$. \square

LEMMA 3. For any $\sigma \in \mathcal{T}_n$,

$$H_0(T_\sigma) \leq \alpha_\sigma.$$

Proof. Let $v \in T_\sigma$ and let D_v be the subtree rooted at v . In the formation of T_σ , there is a unique UNION instruction that makes D_v a component of the forest; let $x[i_v] \equiv y[i_v]$ denote the equivalence instruction that induces this UNION. Clearly, $|W_v(\sigma)| = w_{T_\sigma}(v)$. It is also evident that distinct v give distinct i_v . Hence

$$H_0(T_\sigma) \leq \sum_{1 \leq j < n} \log_2 |W_j(\sigma)| = \alpha_\sigma \quad \square$$

It follows from Lemmas 2 and 3 that $C_\sigma^{\text{QMP}} \leq 5(n-1) + 2\alpha_\sigma$. Taking the average with weight $p_r(\sigma)$, we obtain Theorem 2.

4. Proof of Theorem 1. Because of Theorem 2, it suffices to prove $H_\tau = O(n)$ for $\tau = \tau_0^{(n)}$. Take a random σ in the Spanning Tree Model. Let p_{nk} be the probability that $|V_x| = k$ and $|V_y| = n - k$, where V_x and V_y are the two components containing $x[n-1]$ and $y[n-1]$ just before the last instruction $x[n-1] \equiv y[n-1]$ in σ . It is easy to verify the following facts:

$$(A) \quad p_{nk} = \frac{1}{2(n-1)} \binom{n}{k} \left(\frac{k}{n}\right)^{k-1} \left(\frac{n-k}{n}\right)^{n-k-1}.$$

(B) Let σ_x be the subsequence of σ acting on V_x , and σ_y be the subsequence of σ acting on V_y ; then σ_x is a random equivalence program in the spanning tree model $\tau_0^{(k-1)}$, and similarly σ_y is a random equivalence program in the model $\tau_0^{(n-k)}$.

Fact (B) is immediate from the definition of a random equivalence program. Fact (A) follows from a simplification of the equation

$$p_{nk} = \frac{\binom{n}{k} k(n-k) \binom{n-2}{k-1} k^{k-2} (k-1)! 2^{k-1} (n-k)^{n-k-2} (n-k-1)! 2^{n-k-1}}{n^{n-2} (n-1)! 2^{n-1}}.$$

In the above expression, the factor $\binom{n}{k}$ comes from enumerating the choice of V_x and V_y , $k(n-k)$ comes from enumerating the choice of $x[n-1]$ and $y[n-1]$ within V_x and V_y ; $\binom{n-2}{k-1}$ is the number of ways to interleave σ_x and σ_y , and the remaining numerators give the number of possible σ_x and σ_y . (Facts (A) and (B) were also shown in [5, § 9], although the spanning tree model there was phrased in a slightly different language.)

Let $r_n = H_\tau$ where $\tau = \tau_0^{(n)}$. It follows from Fact (B) that

$$r_n = \log_2 n + \sum_{0 < k < n} p_{nk} (r_k + r_{n-k}) \quad \text{for } n > 1,$$

and

$$r_1 = 0.$$

A recurrence relation of this form with p_{nk} as in (A) was studied in Knuth and Schönhage [5, eq. (12.8)], where it was shown that the solution satisfies $r_n = O(n)$.

We have proved Theorem 1.

5. Remarks. For any equivalence program $\sigma = (x[1] \equiv y[1], x[2] \equiv y[2], \dots, x[n-1] \equiv y[n-1]) \in \mathcal{T}_n$, consider the *union tree* Y_σ defined in Knuth and Schönhage [5, § 13] as follows: For $1 \leq i < n$, construct a new node whose left subtree is the union tree for the current component containing $x[i]$ and whose right subtree is the union tree for the current component containing $y[i]$ (“current” means just before performing the instruction $x[i] \equiv y[i]$); the union tree for a single element is a leaf. (Note that Y_σ is different from the tree T_σ considered before, as can be seen from the fact that Y_σ is always a binary tree.) We can regard α_σ as the “potential” of the tree Y_σ , defined by $\sum_{v \in Y_\sigma} \log_2 w(v)$, where $w(v)$ is the number of leaf-descendants of v . Theorem 2 can then be described as “the expected running time of QMP in a canonical model τ is bounded by the average potential of a random union tree in τ ”.

In the Spanning Tree Model, we have shown that the equivalence algorithm using path compression has an expected $O(n)$ running time for carrying out $n-1$ equivalence instructions. It is easy to show by a similar argument that the expected running time of the first l instructions is $O(l)$. However, it is not known if the expected running time for performing the l th instruction is $O(1)$ for every $1 \leq l < n$. An interesting related open problem is the determination of the average rank of elements in the final forest data structure. In passing, we remark that there are algorithms that run in

worst-case $O(n)$ time on the spanning tree model, and in fact on any canonical model (Tarjan [8]).

Although our motivation for studying this model is mainly theoretical, the result may be relevant in some situations involving sparse graphs. Consider the processing of equivalence instructions in Kruskal's minimum spanning tree algorithm for random weighted input graph G_{en} , such that each connected graph with e edges on n vertices is equally likely to occur, and each of the $e!$ different permutations of edge weights is equally likely to happen. When $e = n - 1$, the distribution of the sequence of equivalence instructions is the same as in the spanning tree model considered in this paper. It is even plausible that as long as $e = O(n)$, the result obtained in the spanning tree model may give a better estimate of the cost than in other models, say, the random graph model [5], [9], as connectivity is a severe constraint (a random graph in that model does not become connected until $e = O(n \log n)$). It is an interesting open problem to confirm this conjecture, and more generally, to analyze the compression algorithm in this "random $G_{e,n}$ " model with general e, n .

Two other randomness models for equivalence programs have been discussed in the literature. In the *Doyle-Rivest model* [2]¹ any pair of equivalence classes is equally likely to be joined. It is not hard to show that $H_\tau = O(n)$ in this case; from the discussions in [5, § 13], one can obtain the recurrence $r_n = \log_2 n + 1/(n-1) \times \sum_k (r_k + r_{n-k})$, where r_n stands for H_τ with n elements. This implies an $O(n)$ expected running time for QMP. In the random graph model [5], [9], the expected time for QMP is an unresolved question. The present approach yields only a trivial $O(n \log n)$ bound, since a component of size $\Omega(n)$ is involved with probability $\Omega(1)$ in the l th equivalence instruction for $l > (\frac{1}{2} + \epsilon)n$, which gives $H_\tau = \Omega(n \log n)$. Bob Sedgewick (private communication, 1979) has done an extensive simulation up to 100,000 nodes. The running time appears definitely nonlinear in n , and is consistent with an $n \log n$ growth. A theoretical resolution of this case is a major remaining open problem in the analysis of set merging algorithms.

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] J. DOYLE AND R. L. RIVEST, *Linear expected time of a simple union-find algorithm*, Inform. Processing Lett., 5 (1976), pp. 146-148.
- [3] 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.
- [4] B. A. GALLER AND M. J. FISCHER, *An improved equivalence algorithm*, Comm. ACM, 7 (1964), pp. 301-303.
- [5] D. E. KNUTH AND A. SCHÖNHAGE, *The expected linearity of a simple equivalence algorithm*, Theoret. Comput. Sci., 5 (1978), pp. 281-315.
- [6] M. S. PATERSON, 1972, unpublished; a description of Paterson's proof was given in MIT class notes (Course 6.851J) by A. R. Meyer and M. J. Fischer, 1973.
- [7] R. E. TARJAN, *Complexity of combinatorial algorithms*, SIAM Rev., 20 (1978), pp. 457-491.
- [8] ———, *Worst-case analysis of set union algorithms*, to appear.
- [9] A. C. YAO, *On the average behavior of set merging algorithms*, (extended abstract), Proc. 8th Annual ACM Symposium on Theory of Computing, 1976, pp. 192-195.

¹ The original model in [2] is defined in terms of FINDs and UNIONs; we modify it here by replacing each UNION with an equivalence instruction and omitting the FINDs.

FINDING EXTREMAL POLYGONS*

JAMES E. BOYCE†, DAVID P. DOBKIN‡,

ROBERT L. (SCOT) DRYSDALE III§ AND LEO J. GUIBAS¶

Abstract. Given n points in the plane, we present algorithms for finding maximum perimeter or area convex k -gons with vertices k of the given n points. Our algorithms work in linear space and time $O(kn \lg n + n \lg^2 n)$. For the special case $k=3$ we give $O(n \lg n)$ algorithms for these problems. Several related issues are discussed.

Key words. extremal polygons, maximum area, maximum perimeter, geometric complexity

1. Introduction. In this paper we present efficient algorithms for certain geometric optimization problems in the plane. Typical of these problems is the following. We are given n points in the plane and wish to choose a convex k -gon with vertices k of the given points and whose perimeter is maximal. The special case $k=2$ is the classical problem of finding the *diameter* of a point set in the plane. An algorithm presented in this paper will find the maximum perimeter k -gon in time $O(kn \lg n + n \lg^2 n)$, and linear space. The correctness of our algorithm is based on certain interesting combinatorial properties of extremal perimeter polygons. Surprisingly, the same combinatorial properties hold for polygons extremal under other measures as well, such as area. Thus an isomorphic algorithm can be used to find the largest area convex k -gon with vertices k of the n given points (within the same time bound). For the case $k=3$, a special trick allows us to solve these problems in time $O(n \lg n)$.

We begin our presentation by studying in § 2, some of the combinatorial properties of extremal polygons. In § 3 we use these properties and a dynamic programming approach to develop an algorithm for finding a maximal *rooted* k -gon in time $O(kn \lg n)$. Then in § 4 we use the rooted polygon algorithm to obtain the results stated above.

The diameter, as well as some other variants of these problems are quite old but fast algorithms for them are relatively new. Several authors have given algorithms for particular cases which require that the convex hull of our given collection of points be found first. Shamos [Sh] was the first to present an algorithm for the diameter problem which works in linear time (once the convex hull is given). He also gave a linear algorithm for finding the maximum area quadrilateral with vertices four of the given points. Dobkin and Snyder [DS] gave a linear time algorithm for the maximum area triangle. Our dynamic programming ideas in § 3 are similar to those discussed by F. Yao in [YF]. The quadrangle inequality of that paper can be used to give us an $O(n^2 \lg n)$ algorithm for the maximum perimeter triangle problem. The problem of finding the minimum area ellipse containing a collection of points was the subject of

* Received by the editors June 23, 1982 and in revised form October 19, 1983.

† Department of Mathematics, Stanford University, Stanford, California 94305. The work of this author was supported by the National Science Foundation under grant MCS 77-23738.

‡ Department of Electrical Engineering and Computer Science, Princeton University, Princeton, New Jersey 08544. The work of this author was performed while he was visiting at Xerox Corporation, Palo Alto Research Center and was supported by the National Science Foundation under grant MCS83-03926.

§ Department of Mathematics, Dartmouth College, Dartmouth, New Hampshire. The work of this author was performed while he was visiting at Xerox Corporation, Palo Alto Research Center and supported by the National Science Foundation under grant MCS 77-05313.

¶ Xerox Corporation Palo Alto Research Center, Palo Alto, California 94304.

recent work by Post [P]. The routing paper of Dolev and Siegel [DoS] uses a divide and conquer approach similar to ours. Finally, A. Yao [YA] gave the first subquadratic algorithm for finding the diameter of a set of points in 3-space.

As is often the case in geometry arguments there is a considerable subtlety in some of the correctness proofs required. We exhibit examples which show that other plausible algorithms may fail to find the truly maximum k -gons. And our techniques do not obviously generalize to dimensions higher than 2, where even the diameter problem is not known to be solvable in nearly linear time. Again we give examples that show how plausible generalizations can fail. These remarks are amplified in § 5.

Some of our results dualize in a natural fashion. We can find under certain conditions, minimum area or perimeter k -gons surrounding (circumscribing) our collection of points, by exactly analogous techniques. A brief description of these results appears in § 6, along with a mention of some applications.

We have also considered the problem of obtaining *minimum* perimeter k -gons with vertices among our n points. We have an $O(k^4 n \lg n + (\frac{\pi k^2}{4}) k' kn)$ algorithm for this problem based on finding an extended Voronoi diagram of our points. We plan to report on this result elsewhere. The case of minimum area seems to be significantly harder (possibly because small perimeter implies that the vertices are well localized in space, but small area does not). The best bound for the minimum area triangle has been obtained by Dobkin and Munro [DM] and is $O(n^2 \lg n)$.

2. The structure of extremal polygons. In this section we investigate a number of properties possessed by maximal polygons in either the area or the perimeter sense. The next two sections use these properties in order to devise efficient algorithms for finding such polygons.

Informally speaking, it is very plausible that the vertices of maximal polygons should be sought *among* the extremal points in our collection. This intuition is brought out by the following theorem. See also Fig. 1. Note that we allow our polygons to contain duplicated vertices.

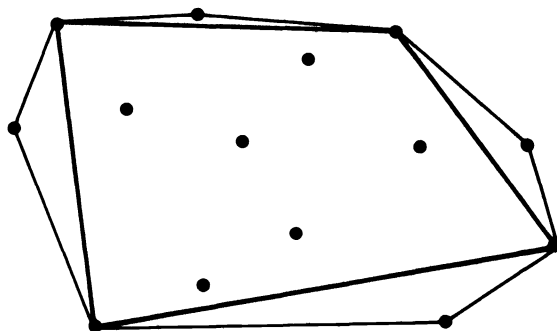


FIG. 1. Maximal k -gons use vertices of the convex hull.

THEOREM 1.1. *The vertices of convex k -gons maximal in area or perimeter are points on the convex hull of our collection of points.*

For perimeters a slightly stronger assertion is actually true. The vertices of maximal k -gons must be essential vertices (corners) of the polygon which is the convex closure of our point collection.

Proof. Let A be a vertex of a maximal k -gon and assume that A is interior to the convex hull. Let B and C denote the neighbor vertices of A on the k -gon.

If our k -gon is maximal in *area*, then consider a line parallel to line BC sweeping the plane from BC towards vertex A . Since A is not on the convex hull, some point A' other than A , will be the last point of our set encountered in the sweep of this line. Clearly the triangle $A'BC$ has larger area than ABC , and thus A' can be substituted for A to give us a larger area k -gon: a contradiction. (If A' makes the polygon nonconvex, then just replace it by its convex hull, which has still larger area.)

Similarly, if our k -gon is maximal in *perimeter*, then consider the ellipse passing through A , with B and C as foci, and its tangent at the point A . If A is interior to the convex hull, then there is a point of our set on the other side of the tangent from the ellipse, and that point can function as the point A' above, since $A'B + A'C > AB + AC$. (Again, convexification may be necessary. But it is well known that a convex polygon enclosing another has larger perimeter.)

Caution must be taken in extending vertices to the boundary. If vertices are moved in turn, a nonsimple polygon may result. However, we are able to circumvent this difficulty by moving all vertices at once along the bisectors of the exterior angles of the k -gon.

In the perimeter case, A must in fact be a corner of the convex hull, since if a point is constrained to lie on a line segment, its sum of distances to two other points is maximized at an endpoint of the segment. This is a simple consequence of the convexity of ellipses. Consider the smallest ellipse with foci the two other points and containing both endpoints of our segment. That ellipse contains in fact the whole segment.

If 2 or more vertices lie on an edge of the boundary, care must be taken in moving vertices to corners. In particular, 2 vertices spread to opposite. In the case of 3 vertices, the extremal vertices spread and the interior vertices need only be moved consistently. Although vertices of maximal area k -gons need not be corners of the convex hull, maximal area k -gons always exist which do have vertices corners of the convex hull. \square

In many situations the above theorem is a powerful tool, as the number of points on the convex hull of a collection of points is typically much less than the number of points in the collection. Several results are known in this direction which are summarized in Santalo [S]. Since we are interested in worst-case behavior, however, it may appear that this theorem does not help us at all, as in the worst case each of the points in our collection could be a vertex of the convex hull. Thus finding the convex hull need not reduce the number of points we must consider.

There is, however, another significant advantage to taking the convex hull, other than throwing away all points that are not vertices of it. This is that there is a well defined cyclic order among the remaining points. The exploitation of this cyclic order is the key so the subsequent lemmas on which the algorithms are based. From now on we will always assume that the points in our collection be on a convex perimeter and thus can be cyclically ordered. It is well known that the convex hull of a set of points can be computed in time $O(n \lg n)$ (see, for example, Graham [G]), and since all of our time bounds are larger than or equal to this, they will not be affected by assuming that this preprocessing step has been done. If fewer than k points are left as vertices of the convex hull, then we can stop. Our maximal polygon is the convex hull with some vertices taken with multiplicity greater than 1.

In the lemmas below we will be considering simple convex polygons with vertices some subset of our points. The ordering of the vertices along the polygon will agree with the cyclic ordering of the points discussed above. We will be interested in maximal polygons that are constrained in various ways. A *rooted* polygon will be a polygon with one of its vertices fixed at a given point. An *interval* is a collection of points

consecutive in the cyclic ordering. Two intervals will be called *nonoverlapping* if they intersect at most in a common endpoint. A *restricted* polygon is one whose vertices are constrained to be in successive nonoverlapping intervals. A *rooted restricted* polygon is a restricted polygon with one vertex constrained to be in a degenerate interval consisting of a single point. Two polygons with vertices points in our collection are said to *interleave*, if between every two successive vertices of one, there is a vertex of the other (possibly coinciding with one of them). It is clear how two k -gons may interleave. See Fig. 2. But a k -gon may also interleave an l -gon, for $k \neq l$, if they have some coincident vertices.

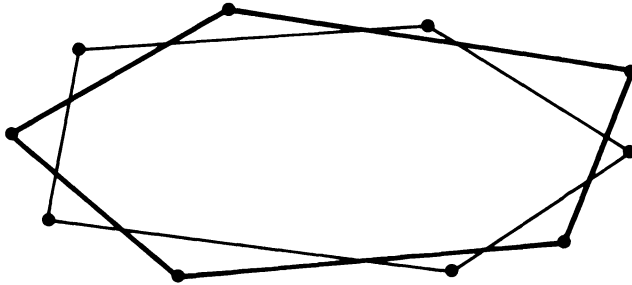


FIG. 2. Two interleaving k -gons.

The following lemmas apply to maximal k -gons in either the perimeter or area sense with vertices in our collection of points. (But both must be maximal in the *same* sense.) They provide the basis for our algorithms.

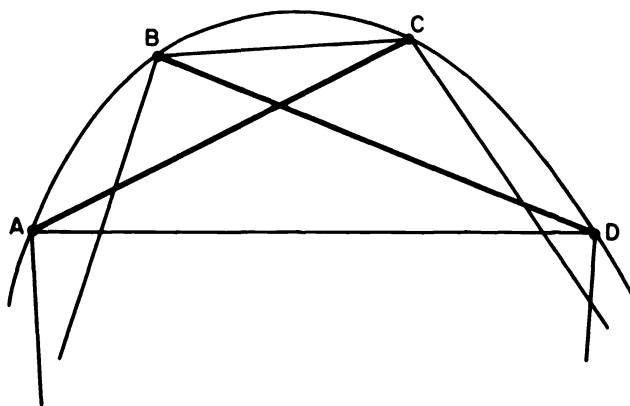
LEMMA 2.1. A maximal rooted k -gon and a maximal rooted $(k+1)$ -gon sharing the same root interleave.

LEMMA 2.2. A (globally) maximal k -gon and a maximal rooted k -gon interleave.

We will say that k consecutive nonoverlapping intervals I_1, I_2, \dots, I_k are *spanning*, if whenever a maximal k -gon has a vertex in one of them, it has exactly one vertex in each of them.

LEMMA 2.3. Let I_1, I_2, \dots, I_k be k spanning intervals, and let x be a point in I_1 . Consider the maximal restricted k -gon rooted at x , with its remaining vertices constrained to lie one in each of the intervals I_2, \dots, I_k respectively. The vertices of this k -gon subdivide each of our intervals into two nonoverlapping parts (both containing the subdividing vertex). Let these parts be called L , and R , for the interval I , in the order in which they occur along the cyclic order. Then both L_1, L_2, \dots, L_k , and R_1, R_2, \dots, R_k are spanning sets of k intervals.

The basic tool in the proofs of these lemmas is what we call the *crossing transform* applied to two polygons, a transformation that is always measure (perimeter or area) increasing. The idea of the crossing transform is illustrated in Fig. 3. It is applicable whenever we have two adjacent vertices of one polygon that are not separated by a vertex of the other. The transform simply interchanges the two noncrossing edges shown with the two crossing diagonal edges. Of course this now has merged our two polygons into a single (nonsimple) polygon. As it turns out however, in our context there will always be another place where the crossing transform can be applied as well, and this second application will break up this polygon into two simple polygons again. The two resulting polygons will be shown to have a combined measure that exceeds the measure of the two original polygons. This statement is in fact true even after the application of a single crossing transform, if we take care to define the perimeter and area of a nonsimple polygon appropriately. It also implies the following interesting combinatorial result.

FIG. 3. *The crossing transform.*

LEMMA 2.4. *If we are given $2k$ points forming the vertices of a convex $2k$ -gon, then the way to break them up into two groups of k each so as to maximize the sum of the perimeters of the two convex k -gons thus formed, is to use all the odd vertices for one k -gon and all even ones for the other.*

We now proceed to make precise the above informal remarks, and prove these lemmas. We intend to apply the notions of perimeter and area to nonsimple polygons, as well as polygons consisting of disjoint collections of vertex cycles. It is clear how to define the perimeter of any such polygon. For area, we need to be more careful. The area of a simple polygon can be thought of as the integral over the plane of a function which is 1 for points inside the polygon, and 0 for points outside. We will use an analogous definition for arbitrary polygons: we just integrate the *winding number*, which counts how many times the polygon wraps around each point (it can be a negative quantity). Thus for areas covered twice we multiply the ordinary area by two, and so on.

We now consider the effect of the crossing transform to a generalized polygon, which in our case consists of two normal convex polygons. The crossing transform applies whenever we have four vertices A, B, C, D occurring in this order in the cyclic ordering, but such that A and D are consecutive vertices of a polygon, as are B and C . The transform breaks the edges AD and BC , and adds the edges AC and BD . Note that each vertex still has degree two, so the outcome is a polygon. It is clear that this transform will always increase the perimeter of the resulting polygon, for we are replacing a pair of opposite sides of a convex quadrilateral with its diagonals. (It is a simple application of the triangle inequality to show that the diagonals of a convex quadrilateral always have longer total length than either pair of opposite sides.)

When it comes to area it is not true that the crossing transform always helps. For this conclusion we need some additional assumptions, which follow from the maximality of the two initial polygons. First we need an elementary geometric fact. Consider our standard convex quadrilateral $ABCD$, and let the sides AB and CD intersect at point X on the opposite side of side BC from A and D , and let the diagonals intersect at the point Y , as in Fig. 4. We claim that the area of BCY is less than the area of ADY . This is easily seen by adding to both triangles the triangle ABY . We must compare now the areas of triangles ABC and ABD . These triangles have a common base, AB , and the height of ABC from C is certainly smaller than that of ABD from D , as follows from the assumption that X and D on opposite half-planes with respect to BC .

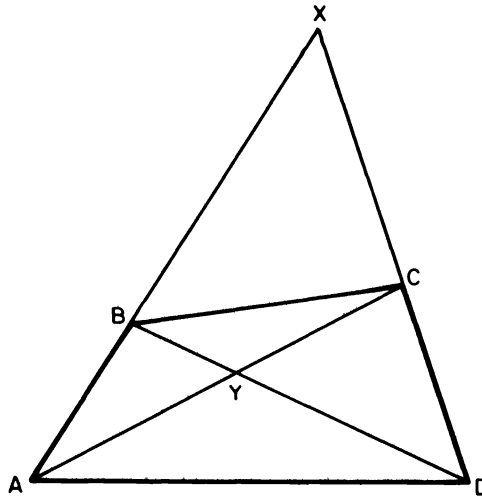
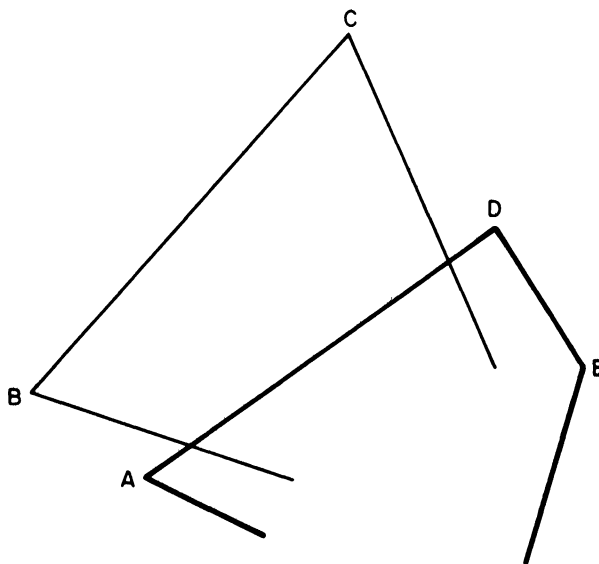


FIG. 4. A geometric inequality.

It is easy to verify that the change in the area of our polygon when the crossing transform is applied to the convex quadrilateral $ABCD$ is $a(ADY) - a(BCY)$, where a denotes the area function. Thus the crossing transform will increase the area, as long as we can guarantee that sides AB and CD intersect on the opposite half-plane of BC from A and D . This will be so, because of the assumption that the (simple) polygon containing edge AD is maximal. For suppose instead that AB and CD intersected on the same side of BC as A and D . Consider E , the other neighbor of D in the polygon containing AD , as in Fig. 5. By assumption AB intersects CD on the opposite side of AD from B and C . Since E lies between D and A in the cyclic ordering, it must a fortiori be the case that AE intersects CD on the other side of DE from A and C . By the same argument we gave above, triangle ACE has larger area than triangle ADE , contradicting the area maximality of the polygon containing AD .

FIG. 5. AB and CD must intersect on the other side of BC .

From what we said so far it follows that given the assumptions of Lemmas 2.1 and 2.2, whenever the crossing transform can be applied it will increase both perimeter and area. Let us think of the intervals defined by successive vertices of one of our polygons as buckets into which the vertices of the other polygon may fall. If a bucket gets no vertices, then these vertices can function as vertices B and C for an application of the crossing transform. From the pigeon-hole principle we know that if that happens then another bucket will get at least two vertices. Two consecutive of these can now function as B and C for another application of the crossing transform, this time with the role of the polygons reversed. It is easy to check that the first application of the crossing transform does not invalidate the second. So if the crossing transform can be applied once, it can be applied twice, and in fact the second application will give us back two simple polygons whose areas (or perimeters) sum to more than those of the original polygons. These polygons may not have the same number of vertices as the originals. However the polygons reached when the crossing transform is no longer applicable will interleave. This completes the proofs of Lemmas 2.1 and 2.2.

Given the machinery we have developed so far, we leave Lemmas 2.3 and 2.4 as simple exercises for the reader.

3. Finding maximal rooted polygons. In this section we develop an algorithm for finding a maximal rooted k -gon in time $O(kn \lg n)$. The algorithm proceeds in stages, by finding successively maximal rooted j -gons for $j = 3, 4, \dots, k$. We will postpone discussion of the initial case $j = 3$ and first talk about how we go from a maximal j -gon to a maximal $(j + 1)$ -gon with the same root.

Lemma 2.1 tells us that exactly one vertex of the $(j + 1)$ -gon, other than the root, must lie in each of the j intervals defined by the vertices of the maximal j -gon. We will use a dynamic programming method for finding the $(j + 1)$ -gon, examining each of the j intervals in turn. Note that if l_i denotes the length of the i th interval, then

$$\sum_{i=1}^j l_i = n + j.$$

The successive examination of intervals gives rise to the formation of partial (or incomplete) polygons and we must take a moment to properly define our measures of area and perimeter for such polygons. We introduce the notion of a *path*, which is just a sequence of vertices. We will use greek letters to denote paths, roman letters to denote points, and semicolon to signify concatenation. Our polygons correspond to closed paths, that is sequences of the form $p_0; p_1; p_2; \dots; p_j; p_0$. We define our measures of area and perimeter for paths as follows:

$$A(\alpha; p_{k-1}; p_k) = A(\alpha; p_{k-1}) + a(p_0; p_{k-1}; p_k),$$

and

$$P(\alpha; p_{k-1}; p_k) = P(\alpha; p_{k-1}) + l(p_{k-1}; p_k),$$

where $a(p_0; p_{k-1}; p_k)$ denotes the area of triangle $p_0 p_{k-1} p_k$, and $l(p_{k-1}; p_k)$ denotes the length of the edge $p_{k-1} p_k$.

For our dynamic programming algorithm we will use a multi-stage graph technique, as discussed for example, in Horowitz and Sahni [HS]. We maintain for each point z in the i th interval the best (i.e. maximal in measure) path with one point in each of the previous intervals and terminating at the root z . Let us denote such an optimal path by τ_z . Given these optimal paths for the i th interval, we now want to compute the optimal paths for the $(i + 1)$ st interval. This can clearly be done in time $l_i l_{i-1}$, by

considering for each point u in the $(i + 1)$ st interval each possible predecessor z in the i th interval. However, once again the idea of the crossing transformation will allow us to do better. This is captured in the following lemma.

LEMMA 3.1. *If y and z are two points on the $(i + 1)$ st arc defined by the vertices of a maximal rooted i -gon. $\beta_y; y'; y$ and $\beta_z; z'; z$ are respectively two optimal i step paths leading to y and z , and if y precedes z in the cyclic order within the $(i + 1)$ st arc, then y' precedes z' in the cyclic order within the i th arc.*

Again, the idea of the proof is that if this was not so then the last edges of the two paths do not cross, and therefore by applying the crossing transform one can get two other paths whose sum of measures exceeds the sum of measures of the old paths, a contradiction. The details are omitted, as they are identical to those discussed in § 2.

Lemma 3.1 implies that given the i th interval optimal paths, we can compute those for the $(i + 1)$ st interval in time $O(l_i \lg l_{i+1} + l_{i-1})$. This is so because we can choose z for the first time to be the median point of the $(i + 1)$ st interval and find the best path to it in l_i steps. Now the predecessor z' of z in the best path divides the i th interval into two subintervals of total length $l_i + 1$. We can now consider the $\frac{1}{4}$ and $\frac{3}{4}$ quartile points in the $(i + 1)$ st interval and for each of them search the appropriate subinterval of the i th interval. Thus together the cost of these searches will be $l_i + 1$. At the next step we will be able to do four points of the $(i + 1)$ st interval in total cost $l_i + 3$, and so on. Thus the total cost for all the searches is

$$\sum_{i=1}^{\lg l_{j+1}} l_i + 2^i - 1 = O(l_i \lg l_{i+1} + l_{i+1}).$$

Since each l_i is bounded by n , we can bound $\lg l_i$ by $\lg n$. If we now sum all the contributions for the successive stages, the total sum is clearly bounded by $O(n \lg n)$. (Note that the last stage is a bit funny, as the last interval contains exactly one vertex, namely the root.) We conclude that once we have a maximal rooted j -gon, we can compute a maximal rooted $(j + 1)$ -gon with the same root in linear space and time $O(n \lg n)$.

THEOREM 3.1. *A rooted constrained k -gon whose vertices are constrained to lie in intervals of total length n can be computed in time $O(n \lg n)$.*

To get started we note that, for the perimeter case, Lemma 2.1 holds even for $j = 2$, so we can begin by finding the maximal chord out of the root z , which is a linear time operation. For the maximum area case we need the following Lemma.

LEMMA 3.2. *The maximum area rooted triangle can be found in time $O(n)$.*

Proof. A method for doing this works like Shamos' diameter algorithm. Let A be the root, and consider its neighbor vertex B in the cyclic order. We can find vertex C , so as to maximize the area of triangle ABC by just examining further vertices along the cyclic ordering as long as the area keeps increasing. The distances of points on a convex figure to a chord form a unimodal distribution, so as soon as we pass the maximum, we know that we have found it. It is clear from convexity that if point B now advances along the cyclic ordering, then the best corresponding C also has to move in the same direction. Thus as B advances, C never has to back up, and this guarantees the linearity of the method. \square

This also follows, of course, from the results of [2]. combining the above observations we get the following result.

THEOREM 3.2. *A maximal rooted k -gon can be computed in time $O(kn \lg n)$ and linear space.*

4. Floating the root. In this section we show how to obtain a (globally) maximum (perimeter or area) k -gon. The word maximum, when used without other qualifiers,

will always refer to a global maximum. To start with, we find a maximum rooted k -gon, with root some arbitrary point z . This rooted k -gon partitions our points into k non-overlapping intervals I_1, I_2, \dots, I_k so that the maximum k -gon has exactly one vertex in each of them (by Lemma 2.2). We will show how, given this partitioning, we can find a maximum k -gon in an additional $O(n \lg^2 n)$ time.

We accomplish this by choosing one of these intervals, say I_1 , and then finding the maximum rooted k -gons with roots each of the points in I_1 . From § 3 we know that we can find a rooted restricted k -gon whose vertices are constrained to be in intervals of total length l in time $O(l \lg l)$. Naively applied, this would give us an $O(n^2 \lg n)$ algorithm for computing all these rooted k -gons. However, we can do better by proceeding exactly as in the previous section.

Lemma 2.3 implies that once we choose a point z in I_1 and find the maximum k -gon rooted there, then this k -gon will partition the original intervals into two collections, each of which is spanning. Thus again we can use a binary subdivision technique on I_1 , so that the cost of computing a maximum k -gon rooted at the median point of I_1 will be $O(n \lg n)$. Then the cost of computing maximum k -gons rooted at the $\frac{1}{4}$ and $\frac{3}{4}$ points of I_1 will jointly be bounded by $O(n \lg n)$, and so on, for $\lg n$ iterations. Therefore all the optimal k -gons with roots in I_1 can be found in time $O(n \lg^2 n)$. The maximum k -gon is the best of them.

THEOREM 4.1. *The maximum area or perimeter k -gon can be computed in time $O(kn \lg n + n \lg^2 n)$, and linear space.*

5. Comments and counterexamples. Our perimeter algorithm for $k=2$ is, of course, finding the diameter of our point set. Note that our method uses time $O(n \lg n)$ to find the diameter, even after the convex hull has been found. Shamos' method, on the other hand, requires only time $O(n)$ for that step. His method is based on supporting lines and uses the lemma that the diameter is always an edge between two points that are extremal along two directions in the plane, 180° apart. One can start two pointers at, say, the points of smallest and largest x value, and then rotate them around the convex hull so as to find all these extremal pairs. Neither pointer ever backs up, so the total cost for this method is $O(n)$.

Unfortunately, the supporting line technique does not generalize to $k > 2$. For example, to find a maximum perimeter triangle, we might consider all triplets of points which are extremal in three directions, 120° apart. As before, all these triplets can be found in linear time once the convex hull is given, but the example below shows that the maximum perimeter triangle need not be among them.

Consider the six points A, B, C, D, E, F with coordinates respectively $(0, 1)$, $(0, -1)$, $(100, 0)$, $(.3, 1.2)$, $(.3, -1.2)$, and $(99.9, .1)$. See Fig. 6. It is easy to check that ABC is the maximum perimeter triangle, but D and E always shelter either A or B from touching the circumscribing triangle, except when A and B are on the same line. In this case F shelters C .

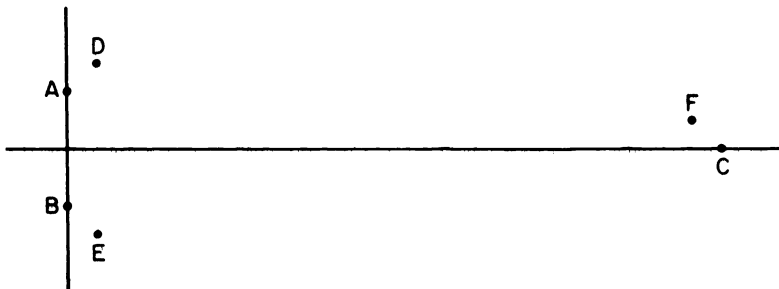


FIG. 6. *The supporting line fails.*

We have, however, a construction that reduces the problem of finding the maximum perimeter rooted triangle to that of finding the diameter of a certain set—a problem that we can solve with a supporting line idea in linear time. This implies that we can find a maximum perimeter triangle in time $O(n \lg n)$.

For our construction we consider the figure obtained by drawing a circle with center each of our points and radius its distance to the root (to be called the *flower*). See Fig. 7. Consider the root R , and the line segment joining it to some point X on a circle with center the point P , as in Fig. 8. The point X will not be contained in any other circle, if and only if point P is an extremal point of our original set in the direction RX . Thus the convex hull of the flower consists of alternating circular arcs and (possibly

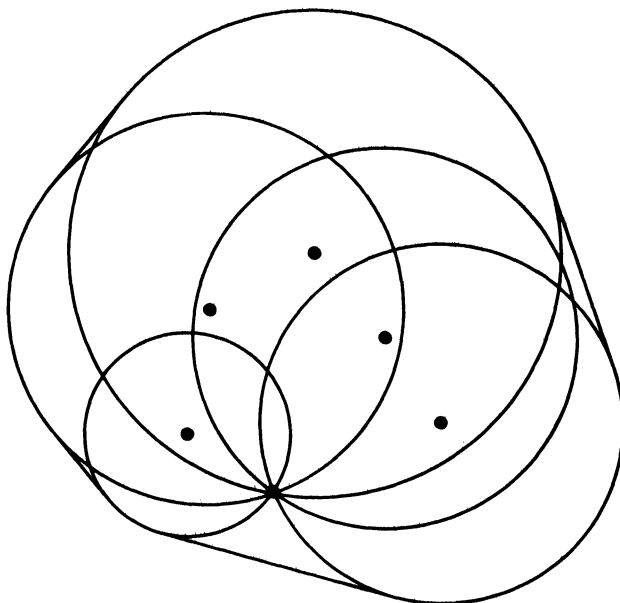


FIG. 7. Reducing a rooted triangle to a diameter problem.

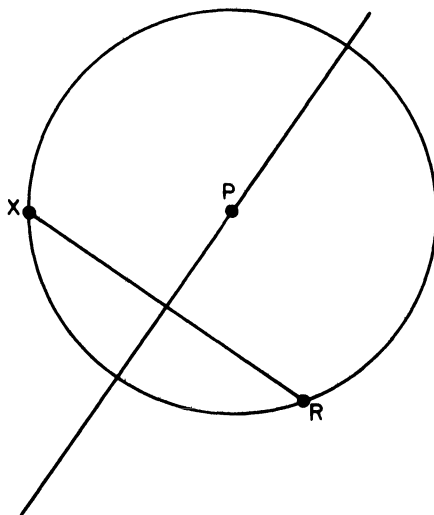


FIG. 8. A condition for point X to be on the boundary.

trivial) straight line segments. Furthermore there is at most one arc from each circle on the hull, and they occur in the same order as the original points. Well-known properties of convex sets [IB] imply that the diameter of the convex hull of the flower connects two points lying on circular arcs and, since the diameter has to be normal to the boundary, it passes through the centers of the corresponding circles. The triangle with vertices R and those centers has perimeter equal in length to this diameter.

Conversely, it is clear that any triangle under consideration corresponds to a line segment with endpoints on or interior to the convex hull and length equal to the perimeter of the triangle. Thus if we can find the diameter of the convex hull of the flower, we have found the maximum perimeter triangle rooted at R . It is easy to check that Shamos' diameter algorithm can be adapted to find the diameter of this continuous figure in linear time. This, coupled with the floating the root technique of § 4 shows the following theorem.

THEOREM 5.1. *The maximum perimeter triangle can be found in time $O(n \lg n)$.*

It may be of interest to note that the supporting line idea does not easily generalize to three dimensions, even for the diameter case, as the supporting hyperplanes at a vertex do not have a linear ordering. We might consider the following variant: let some face of the polyhedron act as a base, and look at the vertex furthest away from it. Let all edges from that vertex to some vertex of the base be candidates for the diameter. Next we roll the polyhedron onto a new base, and repeat this computation. After a Hamiltonian roll through all the faces, we may think that we have found the diameter. Unfortunately we have a simple example that shows that this method can miss.

Consider a right triangular prism with cross-section an equilateral triangle, and a height much larger than the side of the triangle. At each end of the prism construct a regular tetrahedron, using the end of the prism as a base. The prism is long enough that diagonal lines from one end of the prism to the other are only slightly longer than its height, so the diameter of the overall solid connects the apex of one tetrahedron to the apex of the other. However, the apex of a tetrahedron is not the furthest point from any face. The same situation occurs also in two dimensions if we are not careful about resolving ties, as we see if we append two slightly obtuse isosceles triangles to a rectangle, by glueing their long sides to the rectangle's short sides, as in Fig. 9.



FIG. 9. *A counterexample.*

While we are on the subject of counterexamples, it is worth mentioning that two maximal rooted k -gons do not necessarily interleave. The crossing argument breaks down when it forces both roots on the same polygon, and given below is an actual example for $k = 3$ that shows the existence of maximal noninterleaved rooted triangles in the perimeter case. Begin with an equilateral triangle inscribed in the unit circle, with one vertex at $(1, 0)$. Then construct a segment of length .2 tangent to the circle at each vertex, with the segment centered on the vertex. Now perturb this figure by raising A' and C' by .01, and lowering A and B by .01, as in Fig. 10. It can be checked that the largest perimeter triangle rooted at A is ABC , and the maximum rooted triangle rooted at A' is ABC . These do not interleave one another, but both interleave the overall maximum $C'BC$. A simple illustration of the same effect for $k = 2$ is in Fig. 11.

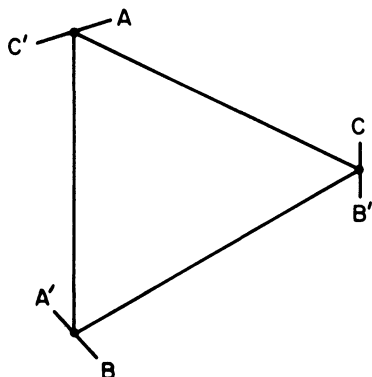


FIG. 10. Maximal rooted triangles do not interleave.

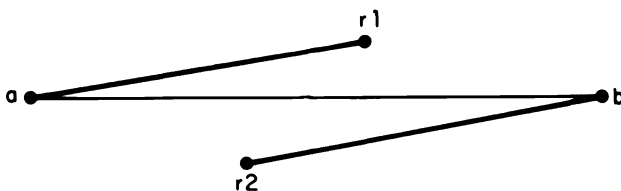


FIG. 11. Two maximal chords that do not cross.

We mentioned in the introduction some algorithms for finding minimum k -gons based on extended Voronoi diagrams. The key lemma there is that minimum k -gons occur as subpolygons of the points corresponding to a particular Voronoi region. One can also consider the furthest point Voronoi, and hope that similar techniques can be used for maximal k -gons. However, this is not obviously the case. Look at a regular $2k$ -gon. Its furthest point k -Voronoi consists of $2k$ wedges, each associated with the k consecutive points “opposite” the wedge. The largest k -gon on the $2k$ points is the regular k -gon using every second point, so it cannot be determined from this furthest point Voronoi.

Finally the Dobkin–Snyder method for finding maximal area triangles in linear time once the convex hull is given, fails to generalize to $k = 5$. Consider the seven points A, B, C, D, E, D', E' with coordinates respectively $(-101, 0), (0, 0), (0, -101), (-51, 1.01), (-50, 1), (1.01, -50), (1, -51)$, as in Fig. 12. The largest pentagon is

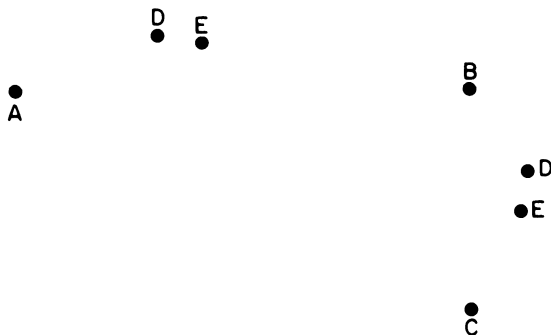


FIG. 12. Missing the largest area pentagon.

$ADBD'C$. The algorithm starts with $ADEBD'$. The D' point moves to C , giving $ADEBC$, then nothing moves. The base point is advanced, so we have $DEBD'C$. A is closer to the DD' line than C , so nothing moves. The base point advances, giving $EBD'E'C$. C advances to A and E' advances to C , giving $EBD'CA$. Nothing moves further. Once again we advance the base point, and get $BD'E'CA$. A will not move, so we advance the base point and get $D'E'CAD$. D advances to B , giving $D'E'CAB$, and then A will not move. Again advancing the base gives $E'CADB$, and nothing moves. Advancing the base again gives $CADEB$, and nothing moves. Finally the base moves a last time to $ADEBD'$, and we are back to where we started.

6. Circumscribed polygons. Our results dualize in an interesting fashion. We can consider lines, or actually halfspaces in the plane, instead of points. Keeping only points on the convex hull corresponds to keeping only those lines whose halfspaces support the intersection of all the halfspaces. Given n such halfspaces in the plane, we can find the k of them whose intersection has minimum area or perimeter by a dual of the original algorithm. Unfortunately this does not quite solve the problem of finding the minimum perimeter (or area) k -gon surrounding a given collection of n points.

A combination of the original and the dual algorithm lets us find an inscribed and a circumscribed k -gon for a collection of points. This is a useful tool for many computer graphics applications, such as hit detection or object intersection. If a point is inside the inscribed k -gon, then it is inside the convex hull of our n points. If it is outside the circumscribing k -gon, then it is outside the convex hull. If it falls in the crack between the two, then a more complicated method can be used.

Such inclusion tests are especially efficient for k -gons of fixed shape, e.g. rectangles. A supporting line idea can be used to find such minimum (in perimeter or area) circumscribing k -gons with sides at *fixed relative angles*. If we fix the orientations of all the sides, then we can find the smallest enclosing k -gon in time $O(kn)$. Once we have the vertices at which the sides of that polygon touch, we can let these vertices rotate around and obtain the smallest polygon for all orientation in time $O(k^2n)$. The extra factor of k comes in because we have to do an area or perimeter computation once we have determined the supporting vertices of the circumscribing k -gon.

Acknowledgments. The authors wish to express their thanks to Greg Nelson and Lyle Ramshaw for their contributions to the counterexamples discussed in § 5, and for many helpful discussions relating to the current work. They also wish to thank Sara Tietz for her assistance in the preparation of this manuscript.

REFERENCES

- [DL] D. P. DOBKIN AND R. J. LIPTON, *On the complexity of computation under varying sets of primitives*, J. Comp. Syst. Sciences, 18 (1979), pp. 86–91.
- [DS] D. P. DOBKIN AND L. SNYDER, *On a general method for maximizing and minimizing among certain geometric problems*, Twentieth IEEE Symposium on the Foundations of Computer Science, 1979, pp. 9–17.
- [DM] D. P. DOBKIN AND I. MUNRO, personal communication.
- [DoS] D. DOLEV AND A. SIEGEL, *The separation for general single-layer wiring barriers*, in VLSI Systems and Computations, Computer Science Press, Potomac, MD, 1981, pp. 143–152.
- [G] R. I. GRAHAM, *An efficient algorithm for determining the convex hull of a finite planar set*, Info. Proc. Lett., 1 (1972), pp. 132–133.
- [HS] E. HOROWITZ AND SARTAJ SAHNI, *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, MD, 1978.

- [IB] I. M. IAGLOM AND V. G. BOLTYANSKII, *Convex Figures*, Holt, Rinehart and Winston, New York, 1961.
- [P] M. J. POST, *A minimum spanning ellipse algorithm*, Twenty-second Annual IEEE Symposium on the Foundations of Computer Science, 1981, pp. 115–22.
- [S] L. A. SANTALO, *Integral Geometry and Geometric Probability*, Addison-Wesley, Reading, MA, 1976.
- [Sb] M. I. SHAMOS, *Problems in Computational Geometry*, unpublished manuscript, June 1974, revised May 1975, Feb. 1977.
- [YA] A. C. YAO, *Fast algorithms for minimum spanning trees in k dimensions*, Fifteenth Annual Allerton Conference on Communication Control and Computing. Sept 1977, pp. 553–556.
- [YF] F. F. YAO, *Efficient dynamic programming using quadrangle inequalities*, Twelfth ACM Symposium on the Theory of Computation, 1980, pp. 429–435.

SIMPLICITY, RELATIVIZATIONS AND NONDETERMINISM*

JOSÉ L. BALCÁZAR†

Abstract. Relativizations of complexity classes in which simple sets exist are considered. A recursive oracle is constructed relative to which a simple set exists for NP. Some other general theorems are proven, showing that the time bounds are not a crucial hypothesis; bounds on the way in which the oracle is accessible, namely, the number of queries and/or the number of nondeterministic steps, are shown to be the fundamental hypothesis. As a result, simple sets are shown to exist in many different relativized complexity classes.

Key words. complexity classes, relativizations, nondeterminism, bounded queries, immunity, simplicity, NP

Introduction. The relationship between deterministic and nondeterministic models of computation has been investigated for many years. The central problems appear to be fundamentally difficult. The open problem that has dominated recent work is the question of the deterministic and nondeterministic models restricted to polynomial running times, that is, the “ $P = ?NP$ ” problem.

A simple analogy may be drawn between the class P and the class of recursive sets on the one hand, and the class NP and the class of recursively enumerable sets on the other hand: the class NP can be defined by applying polynomially bounded existential quantifiers to predicates in P. Such an analogy suggests reasons for translating the definitions and, when possible, the results of elementary recursive function theory to the setting of polynomial time-bounded computation. As examples of these “translations,” recall the Hartmanis–Berman conjecture that all of the NP-complete sets are polynomially isomorphic, and the polynomial hierarchy specified by alternation of polynomially bounded quantifiers on predicates in P.

However, even elementary propositions of recursive function theory become difficult in the setting of polynomial time bounds; in fact, some are unsolved problems. Recently, two such notions have been investigated, the notion of “immune” set and the notion of “simple” set. Since the question $P = ?NP$ is open, it is not surprising that the existence of a “P-immune” set in NP or of a “NP-simple” set in NP is not known. This is the subject of the present paper.

The proof of Baker, Gill, and Solovay [1] of the existence of a set A such that $P(A) \neq NP(A)$ sets the stage for numerous investigations of the properties of relativizations of P and NP. Other such separating theorems have been developed and there are two specific studies that are important for the present work. First, Kintala [6], [7] considered relativizations of machines that run in polynomial time but have restrictions on the number of nondeterministic steps in any computation. Thus, there is a recursive set A such that for every integer k , the class of sets recognized relative to A by polynomial time-bounded oracle machines with at most n^k nondeterministic steps in any computation is properly included in the corresponding class specified by machines that may use n^{k+1} nondeterministic steps. Second, Xu, Doner, and Book [11] observed that the separating theorems proved by methods similar to Baker, Gill, and Solovay

* Received by the editors July 5, 1983, and in revised form April 17, 1984. The research reported in this paper was performed while the author visited the Department of Mathematics, University of California at Santa Barbara. This work was supported in part by a grant from the USA–Spanish Joint Committee for Educational and Cultural Affairs, and by the National Science Foundation under grants MCS80-11979 and MCS83-12472.

† Facultat d'Informàtica, Universitat Politècnica de Barcelona, Jordi Girona Salgado, 31, Barcelona, 34, Spain.

do not really depend on time as a bound, but rather the number of oracle queries allowed in a computation and the number of nondeterministic steps in a computation combine to yield these results. Thus, they established a very general separating theorem for relativizations of classes specified by machines with restricted nondeterminism.

If \mathcal{C} is a class of sets, then a set L_1 is \mathcal{C} -immune if L_1 is infinite and no infinite subset of L_1 is in \mathcal{C} , and a set L_2 is \mathcal{C} -simple if L_2 has an infinite complement, L_2 is in \mathcal{C} , and the complement of L_2 has no infinite subset in \mathcal{C} .

Homer and Maass [5] showed that there is a recursive set A such that there is a set in $\text{NP}(A)$ that is $\text{P}(A)$ -immune. Also, using priority methods, they showed that there is a recursively enumerable set B such that $\text{NP}(B)$ contains a set that is $\text{NP}(B)$ -simple. Schöning and Book [9] used a simple diagonalization in a different proof of the first result and they extended the argument to a wide variety of other classes by focusing not on time but rather on the number of oracle queries allowed in any computation and on the amount of nondeterminism allowed in any computation. Thus, Schöning and Book established two very general “immunity theorems” that establish “strong separation” of relativized classes, separations witnessed by the appropriate immune sets.

In this paper we establish a number of results about simple sets. The first result, Theorem 1, strengthens the result of Homer and Maass mentioned above: there is a recursive set A such that $\text{NP}(A)$ has a set that is $\text{NP}(A)$ -simple. The proof is by means of a straightforward diagonalization (a “slow” diagonalization in terms of [4]) and can be “lifted” to other circumstances. Thus, Theorem 3 and Theorem 5 provide very general results on the existence of simple sets that parallel the results of Schöning and Book. A number of applications are given.

These results add very strong evidence to the argument that the study of determinism vs. nondeterminism by means of relativizations has not illuminated the basic difficulties but instead has illustrated the power of nondeterminism in steps that write on the query tape and so generate a very large set of strings to be queried. This point is made stronger when one notes that Theorem 5 is established in the setting of an infinite hierarchy of functions that bound the amount of nondeterminism allowed in computations.

1. Preliminaries. Throughout this paper, we consider decision problems encoded as subsets of Γ^* where $\Gamma = \{0, 1\}$. For a word w , $|w|$ denotes the length of w . We assume some fixed ordering \cong of Γ^* such that $|x| < |y|$ implies $x < y$.

The computational model considered here is the multitape oracle Turing machine, deterministic or nondeterministic. For relativized computation oracle machines are assumed to have a distinguished work tape, the *query* tape, and three distinguished states, QUERY, YES, and NO. If some computation of such a machine enters the state QUERY, then at the next step the machine transfers into the state YES if the string currently appearing on the query tape is in a fixed *oracle set*; otherwise, the machine transfers into the state NO; in either case the query tape is instantly erased. For such a machine M and oracle set A , the set of strings *accepted by M* relative to the oracle set A is $L(M, A) = \{w \mid \text{there is an accepting computation of } M \text{ on input } w \text{ relative to oracle set } A\}$.

Oracle machines are defined in the standard way and may be bounded with respect to time or space by appropriate bounding functions. Time (space) bounds are assumed to be running times so that a “clock” may be added to any such machine. Querying the oracle costs just one step in time and the length of the query tape is bounded by whatever space bound is imposed.

Let f be a running time and let \mathbf{T} be a class of running times. We denote by $\text{NTIME}(f, A)$ the class of sets accepted by nondeterministic machines with running time f relative to oracle set A . Similarly, $\text{DTIME}(f, A)$ denotes the corresponding class specified by deterministic machines. Also, $\text{NTIME}(\mathbf{T}, A) = \bigcup_{f \in \mathbf{T}} \text{NTIME}(f, A)$ and $\text{DTIME}(\mathbf{T}, A) = \bigcup_{f \in \mathbf{T}} \text{DTIME}(f, A)$. Refinements of these classes will be introduced in § 4.

When considering machines that are nondeterministic, the expression “always halts” means that relative to every oracle set, every computation on every input must halt. For example, this is the case for time-clocked machines specifying classes such as $\text{DTIME}(f, A)$ or $\text{NTIME}(f, A)$.

Let \mathcal{C} be a class of subsets of Γ^* . Denote by $\text{co-}\mathcal{C}$ the class $\{\Gamma^* - L \mid L \in \mathcal{C}\}$. A set L is \mathcal{C} -immune if L is infinite and no infinite subset of L is in \mathcal{C} . A set L is \mathcal{C} -simple if L is in \mathcal{C} and $\Gamma^* - L$ is \mathcal{C} -immune. In what follows, for any set L , the set $\Gamma^* - L$ will be denoted \bar{L} .

2. A simple set for NP relativized. The first result is the existence of a recursive set A such that $\text{NP}(A) = \{L(M, A) \mid M \text{ is a nondeterministic polynomial time-bounded oracle machine}\}$ has a set that is $\text{NP}(A)$ -simple. The existence of a \mathcal{C} -simple set for any class \mathcal{C} shows a strong separation between \mathcal{C} and $\text{co-}\mathcal{C}$. The immunity results in [9] may be viewed as a strong separation between classes specified by deterministic machine vs. nondeterministic machines. Similarly, the existence of simple sets implies a strong separation between a class \mathcal{C} and the corresponding class $\text{co-}\mathcal{C}$, where \mathcal{C} is specified by nondeterministic machines; this separation is witnessed by a set in $\text{co-}\mathcal{C}$ which is not “infinitely approximable” within \mathcal{C} .

THEOREM 1. *There is a recursive set A such that $\text{NP}(A)$ contains a simple set.*

Proof. The basic construction diagonalizes over an enumeration of the clocked nondeterministic polynomial time-bounded oracle machines so that for any fixed oracle set A , each set in $\text{NP}(A)$ is presented infinitely often. Let $\text{NP}_1, \text{NP}_2, \dots$ be an enumeration of such machines; for each i , let q_i be a nondecreasing polynomial bounding NP_i 's running time.

For any set $A \subseteq \Gamma^*$, let $L(A) = \{w \mid w \notin \{0\}^*, \text{ or } w = 0^m \text{ and some word in } A \text{ has length of } m\}$. Clearly, $L(A) \in \text{NP}(A)$. The construction of A is based on a diagonalization over $\text{NP}(A)$ such that $L(A)$ is $\text{NP}(A)$ -simple. The set A is constructed in stages so that at each stage n , the intersection of $L(A)$ with $L(\text{NP}_j, A)$ for each $j \leq n$ is forced, when possible, to be nonempty.

Construct A by performing in the natural order $0, 1, 2, \dots$ the stages as follows:

Stage 0

$$A_0 := \{0\}^*;$$

$$m_0 := 0;$$

$$R_0 := \emptyset;$$

end stage;

Stage n ($n \geq 1$)

$$R_n := R_{n-1} \cup \{n\};$$

$$m_n := \min \{m \mid \max \{q_j(m_{n-1}) \mid j < n\} < m, \text{ and } \max \{q_j(m) \mid j \leq n\} < 2^m\};$$

$$A_n := A_{n-1} - \{0^{m_n}\};$$

if there exists $j \in R_n$ such that $0^{m_n} \in L(\text{NP}_j, A_n)$

then

let j_n be the least such j ;

choose any accepting computation of $L(\text{NP}_{j_n}, A_n)$ on input 0^{m_n} ;

let w_n be the least word of length m_n not queried in the chosen computation;

$$A_n := A_n \cup \{w_n\};$$

$$R_n := R_n - \{j_n\};$$

end if;

end stage.

The set A is defined as $A := \{x \in \Gamma^* \mid x \in A_n \text{ for almost every } n\}$.

The conditions imposed on m_n guarantee that adding or deleting words of length m_n does not alter the previous computations. In any single computation of a machine NP_i on an input x , at most $q_i(|x|)$ words can be queried; since there are 2^{m_n} words of length m_n , there is a word w_n available if it is needed. Thus, the construction can be performed.

It is clear that the set A is recursive. We show that $\overline{L(A)} = \Gamma^* - L(A)$ is infinite. By the definition of $L(A)$, $L(A)$ is finite if and only if $L(A)$ contains all but finitely many words of the form 0^{m_n} . This implies that words of length m_n are added to A in all but finitely many stages n ; hence, the "then" case occurred at all but finitely many stages, and by the construction we see that the set $R = \bigcup_{n \geq 0} R_n$ is finite; indeed, one number is added and one deleted at each such stage. But no index of the empty set can be deleted from R at any stage and every index is added to R at its own stage; since there are infinitely many indices of the empty set, R is infinite. Thus, $\overline{L(A)}$ is infinite as claimed.

Now suppose that for some j , $L(\text{NP}_j, A) \subseteq \overline{L(A)}$ and $L(\text{NP}_j, A)$ is infinite. Since $\overline{L(A)} \subseteq \{0^{m_n} \mid n \geq 0\}$, this means $L(\text{NP}_j, A) \subseteq \{0^{m_n} \mid n \geq 0\}$ so for infinitely many n , $0^{m_n} \in L(\text{NP}_j, A)$. Since only finitely many indices are less than j , there is some stage n such that j is the least index in R with $0^{m_n} \in L(\text{NP}_j, A)$. At this stage w_n is added to $A_n \subseteq A$ so that $0^{m_n} \in L(A)$, contradicting $L(\text{NP}_j, A) \subseteq \overline{L(A)}$. Thus, for any j , if $L(\text{NP}_j, A) \subseteq \overline{L(A)}$, then $L(\text{NP}_j, A)$ is finite.

Hence, $\overline{L(A)}$ is $\text{NP}(A)$ -immune and so $L(A)$ is $\text{NP}(A)$ -simple. \square

It is not difficult to combine this diagonalization with the one used by Schöning and Book [9] so that the resulting set A has the property that *simultaneously* $\text{NP}(A)$ has one set that is $\text{P}(A)$ -immune and another set that is $\text{NP}(A)$ -simple. Here we give only the construction. We assume an enumeration P_1, P_2, \dots of the clocked deterministic polynomial time-bounded oracle machines. For each i , let q_i be a nondecreasing polynomial bounding both P_i 's running time and also NP_i 's running time.

For any set $A \subseteq \Gamma^*$, define $L_{\text{even}}(A) = \{0^m \mid \text{there exists } w \in A \text{ such that } |w| = 2m\}$ and $L_{\text{odd}}(A) = \{0^m \mid \text{there is no } w \in A \text{ such that } |w| = 2m + 1\}$. Clearly, $L_{\text{even}}(A) \in \text{NP}(A)$ and $L_{\text{odd}}(A) \in \text{co-NP}(A)$.

The construction diagonalizes over $\text{P}(A)$ at even stages and $\text{NP}(A)$ at odd stages.

Stage 0

$$A_0 := \{0^k \mid k \text{ is odd}\};$$

$$m_0 := 0;$$

$$R_0 := \emptyset;$$

$$S_0 := \emptyset;$$

end stage;

Stage $2n - 1$ ($n \geq 1$)

$$R_{2n-1} := R_{2n-2};$$

$$S_{2n-1} := S_{2n-2} \cup \{n\};$$

$$m_{2n-1} := \min \{m \mid \max \{q_j(m_{2n-2}) \mid j < n\} < 2m + 1, \text{ and} \\ \max \{q_j(m) \mid j \leq n\} < 2^{2m+1}\};$$

$A_{2n-1} := A_{2n-2} - \{0^{2m_{2n-1}+1}\};$
if there is a $j \in S_{2n-1}$ such that $0^{m_{2n-1}} \in L(NP_j, A_{2n-1})$
then
 let j_{2n-1} be the least such j ;
 choose an accepting computation of $L(NP_{j_{2n-1}}, A_{2n-1})$ on $0^{m_{2n-1}}$;
 let w_{2n-1} be the least word of length $2m_{2n-1} + 1$ not queried
 in this computation;
 $A_{2n-1} := A_{2n-1} \cup \{w_{2n-1}\};$
 $S_{2n-1} := S_{2n-1} - \{j_{2n-1}\};$
end if;
end stage;
Stage $2n$ ($n \geq 1$)
 $R_{2n} := R_{2n-1} \cup \{n\};$
 $S_{2n} := S_{2n-1};$
 $m_{2n} := \min \{m \mid \max \{q_j(m_{2n-1}) \mid j < n\} < 2m, \text{ and } \sum_{j \leq n} q_j(m) < 2^{2m}\};$
if there is a $j \in R_{2n}$ such that $0^{m_{2n}} \in L(P_j, A_{2n-1})$
then
 let j_{2n} be the least such j ;
 $R_{2n} := R_{2n} - \{j_{2n}\};$
 $A_{2n} := A_{2n-1};$
else
 let w_{2n} be the least word of length $2m_{2n}$ not queried in any
 computation of $L(P_j, A_{2n-1})$ on $0^{m_{2n}}$ for every $j \in R_{2n}$;
 $A_{2n} := A_{2n-1} \cup \{w_{2n}\};$
end if;
end stage.

The set A is defined as $A := \{x \in \Gamma^* \mid x \in A_n \text{ for almost every } n\}$.

Arguments similar to those used for the first construction show that both $L_{\text{even}}(A)$ and $L_{\text{odd}}(A)$ are infinite, that $L_{\text{even}}(A)$ is $P(A)$ -immune, and that $L_{\text{odd}}(A)$ is $NP(A)$ -immune. Thus, we have the following result.

THEOREM 2. *There is a recursive set A such that $NP(A)$ has both a $P(A)$ -immune set and also an $NP(A)$ -simple set.*

It is not known whether there is a set A such that some set L in $NP(A)$ is simultaneously $P(A)$ -immune and $NP(A)$ -simple, that is, L is in $NP(A)$, L is infinite, no infinite subset of L is in $P(A)$, and no infinite subset of \bar{L} is in $NP(A)$. We continue to investigate this problem.

3. Simple sets for other relativized classes. The proof technique used to establish Theorem 1 is applicable to a wide variety of complexity classes other than NP . Clearly no class closed under complementation admits a simple set. But our investigation is concerned with the necessity of the polynomial time bound. A careful analysis of the proof of Theorem 1 shows that the polynomials are not used as a bound on the running times but rather that running times bound the number of oracle queries in computations and also the number of nondeterministic steps in computations. This is by no means surprising when one considers the results in [2], [3], [9], [10], [11]. In this section we state two results whose proofs are based on the constructions in § 2.

Let f be a function on the natural numbers. For any set A , let $L_f(A)$ be defined as $L_f(A) := \{0^m \mid \text{for all } w \in A, |w| \neq f(m)\}$.

If f is a running time, then it is clear that for every set A , $\overline{L_f(A)} \in \text{NTIME}(f, A)$. By diagonalizing over a class of machines, it is possible to construct A so that $L_f(A)$

is immune with respect to this class. Thus, if $\overline{L_f(A)}$ is in the class, then it will be simple with respect to this class.

THEOREM 3. *Let $\mathbf{M} = \{M_i \mid i \geq 1\}$ be an effective enumeration of a class of nondeterministic oracle machines that always halt, and let $\mathbf{T} = \{t_i \mid i \geq 1\}$ be a class of running times. For every set B , let $L(\mathbf{M}, \mathbf{T}, B)$ denote the collection of sets $L(M, B)$ such that M is in \mathbf{M} and for some t in \mathbf{T} and all inputs w to M , some accepting computation of M on w makes at most $t(|w|)$ oracle queries. Suppose that (i) for every $f, g \in \mathbf{T}$, $f(n) < 2^{g(n)}$ for all but finitely many n , and (ii) there is a finite set F such that for every set B , there are infinitely many i with $L(M_i, B) = F$. Then for any fixed $f \in \mathbf{T}$, there is a recursive set A such that $L_f(A)$ is $L(\mathbf{M}, \mathbf{T}, A)$ -immune; hence if $\overline{L_f(A)} \in L(\mathbf{M}, \mathbf{T}, A)$, then $\overline{L_f(A)}$ is $L(\mathbf{M}, \mathbf{T}, A)$ -simple.*

Clearly, Theorem 1 is a corollary of Theorem 3. Before proving it, let us show how the theorem applies. All these corollaries follow easily from Theorem 3.

For every set A , let $\text{NEXT}(A)$ denote the collection of sets recognized relative to A by nondeterministic oracle machines that run in time 2^{in} for some $i > 0$.

COROLLARY 3.1. *There is a recursive set A such that $\text{NEXT}(A)$ has a set that is $\text{NEXT}(A)$ -simple.*

For each integer $i > 0$, define $\text{exp}(2, 1, in) = 2^{in}$ and for integer $j > 0$, define $\text{exp}(2, j+1, in) = 2^{\text{exp}(2, j, in)}$. Fix an integer $h > 0$ and let $\mathbf{T} = \{\text{exp}(2, h, in) \mid i > 0\}$.

COROLLARY 3.2. *There is a recursive set A such that $\text{NTIME}(\mathbf{T}, A)$ has a set that is $\text{NTIME}(\mathbf{T}, A)$ -simple.*

For every set A , let $\text{NPQUERY}(A)$ ($\text{PQUERY}(A)$) be the collection of sets $L(M, A)$ where M is a nondeterministic (deterministic) oracle machine that uses polynomial work space and can make at most a polynomial number of oracle queries in any accepting computation. See [2] for interesting properties of these classes.

COROLLARY 3.3. *There is a recursive set A such that $\text{NPQUERY}(A)$ has a set that is $\text{NPQUERY}(A)$ -simple.*

If one considers the "bounded query" machines [2], [3] that specify classes of the form $\text{NPQUERY}(A)$ and allow bounds of the form $\text{exp}(2, h, in)$, then one can apply Theorem 3 to obtain a result similar to Corollary 3.3.

Let us turn to the general theorem.

Proof of Theorem 3. Without loss of generality we assume that each machine M_i operates within the bound $t_i \in \mathbf{T}$ on the number of queries. This can be achieved by constructing a new enumeration $M_{(i,j)}$ obtained by adding a "clock" t_j from \mathbf{T} that stops machine M_i if it attempts to query the oracle more than the allowed number of times. Then an effective "renaming" of the enumeration of \mathbf{T} allows us to assume that t_i bounds the number of queries of M_i .

Fix $f \in \mathbf{T}$ and perform the construction as follows. Note that it is an easy adaptation of the proof of Theorem 1.

Stage 0

$$A_0 := \{0\}^*;$$

$$m_0 := 0;$$

$$R_0 := \emptyset;$$

end stage;

Stage n ($n \geq 1$)

$$R_n := R_{n-1} \cup \{n\};$$

$$m_n := \min \{m \mid \max \{t_j(m) \mid j \leq n\} < 2^{f(m)} \text{ and } m \text{ is greater than } |w| \text{ for any } w \text{ queried to the oracle in a computation chosen at earlier stages}\};$$

$$A_n := A_{n-1} - \{0^{f(m_n)}\};$$

if there is a $j \in R_n$ such that $0^{m_n} \in L(M_j, A_n)$
then
 let j_n be the least such j ;
 fix an accepting computation that queries the oracle at most $t_{j_n}(m_n)$ times;
 let w_n be the least word of length $f(m_n)$ that has not been queried in the
 fixed computation;
 $A_n := A_n \cup \{w_n\}$;
 $R_n := R_n - \{j_n\}$;
end if;
end stage.

As in Theorem 1, the conditions imposed in m_n guarantee that the construction can be performed, and that previously considered computations do not change. On the other hand, such a m_n exists, as follows from the hypothesis. All but finitely many indices of the finite set cited in the hypothesis must remain forever in R , so $L_f(A)$ is infinite. For each i , if $L(M_i, A) \subseteq \{0\}^*$ and $L(M_i, A)$ is infinite, then at some stage n , the index i is removed from R_n . Thus, $L_f(A)$ is $L(\mathbf{M}, \mathbf{T}, A)$ -immune. \square

This theorem parallels the first immunity theorem in [9], which asserts that under similar hypotheses immune sets exist in relativizations of complexity classes (possibly, those specified by nondeterministic machines). Under hypotheses strong enough to imply both the hypothesis of Theorem 3 and that of the first immunity theorem, the constructions can be merged in the same way as was done in § 2 to obtain Theorem 2. We omit the proof; it involves no new ideas.

THEOREM 4. *Let \mathbf{T} be a class of running times, and let $\mathbf{M}_1 = \{M_{1,i} \mid i \geq 1\}$ and $\mathbf{M}_2 = \{M_{2,i} \mid i \geq 1\}$ be effective enumerations of deterministic and, respectively, nondeterministic oracle Turing machines that always halt. Further, assume that*

- (i) *for every $f \in \mathbf{T}$, and every integer, there is a $g \in \mathbf{T}$ such that $c \cdot f(n) < g(n)$ for almost all n ;*
- (ii) *for every $f, g \in \mathbf{T}$, $f(n) < 2^{g(n)}$ for almost all n ;*
- (iii) *for every i , there is a $t \in \mathbf{T}$ that bounds the number of queries in the computations of $M_{1,i}$;*
- (iv) *for every i , there is a $t \in \mathbf{T}$ that bounds the number of queries in some accepting computation on any input word $w \in L(M_{2,i}, A)$;*
- (v) *there is a finite set that appears infinitely often in both the classes $L(\mathbf{M}_1, \mathbf{T}, A)$ and $L(\mathbf{M}_2, \mathbf{T}, A)$;*
- (vi) *for some fixed $f_1, f_2 \in \mathbf{T}$ independent of A , the sets $\overline{L_{f_1}(A)}$ and $\overline{L_{f_2}(A)}$ are in $L(\mathbf{M}_2, \mathbf{T}, A)$ and the intersection of the range of f_1 and the range of f_2 is finite.*

Then there is a recursive A such that $\overline{L_{f_1}(A)}$ and $L_{f_2}(A)$ are, respectively, $L(\mathbf{M}_1, \mathbf{T}, A)$ -immune and $L(\mathbf{M}_2, \mathbf{T}, A)$ -simple.

Observe that hypotheses (i) and (ii) together imply that the sum of a finite fixed number of functions of \mathbf{T} is bounded by $2^{g(n)}$ for any $g \in \mathbf{T}$. This is the only nontrivial fact needed in applying the hypotheses to a construction similar to that used in the proof of Theorem 2.

Theorem 4 applies to the classes DEXT and NEXT, to the complexity classes specified by other hyperexponential time bounds $\exp(2, h, in)$ as defined above, to PQUERY and NPQUERY, and to many other complexity classes.

4. Refining nondeterminism. Some interesting work has been done in recent years regarding the existence of properly infinite hierarchies of relativized complexity classes where each class in the hierarchy is defined by bounding the number of nondeterministic steps the machines specifying the class are allowed to make. Kintala [6], [7] considered

oracle machines that operate in polynomial time. For each integer i and each set A , let $P(A)_{n^i}$ be the collection of all sets $L(M, A)$ where M operates in polynomial time and in every accepting computation on any input of length n , M can make at most n^i nondeterministic steps. In a similar way, for each integer i and each set A , define $P(A)_{(\log n)^i}$. Kintala showed that there are recursive sets A and B such that for all $i \geq 0$, $P(A)_{n^i} \subsetneq P(A)_{n^{i+1}}$, and $P(B)_{(\log n)^{i+1}} \subsetneq P(B)_{(\log n)^{i+2}}$. Xu, Doner, and Book [11] established a general separating theorem that yields Kintala's results as corollaries. Schöning and Book [9] established a general immunity theorem that yields as corollaries the existence of recursive sets A and B such that for all $i > 0$, $P(A)_{n^{i+1}}$ has a set that is $P(A)_{n^i}$ -immune and $P(B)_{(\log n)^{i+2}}$ has a set that is $P(B)_{(\log n)^{i+1}}$ -immune. Here we establish a general simplicity theorem that parallels the result of Schöning and Book.

A machine M operates in nondeterminism $g(n)$ if for every input string x to M , every computation of M on x has at most $g(|x|)$ nondeterministic steps.

Let \mathbf{M} be a class of nondeterministic oracle machines, and let \mathbf{T} and \mathbf{G} be classes of nondecreasing functions. Assume that for each M in \mathbf{M} there are functions $t \in \mathbf{T}$ and $g \in \mathbf{G}$ such that for every input string x to M , every computation of M on x makes at most $t(|x|)$ oracle queries, and M operates in nondeterminism g . Further, assume that for every $t \in \mathbf{T}$ and $g \in \mathbf{G}$ there is some M in \mathbf{M} satisfying this condition. For every set A and $g \in \mathbf{G}$, define $D(\mathbf{M}, A)_g = \{(M, A) \mid M \in \mathbf{M} \text{ operates in nondeterminism } g\}$.

THEOREM 5. *Let $\mathbf{M} = \{M_i \mid i \geq 1\}$ be an effective enumeration of a class of nondeterministic oracle machines that always halt, and let $\mathbf{T} = \{t[i] \mid i \geq 1\}$ and $\mathbf{G} = \{g[i] \mid i \geq 1\}$ be classes of running times such that $\langle \mathbf{M}, \mathbf{T}, \mathbf{G} \rangle$ satisfy the above conditions. Suppose that the following hold:*

- (i) *for every $t \in \mathbf{T}$ and $g \in \mathbf{G}$, $t(n) < 2^{g(n)}$ for almost all n ;*
- (ii) *for every $g \in \mathbf{G}$ and every set B , $\overline{L_g(A)} \in D(\mathbf{M}, B)_g$;*
- (iii) *there is a finite set F such that for all $g \in \mathbf{G}$, all $t \in \mathbf{T}$, and all sets B , there are infinitely many i such that $F = L(M_i, B)$, M_i operates in nondeterminism g , and for every input w , every accepting computation of M_i on w relative to B queries the oracle at most $t(|w|)$ times.*

Then there is a recursive set A such that for every $g \in \mathbf{G}$, the set $\overline{L_g(A)}$ is $D(\mathbf{M}, A)_g$ -simple.

Again we turn to some examples before giving the proof of Theorem 5.

COROLLARY 5.1. *There is a recursive set A such that for every $i > 0$, $P(A)_{n^i}$ has a set that is $P(A)_{n^i}$ -simple.*

COROLLARY 5.2. *There is a recursive set A such that for every $i > 1$, $P(A)_{(\log n)^i}$ has a set that is $P(A)_{(\log n)^i}$ -simple.*

For every set B and every integer $i > 0$, let $\text{PQUERY}(B)_{n^i}$ and $\text{PQUERY}(B)_{(\log n)^i}$ be the restrictions of $\text{NPQUERY}(B)$ (or extensions of $\text{PQUERY}(B)$) defined analogously to $P(B)_{n^i}$ and $P(B)_{(\log n)^i}$.

COROLLARY 5.3. *There are recursive sets A and B such that for every $i > 0$, $\text{PQUERY}(A)_{n^i}$ has a set that is $\text{PQUERY}(A)_{n^i}$ -simple and $\text{PQUERY}(B)_{(\log n)^{i+1}}$ has a set that is $\text{PQUERY}(B)_{(\log n)^{i+1}}$ -simple.*

Fix an integer $h > 0$ and let $\mathbf{T} = \{\exp(2, h, in) \mid i > 0\}$. For every set B and every integer $i > 0$, let $\text{DTIME}(T, B)_{\exp(2, h, in)}$ be the collection of all sets $L(M, B)$ where M operates in time bounded by some function in T and in nondeterminism $\exp(2, h, in)$, and let $\text{DQUSP}(T, B)_{\exp(2, h, in)}$ be the collection of all sets $L(M, B)$ where M operates in space bounded by some function in T , the number of oracle queries made in any of M 's accepting computations is bounded by some function in T , and M operates in nondeterminism $\exp(2, h, in)$. See [3].

COROLLARY 5.4. *There is a recursive set A such that $\text{DTIME}(T, A)_{\exp(2, h, in)}$ has a $\text{DTIME}(T, A)_{\exp(2, h, in)}$ -set simple set.*

COROLLARY 5.5. *There is a recursive set A such that $DQUSP(T, A)_{\exp(2,h,in)}$ has a $DQUSP(T, A)_{\exp(2,h,in)}$ -simple set.*

For other examples of classes to which Theorems 3–5 apply, see [3], [9], [10], [11].

Now we turn to the proof of Theorem 5. Let us assume any standard polynomial-time computable tripling function, so that $i = \langle \xi, \eta, \zeta \rangle$ is one-one and onto from \mathbb{N}^3 to \mathbb{N} . We assume that the inverses (the projections) are also polynomial-time computable.

Proof of Theorem 5. First we need a recursive presentation of $\bigcup_g D(\mathbf{M}, A)_g$. In order to do this, we build a new enumeration of machines which we call $\mathbf{M}' = \{M'_i \mid i \geq 1\}$, by constructing each machine M'_i , $i = \langle \xi, \eta, \zeta \rangle$, behaving like the ξ th machine in \mathbf{M} , M_ξ , with a clock for $t[\eta]$ that stops the machine if more than $t[\eta]$ queries are made, and with a clock for $g[\zeta]$ bounding the number of nondeterministic steps in the computations. Notice that the machines M'_i such that $i = \langle \xi, \eta, \zeta_0 \rangle$ for a fixed ζ_0 form a recursive presentation of $D(\mathbf{M}, A)_{g[\zeta_0]}$.

From the index of the machine M'_i it is possible to recover the bounds t and g corresponding to M'_i . For the sake of clarity, when these bounds are needed we will say “let t, g be the bounds corresponding to the machine M ”, instead of indicating the index of t and g by means of the projections.

Construct the oracle A by performing in their natural order stages $0, 1, \dots$, as follows:

Stage 0

$A_0 := \{0\}^*$;

$m_0 := 0$;

$R_0 := \emptyset$;

end stage;

Stage n (for $n \geq 1$)

$R_n := R_{n-1} \cup \{n\}$;

$A_n := A_{n-1}$;

$m_n := \min \{m \mid |w| < m \text{ for any } w \text{ queried at earlier stages, and } t(m) < 2^{g(m)} \text{ for the bounds } t \text{ and } g \text{ corresponding to machine } M_j, \text{ for each } 1 \leq j \leq n\}$;

$A_n := A_n - \{0^{g(m_n)} \mid g \text{ is the nondeterminism bound corresponding to machine } M_j, \text{ for each } 1 \leq j \leq n\}$;

if there is a $j \in R_n$ such that $0^{m_n} \in L(M_j, A)$

then

let j_n be the smallest such j ;

let t and g be the bounds corresponding to M_{j_n} ;

fix an accepting computation with query bound $t(m_n)$ and nondeterminism bound $g(m_n)$;

let w_n be the least word of length $g(m_n)$ not queried in this computation;

$R_n := R_n - \{j_n\}$;

$A_n := A_n \cup \{w_n\}$;

end if

end stage.

As in the construction in § 2, every index of the finite set F must remain in R_n from some stage on. So, case “then” must fail to appear infinitely often.

For any $g \in G$, once some machine operating in nondeterminism g has entered R_n , at each stage in which case “then” does not occur no word of length $g(m_n)$ is allowed to remain in A ; so $L_g(A)$ is infinite.

On the other hand, let M_j be any machine of \mathbf{M}' operating in nondeterminism g , and assume that $L(M_j, A)$ is infinite and that $L(M_j, A) \subseteq L_g(A)$; eventually a word

$0^{m_n} \in L(M_j, A)$ will be found, because after finitely many stages j must be the least index to be deleted from R_n . But this stage will add to A a word w_n of length $g(m_n)$ yielding $0^{m_n} \notin L_g(A)$. So, no infinite set in $D(\mathbf{M}, A)_g$ is included in $L_g(A)$.

Hence, $L_g(A)$ is $D(\mathbf{M}, A)_g$ -immune. But $\overline{L_g(A)} \in D(\mathbf{M}, A)_g$ by hypothesis, so $\overline{L_g(A)}$ is $D(\mathbf{M}, A)_g$ -simple. \square

It is possible to combine this result with the second immunity theorem in [9] in the same way as Theorems 2 and 4. No new ideas are needed. We omit this combined result.

Acknowledgments. The author wishes to thank Ronald Book and Uwe Schöning for their help, friendship, and kind sharing of their own ideas. Contributions of an anonymous referee to the clarity and correctness of this paper, particularly Theorem 5, are gratefully acknowledged. Finally, the author wishes to thank Ms. Leslie Wilson for her assistance with the manuscript.

REFERENCES

- [1] T. BAKER, J. GILL AND R. SOLOVAY, *Relativizations of the P = ?NP question*, this Journal, 4 (1975), pp. 431-442.
- [2] R. BOOK, *Bounded query machines: on NP and PSPACE*, Theoret. Comput. Sci., 15 (1981), pp. 27-39.
- [3] R. BOOK, C. WILSON AND XU MEI-RUI, *Relativizing time, space, and time-space*, this Journal, 11 (1982), pp. 571-581.
- [4] W. GASARCH AND S. HOMER, *Relativizations of the exponential time hierarchy*, unpublished manuscript, 1982.
- [5] S. HOMER AND W. MAASS, *Oracle dependent properties of the lattice of NP sets*, Theoret. Comput. Sci., 24 (1983), pp. 279-289.
- [6] C. M. R. KINTALA, *Computations with a restricted number of nondeterministic steps*, Ph.D. dissertation, Pennsylvania State University, 1977.
- [7] C. M. R. KINTALA AND P. FISCHER, *Refining nondeterminism in relativized polynomial time-bounded computations*, this Journal, 9 (1980), pp. 46-53.
- [8] N. LYNCH, *On reducibility to complex or sparse sets*, J. Assoc. Comput. Mach., 22 (1975), pp. 341-345.
- [9] U. SCHÖNING AND R. BOOK, *Immunity, relativizations, and nondeterminism*, this Journal, 13 (1984), pp. 329-337.
- [10] A. SELMAN, XU MEI-RUI AND R. BOOK, *Positive relativizations of complexity classes*, this Journal, 12 (1983), pp. 565-579.
- [11] XU MEI-RUI, J. DONER AND R. BOOK, *Refining nondeterminism in relativized complexity classes*, J. Assoc. Comput. Mach., 30 (1983), pp. 677-685.

POLYNOMIAL TIME ALGORITHMS FOR THE MIN CUT PROBLEM ON DEGREE RESTRICTED TREES*

MOON-JUNG CHUNG[†], FILLIA MAKEDON[‡],
IVAN HAL SUDBOROUGH[§] AND JONATHAN TURNER[¶]

Abstract. Polynomial algorithms are described that solve the MIN CUT LINEAR ARRANGEMENT problem on degree restricted trees. For example, the *cutwidth* or *folding number* of an arbitrary degree d tree can be found in $O(n(\log n)^{d-2})$ steps. This has applications to integrated circuit layout, in particular the layout of Weinberger arrays [41]. This also yields an algorithm for determining the black/white pebble demand of degree three trees. We also show that for degree three trees, cutwidth is identical to search number and give a forbidden subgraph characterization of degree three trees having cutwidth k .

Key words. MIN CUT LINEAR ARRANGEMENT problem, cutwidth, search number, black/white pebble demand, integrated circuit layout, VLSI

1. Introduction. Let $G = (V, E)$ be a finite undirected graph. A (one-dimensional) *layout* of G is a one-to-one function σ mapping the set of vertices V onto $\{1, 2, \dots, |V|\}$. We consider the following layout problem:

MIN CUT LINEAR ARRANGEMENT PROBLEM (MIN CUT)

Instance: A finite undirected graph $G = (V, E)$ and a positive integer k .

Question: Does there exist a layout σ such that, for all i ($1 \leq i < |V|$), there are at most k edges in the set $cut_\sigma(i) = \{\{x, y\} \in E \mid \sigma(x) \leq i < \sigma(y)\}$?

The *cutwidth* of G with respect to a layout σ , denoted $\gamma_\sigma(G)$ is defined as $\max \{|cut_\sigma(i)| : 1 \leq i < |V|\}$. The *cutwidth* of G , denoted $\gamma(G)$, is defined as the minimum over all layouts σ of $\gamma_\sigma(G)$. A simple example is shown in Fig. 1.

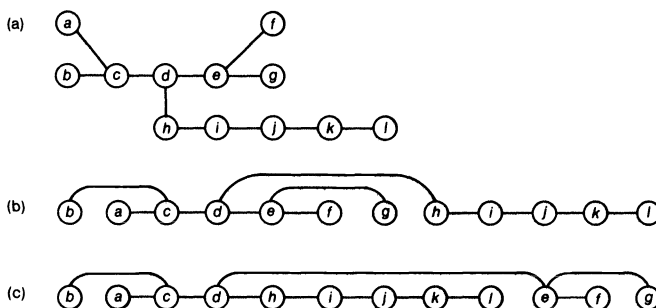


FIG 1. (a) A tree T . (b) A layout minimizing the sum of the edge lengths. (c) A layout minimizing the cutwidth.

* Received by the editors October 31, 1983.

[†] Department of Mathematical Sciences, Rensselaer Polytechnic Institute, Troy, New York. The research of this author was supported in part by the National Science Foundation under grants MCS 79-08919 and 81-09280.

[‡] Department of Computer Science, Illinois Institute of Technology, Chicago, Illinois 60616. The research of this author was supported in part by the National Science Foundation under grants MCS 79-08919 and 81-09280.

[§] Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois 60201. The research of this author was supported in part by the National Science Foundation under grants MCS 79-08919 and 81-09280.

[¶] Bell Laboratories, Murray Hill, New Jersey 07974. Current address, Computer Science Department, Box 1045, Washington University, St. Louis, Missouri 63130.

MIN CUT is one of several one-dimensional layout problems for undirected graphs. Other layout problems are the BANDWIDTH MINIMIZATION problem and the OPTIMAL LINEAR ARRANGEMENT problem [2], [9], [10], [13], [15], [22], [30], [38]. The MIN CUT problem for general graphs is known to be NP-complete [11], [39]. A recent result shows that it is NP-complete even when restricted to graphs with maximum vertex degree three [23]. F. R. K. Chung [3] and others call the cutwidth of a graph its “folding number.”

The complexity of the MIN CUT problem when restricted to trees has been an open problem of some recognized importance. Lengauer [19] described a polynomial time approximation algorithm for this problem. Lengauer’s algorithm obtains a layout σ for any tree T such that $\gamma_\sigma(T) < 2\gamma(T)$. In addition, Lengauer gave a linear time algorithm to obtain an optimum layout of a complete k -ary tree, where k is any positive integer. Some of the known applications for the MIN CUT LINEAR ARRANGEMENT problem when restricted to trees are discussed below.

1.1. Black/white pebble demand for binary trees. Let $G = (V, E)$ be a directed acyclic graph. The black/white demand of G is the minimum number of pebbles required to play the black/white pebble game on G . The rules of the black/white pebble game are:

- A white pebble may be placed on any vertex at any time.
- A white pebble may be removed from a vertex only if all the predecessors of that vertex are pebbled.
- A black pebble may be placed on a vertex only if the predecessors of that vertex are pebbled.
- A black pebble may be removed from a vertex at any time.

The object of the pebble game is to place a pebble on a distinguished vertex called the *sink*, using as few pebbles as possible.

Let $T = (V, E)$ be a directed binary tree with a sink vertex of degree one. The number of black and white pebbles needed to pebble the sink of T is equal to the cutwidth of the underlying undirected tree. (We wish to thank Nick Pippinger for pointing this out to us [34], [35], [36].) This can be seen by the following observations:

1. Let T be such a tree and let S be a black/white pebble game strategy for T using k pebbles. (We can assume that S does not involve recomputation [21].) A *defining move* of the sequence of steps in S is a move that either adds a black pebble to a vertex or deletes a white pebble from a vertex. Define a layout σ of T by $\sigma(x) = i$ if and only if the i th defining move of S involves vertex x . It follows that $\gamma_\sigma(T) \leq k$. This is because at the i th defining move if the edge (y, z) is in $cut_\sigma(i)$ then there is a pebble on y . Since every vertex has out-degree at most one, it follows that each of these pebbles is on a unique vertex. Hence the size of $cut_\sigma(i)$ is bounded by k for all i .

2. Let T be a directed binary tree whose sink has degree one. Given a cutwidth k layout σ of T we can construct a pebbling strategy S for the black/white pebble game on T that uses at most k pebbles. During step i of this strategy the goal is to add a black pebble or to remove a white pebble from vertex $\sigma^{-1}(i)$. To accomplish this, white pebbles are added to all unpebbled predecessors of $\sigma^{-1}(i)$, a black pebble is placed on $\sigma^{-1}(i)$ or a white pebble is removed, and then black pebbles are removed from all vertices $\sigma^{-1}(j)$, $j \leq i$ which are not predecessors of some vertex $\sigma^{-1}(h)$, $h > i$. By induction on i it can be shown that the number of pebbles on the vertices of T at each step i is not greater than the sum of the number of edges passing over $\sigma^{-1}(i)$ in the layout and the number of predecessors of $\sigma^{-1}(i)$. Using the fact that every vertex in T has at most two predecessors and one successor and the fact that any vertex with

two predecessors has a successor, this sum can be shown to be bounded by k . As this is true at every step, the black/white pebble demand of T is bounded by its cutwidth. (Note: it follows by a similar argument that if T is a directed tree with in degree $\leq d$ in which the sink has degree one, then the black/white pebble demand of T is not greater than $\gamma(T) + \lfloor (d-1)/2 \rfloor$. Thus it follows that the black/white pebble demand of such a tree is between $\gamma(T)$ and $\gamma(T) + \lfloor (d-1)/2 \rfloor$. So, the algorithm we present for determining the cutwidth of a tree also gives approximate information about the black/white pebble demand for arbitrary degree bounded trees.)

Previous work on the black/white pebble demand of trees has appeared in [16], [17], [21], [26]. Our MIN CUT algorithm gives an $O(n \ln n)$ algorithm for determining the black/white pebble demand of binary trees.

1.2. VLSI layout. A central problem in VLSI is area efficient embeddings of various graphs in the plane. There are methodologies for automated component placement that suggest placing circuit elements in rows or along a single line [5], [6], [7], [20], [27], [33], [40], [43]. For example, Dolev and Trickey [5] have such a strategy in mind when they consider the MIN CUT LINEAR ARRANGEMENT problem for trees with the additional restriction that the edges are not allowed to cross. (They give an $O(n \log n)$ algorithm for this planar layout version of MIN CUT on trees.) Foster and Kung [7] consider the construction of VLSI circuits for regular languages with programmable building blocks. The circuits form degree three trees and, when automatic construction is desired from a given regular expression, one assigns the active elements to positions along the bottom row of a "programmable recognizer array" (PRA); the connections using at most $\log n$ tracks above. To minimize the number of tracks (and hence the area) for such circuits one positions the basic cells along the bottom row of the PRA in such a way that cutwidth is minimized.

Another important application is the layout of logic circuits using Weinberger gate arrays [1], [41], [43]. This technique is used in several experimental silicon compilers.

We describe an algorithm which solves the MIN CUT problem for trees. The algorithm obtains the optimum cutwidth or folding number for an arbitrary tree. In addition, for any fixed $d \geq 3$, the algorithm takes at most $O(n(\log n)^{d-2})$ steps to determine the cutwidth of a degree d tree with n vertices. We observe that the degree of the polynomial time bound grows with the degree of the tree. (Recently, Yannakakis [42] has also found an $O(n \log n)$ algorithm for all trees.)

We also give an algorithm that not only determines the cutwidth but produces an optimal layout as well.

It should, perhaps, be noted that a layout to minimize cutwidth is not, in general, the same as a layout to minimize the sum of all the edge lengths (the latter being the goal of the OPTIMAL LINEAR ARRANGEMENT problem). For example, in Fig. 1, the first layout is among the best for OPTIMAL LINEAR ARRANGEMENT, but is not optimal for cutwidth, and the second layout is among the best for cutwidth, but is not a good layout for OPTIMAL LINEAR ARRANGEMENT. There are several results concerning the OPTIMAL LINEAR ARRANGEMENT problem on trees in the literature [2], [12], [15], [38]. The best result currently is due to F. R. K. Chung [2] and gives an $O(n^{1.58})$ algorithm. Optimal layouts for cutwidth are quite obviously, in general, not good layouts for bandwidth. It is known that the BANDWIDTH MINIMIZATION problem, even for degree three trees, is NP-complete [9].

In § 2 we give a general characterization of cutwidth k trees. For the special case of degree three trees we give a specific sequence of tree families with the property: a

tree has cutwidth at most k if and only if it does not contain a homeomorphic image of a tree in the $(k + 1)$ st family. In fact, the same result was obtained by Parsons [31], [32] for the notion of search number. Thus, we obtain the somewhat surprising result that, for the class of trees with degree three, a tree has cutwidth k if and only if it has search number k . (The search number of a graph is defined in [25]. In fact, a more recent result shows that search number and cutwidth are the same for all degree three graphs [22]. This uses the fact that “recontamination” does not reduce search number [14].) In § 3 we give an $O(n(\log n)^{d-1})$ algorithm for determining the cutwidth of any degree d tree. In § 4 we give a related algorithm for determining the cutwidth of a degree three tree in time $O(n \log n)$. This algorithm is then used to speed up the degree d algorithm to $O(n(\log n)^{d-2})$. The decision algorithms described in §§ 3 and 4 yield information that can be used to produce an optimal layout of the tree. An algorithm to construct the layout is described in § 5. We conclude with a list of open problems.

2. Characterization of trees with cutwidth k . Let $T = (V, E)$ be a tree and let $\{u, x_1, \dots, x_r\} \subseteq V$. Define $T(u, x_1, \dots, x_r)$ as the largest subtree of T that contains u but does not contain any of x_1, \dots, x_r . This definition is illustrated in Fig. 2. Let σ be a layout of T . The vertex which is mapped to 1 by σ is referred to as the *leftmost vertex* in the layout. The vertex which is mapped to $|V|$ by σ is referred to as the *rightmost vertex*.

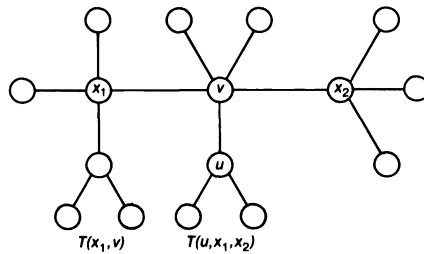


FIG. 2. Notation for undirected tree.

THEOREM 2.1 (general characterization theorem). *Let T be an undirected tree. $\gamma(T) \leq k \Leftrightarrow$ every vertex u of degree at least two has neighbors x_1, x_2 such that $\gamma(T(u, x_1, x_2)) \leq k - 1$.*

It follows from Theorem 2.1 that the tree in Fig. 3 has cutwidth four since vertex u does not have neighbors x_1, x_2 such that $\gamma(T(u, x_1, x_2)) \leq 2$.

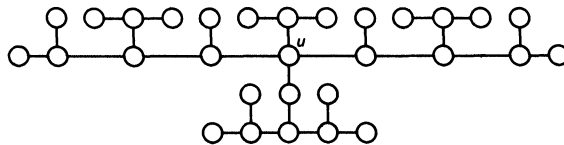


FIG. 3. Application of Theorem 2.1.

Proof. (\Rightarrow) Assume to the contrary that $\gamma(T) \leq k$ and there exists a vertex u of degree ≥ 2 such that for every pair of vertices x_1, x_2 adjacent to u , $\gamma(T(u, x_1, x_2)) \geq k$. Now let σ be a layout of T such that $\gamma_\sigma(T) \leq k$ and let P be the path connecting the leftmost and rightmost vertices of T under σ . If u is an internal vertex of P let x_1, x_2 be the neighbors of u on P . As shown in Fig. 4(a), P passes entirely over $T(u, x_1, x_2)$

in the layout but since $\gamma(T(u, x_1, x_2)) \geq k$ it follows that the cutwidth of the layout exceeds k , contradicting the assumption. If u is an endpoint of P then let x be the neighbor of u on P . In this case P passes over $T(u, x)$ and again we obtain a contradiction since $\gamma(T(u, x)) \geq k$. Finally if u is not on P , let x be the neighbor of u on the path from u to P . Once again, P passes over $T(u, x)$ yielding a contradiction.

(\Leftarrow) Consider two cases. First suppose that every vertex u has a neighbor x such that $\gamma(T(u, x)) \leq k - 1$. Starting from any vertex y_1 , construct a path y_1, \dots, y_r , where $\gamma(T(y_i, y_{i+1})) \leq k - 1$ for $1 \leq i < r$ and $\gamma(T(y_r, y_{r-1})) \leq k - 1$. This construction is shown in Fig. 4(b). It is clear that $\gamma(T) \leq k$.

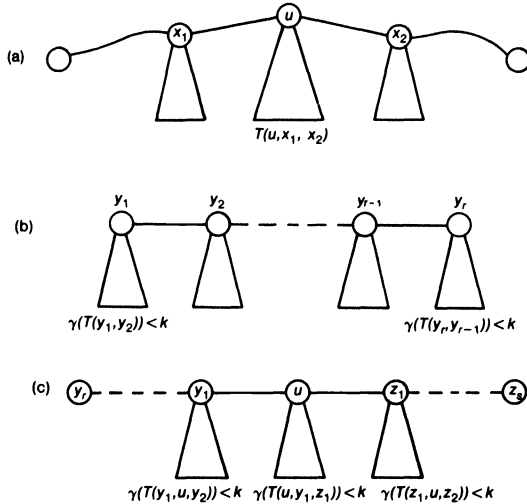


FIG. 4. Illustrations for Theorem 2.1.

Next suppose that there is some vertex u such that for all neighbors x of u , $\gamma(T(u, x)) \geq k$. By the hypothesis, u has neighbors y_1, z_1 such that $\gamma(T(u, y_1, z_1)) \leq k - 1$. Now if y_1 is not a leaf then it has a neighbor y_2 such that $\gamma(T(y_1, u, y_2)) \leq k - 1$. Similarly if y_2 is not a leaf then it has a neighbor y_3 such that $\gamma(T(y_2, y_1, y_3)) \leq k - 1$. Continuing in this fashion one can construct a path u, y_1, \dots, y_r such that for $1 \leq i \leq r - 2$, $\gamma(T(y_{i+1}, y_i, y_{i+2})) \leq k - 1$, and y_r is a leaf. One can construct a similar path u, z_1, \dots, z_s . This construction is illustrated in Fig. 4(c). Again it is clear that $\gamma(T) \leq k$. \square

Let $T_d(k)$ denote the set of smallest trees with degree d and cutwidth k .

COROLLARY 2.1. $T_3(1)$ is the singleton set containing the tree with two vertices. For $k > 1$ each tree in $T_3(k)$ can be formed by identifying a leaf in three (not necessarily distinct) trees from $T_3(k - 1)$.

This construction is illustrated in Fig. 5 for $T_3(2)$, $T_3(3)$, and $T_3(4)$.

Proof. The proof is by induction. The basis, $k = 1$ is immediate. Assume then that $k > 1$ and let T be any tree in $T_3(k)$. By Theorem 2.1, T must contain a vertex u with neighbors x_1, x_2, x_3 such that $\gamma(T(u, x_1, x_2)) \geq k - 1$, $\gamma(T(u, x_1, x_3)) \geq k - 1$ and $\gamma(T(u, x_2, x_3)) \geq k - 1$. Since T is a smallest degree three tree with cutwidth k , it follows that $T(u, x_1, x_2)$, $T(u, x_1, x_3)$ and $T(u, x_2, x_3)$ must be smallest degree three trees with cutwidth $k - 1$, that is they must be in $T_3(k - 1)$. \square

Let $n_d(k)$ be the number of vertices in a smallest degree d tree with cutwidth k .

COROLLARY 2.2. $n_3(k) = 3^{k-1} + 1$.

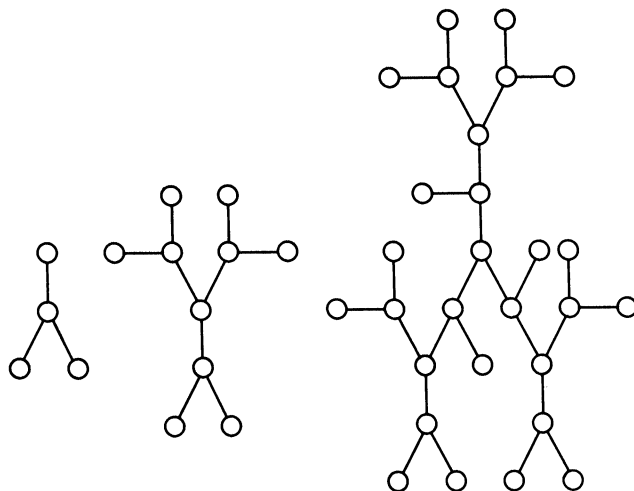


FIG. 5. $T_3(2)$, $T_3(3)$, and $T_3(4)$.

Proof. By induction. The basis $k = 1$ is immediate from Corollary 2.1. Assume then that $n_3(k - 1) = 3^{k-2} + 1$. By Corollary 2.1,

$$n_3(k) = 3n_3(k - 1) - 2 = 3(3^{k-2} + 1) - 2 = 3^{k-1} + 1. \quad \square$$

The next corollary relates the cutwidth of a tree to its search number. To understand the notion of search number, let $G = (V, E)$ be a graph and think of the vertices of G as rooms, and the edges as interconnecting corridors. Now, assume that there is an escaped convict lurking somewhere within G . Your job is to organize a search party to capture the fugitive and since you have limited resources you want to do it with the fewest possible number of searchers. The *search number* of G is the minimum number of searchers required to guarantee that the fugitive is captured. The following result follows immediately from Theorem 2.1 and a characterization of trees with search number k given by Parsons [31]. (We are indebted to S. L. Hakimi for pointing out Parson's result.)

COROLLARY 2.3. *A degree three tree has cutwidth $k \Leftrightarrow$ it has search number k .*

The next theorem provides a forbidden subgraph characterization of degree three trees with cutwidth k . First we require some definitions. Let $f(T)$ be the set of trees obtained from T by replacing a single edge $\{u, v\}$ with the tree shown in Fig. 6, where x and y are new vertices and neither u nor v is adjacent to a leaf of T . If S is a set of trees, $f(S)$ is the union of the sets $f(T)$ for all T in S . Let $L_3(1)$ be the singleton set containing the tree on two vertices. $M_3(k)$ is the union of $L_3(k)$ and $f(L_3(k))$. $L_3(k + 1)$ is the set of trees that are obtained by identifying a leaf in three (not necessarily distinct) trees from $M_3(k)$. For $k \leq 5$, $L_3(k) = T_3(k)$.

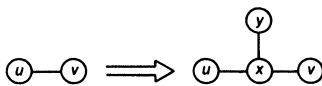


FIG. 6. Definition of $f(T)$.

THEOREM 2.2. *Let T be a degree three tree. $\gamma(T) = k \Leftrightarrow T$ contains a homeomorphic image of a tree in $L_3(k)$ and does not contain a homeomorphic image of any tree in $L_3(k + 1)$.*

Proof. By induction. The basis, $k = 1$, is obvious. By Theorem 2.1, T has some vertex u with neighbors x_1, x_2, x_3 such that $\gamma(T(u, x_1, x_2)) \geq k - 1$, $\gamma(T(u, x_1, x_3)) \geq k - 1$ and $\gamma(T(u, x_2, x_3)) \geq k - 1$. By the induction hypothesis each of these subtrees contains a homeomorphic image of a tree in $L_3(k - 1)$. Consider the subtree consisting of these three images together with the paths that join them to u . This is a homeomorphic image of a tree in $L_3(k)$. Since each tree in $L_3(k + 1)$ has cutwidth $k + 1$, T cannot contain a homeomorphic image of any tree in $L_3(k + 1)$. \square

Theorem 2.2 can be generalized to give a somewhat more complex forbidden subgraph characterization of degree d trees.

The next theorem relates the number of vertices in a degree bounded tree to its cutwidth. This will be used in the complexity analysis of our cutwidth minimization algorithm.

THEOREM 2.3. *Let T be a degree d tree with cutwidth k . T contains at least $(d/(d - 2))^{k-1} + 1$ vertices.*

Proof. By Theorem 2.1 T contains a vertex u with neighbors x_1, \dots, x_r ($r \leq d$) such that $\gamma(T(u, x_i, x_j)) \geq k - 1$ for $1 \leq i < j \leq r$. Now select x_i, x_j so that $|T(x_i, u)| \geq |T(x_j, u)|$ for $1 \leq h \leq r, i \neq h \neq j$. If $|T| = n$ and $|T(u, x_i, x_j)| = m$ then $(m - 1) \leq ((r - 2)/r)(n - 1)$ or $n \geq (d/(d - 2))(m - 1) + 1$, since $r/(r - 2) \geq d/(d - 2)$. Since this holds for all trees T and since $m \geq n_d(k - 1)$ it follows that

$$n_d(k) \geq \frac{d}{d-2}(n_d(k-1) - 1) + 1 \geq \left(\frac{d}{d-2}\right)^{k-1} (n_d(1) - 1) + 1 = \left(\frac{d}{d-2}\right)^{k-1} + 1. \quad \square$$

COROLLARY 2.4. *Let T be a degree d tree with n vertices. $\gamma(T) < (d/2) \ln n + 1$.*

Proof. Let $k = \gamma(T)$. By the theorem

$$n > (d/(d - 2))^{k-1}, \quad k - 1 < \frac{\ln n}{\ln(d/(d - 2))} < \frac{d}{2} \ln n. \quad \square$$

3. A cutwidth minimization algorithm for trees. This section describes a general algorithm for the cutwidth minimization problem on trees. The time bound for the algorithm is

$$O\left(\binom{k+d-1}{d-1} n \log n\right),$$

where n is the number of vertices in the tree, k is its cutwidth and d is the maximum vertex degree. For trees with fixed maximum degree this quantity is $O(n(\ln n)^{d-1})$.

In the remainder of this section we assume that all trees are directed and rooted. This is strictly for notational convenience. The cutwidth of a directed tree is the same as the cutwidth of the underlying undirected tree.

If T is a tree, $T[u]$ denotes the induced subtree with root u . $T[u, x_1, \dots, x_r] = T[u] - \cup_{i=1}^r T[x_i]$. These definitions are illustrated in Fig. 7.

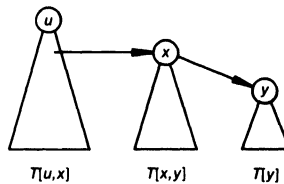


FIG. 7. Notation for directed trees.

Let T be a tree with root u and children v_1, \dots, v_d of u . Define $\delta(T) = \min_{1 \leq i \leq d} \max \{ \gamma(T[v_i]), \gamma(T[u, v_i]) \}$ when $d \geq 1$ and, $\delta(T) = 0$ when $d = 0$. Note that $\delta(T) \leq \gamma(T) \leq \delta(T) + 1$, since T is formed by joining $T[v_i]$ and $T[u, v_i]$ together with an edge, for every i .

Let x be a vertex in a tree T and let k be a positive integer. We say that x is k -critical if $k = \delta(T[x])$ and for all children y of x , $\gamma(T[x, y]) \geq k$.

The next theorem is essentially a restatement of Theorem 2.1. We will find this form more convenient in what follows.

THEOREM 3.1. *Let T be a tree with root u and let $k = \delta(T)$. $\gamma(T) = k \Leftrightarrow T$ has no k -critical vertex or T has exactly one k -critical vertex x and x has children y, z such that $\gamma(T[u, y, z]) < k$.*

Proof. (\Rightarrow) Let T' be the underlying undirected tree. If T has two k -critical vertices v, x then one can show that at least one of them, say x , does not have neighbors y, z such that $\gamma(T'(x, y, z)) < k$, contradicting Theorem 2.1. Similarly, if T has a k -critical vertex x with no children y, z such that $\gamma(T[u, y, z]) < k$, then x has no neighbors y, z such that $\gamma(T'(x, y, z)) < k$, contradicting Theorem 2.1. (Note $T'(x, y, z)$ is the underlying tree for $T[u, y, z]$.)

(\Leftarrow) If T has no k -critical vertex then every vertex x in T' has neighbors y, z such that $\gamma(T'(x, y, z)) < k$, and by Theorem 2.1 $\gamma(T) \leq k$. Since $\delta(T) = k$, $\gamma(T) = k$. If T has exactly one k -critical vertex x that satisfies the condition stated, then Theorem 2.1 applies and again $\gamma(T) = k$. \square

Using Theorem 3.1 we can compute the cutwidth of small trees by hand. We will illustrate this procedure with an example before giving the formal presentation of the algorithm. Given a tree T , we work from the bottom up assigning labels to each of the vertices in the tree. These labels consist of a decreasing sequence of integers; the largest integer in the sequence is the cutwidth of the subtree whose root is the associated vertex. Consider the tree T_1 shown in Fig. 8(a). The label next to vertex b means that the cutwidth of the subtree containing just vertex b is 0. Given the labels on b and c we want to use Theorem 3.1 to determine the cutwidth of T_1 . The first step is to determine $\delta(T_1)$. In this case, one can see that $\delta(T_1) = 1$, hence the cutwidth of T_1 is either 1 or 2. The next step is to determine if T_1 contains a 1-critical vertex. In fact, a is 1-critical, so the next step is to determine if a satisfies the condition given in the theorem. In this case the answer is yes, since $\gamma(T_1[a, b, c]) = 0 < 1$. Thus, according to the theorem, $\gamma(T_1) = 1$, which is clearly true. The label next to vertex a in the figure is $[1, 0]$. The meaning of this label is that (1) the cutwidth of $T_1[a]$ is 1 and (2) $T_1[a]$ contains a 1-critical vertex with children b, c such that the cutwidth of $T_1[a, b, c]$ is 0.

Now consider the tree T_2 shown in Fig. 8(b). Using the result for T_1 , one can show that $\delta(T_2) = 1$. Thus, we want to determine if T_2 contains a 1-critical vertex. In fact d is 1-critical, so the next step is to determine if d has children x, y such that

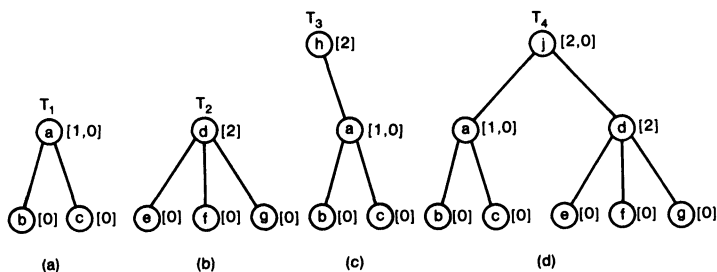


FIG. 8. Example to illustrate cutwidth computation.

$\gamma(T_2[d, x, y]) < 1$. Since it does not, we conclude from Theorem 3.1 that $\gamma(T_2) = 2$. The label next to vertex d means that the cutwidth of $T_2[d]$ is 2 and $T_2[d]$ contains no 2-critical vertex. A similar argument can be used to determine the cutwidth of T_3 in Fig. 8(c).

Now consider T_4 in Fig. 8(d). Using the results of the previous examples we can show that $\delta(T_4) = 2$. We can also see that j is a 2-critical vertex. Since j has children a, d such that $\gamma(T_4[j, a, d]) = 0 < 2$ it follows from Theorem 3.1 that the cutwidth of $T_4 = 2$. Furthermore, the label for vertex j is $[2, 0]$.

We now proceed with the formal presentation of the algorithm. We will use the usual lexicographic ordering on decreasing sequences of integers. $[a_1, \dots, a_r] < [b_1, \dots, b_s]$ if (1) for some i ($1 \leq i \leq \min\{r, s\}$), $a_i < b_i$ and for all j ($1 \leq j < i$), $a_j = b_j$ or (2) $r < s$ and for $1 \leq j \leq r$, $a_j = b_j$. We will also apply set operations to such sequences with the obvious interpretation.

Let T be a tree with root u . We define $\Gamma(T)$ recursively as follows:

- If $\gamma(T) = 0$ then $\Gamma(T) = [0]$.
- If $\gamma(T) = k$ and T contains no k -critical vertex then $\Gamma(T) = [k]$.
- If $\gamma(T) = k$ and T contains a k -critical vertex x then,

$$\Gamma(T) = [k] \cup \min_{y,z} \Gamma(T[u, y, z])$$

where y, z range over all children of x .

We can restate this definition in iterative form as follows. $\Gamma(T) = [a_1, \dots, a_r]$ if $a_1 > \dots > a_r \geq 0$ and T contains vertices x_1, \dots, x_{r-1} where x_i has children y_i, z_i such that

1. For $1 \leq i \leq r$, $\gamma(T[u, y_1, z_1, \dots, y_{i-1}, z_{i-1}]) = a_i$.
2. For $1 \leq i < r$, x_i is an a_i -critical vertex in $T[u, y_1, z_1, \dots, y_{i-1}, z_{i-1}]$.
3. $T[u, y_1, z_1, \dots, y_{r-1}, z_{r-1}]$ contains no a_r -critical vertex.
4. There is no sequence $[b_1, \dots, b_s] < [a_1, \dots, a_r]$ that also satisfies conditions 1-3.

Let T be a tree with root u having children v_1, \dots, v_d . In Fig. 9, an algorithm called Γ is described which computes $\Gamma(T)$ from $\Gamma(T[v_1]), \dots, \Gamma(T[v_d])$. By applying Γ recursively we can attach a label $\lambda(x) = \Gamma(T[x])$, to each vertex x . A procedure for computing these labels is shown in Fig. 10. An example of a tree with the labels attached to the vertices is shown in Fig. 11. Once the labels have been computed finding an optimal layout is straightforward. The layout algorithm is described in § 5.

```

[1] procedure Gamma( $S_1, \dots, S_d$ )
[2]   { Relabel if necessary so that  $S_1 \geq \dots \geq S_d$  }
[3]   if  $d = 0$  then return [0]
[4]   if  $d = 1$  then begin
[5]     if  $\min S_1 \neq 0$  then return  $S_1$ 
[6]      $y \leftarrow \min \{x > 0 \mid x \in S_1\}$ 
[7]     return  $\{y\} \cup \{x > y \mid x \in S_1\}$ 
[8]   end
[9]   {  $d \geq 2$  }
[10]   $\bar{S}_1 \leftarrow \Gamma(S_1)$ 
[11]   $\bar{S}_2 \leftarrow \Gamma(S_2, \dots, S_d)$ 
[12]   $k \leftarrow \max \{S_1 \cup \bar{S}_1\}$ 
[13]  {  $k = \delta(T)$  }
[14]  if  $k = \max S_2$  then return  $[k+1]$ 
[15]  if  $k \neq \max S_1$  or  $k = \min S_1$  then begin
[16]    if  $k = \max S_1$  then return  $[k] \cup S_2$ 
[17]    else return  $[k]$ 
[18]  end
[19]  {  $T[v_1]$  contains a  $k$ -critical vertex }
[20]  if  $k = \max S_1$  then return  $[k+1]$ 
[21]   $H \leftarrow \Gamma(S_1 - [k], S_2, \dots, S_d)$ 
[22]  if  $k = \max H$  then return  $[k+1]$ 
[23]  else return  $[k] \cup H$ 
[24] end
    
```

FIG. 9. Recursive algorithm for computing $\Gamma(T)$ from $\Gamma(T[u, v_1]), \dots, \Gamma(T[u, v_d])$.

```

[1] procedure Label( $T, x$ )
[2]   Let  $v_1, \dots, v_d$  be the children of  $x$ .
[3]   Let  $S_i = \text{Label}(T[v_i], v_i)$  for  $1 \leq i \leq d$ .
[4]    $\lambda(x) = \text{Gamma}(S_1, \dots, S_d)$ 
[5]   return  $\lambda(x)$ 
[6] end
    
```

FIG. 10. Vertex labeling procedure.

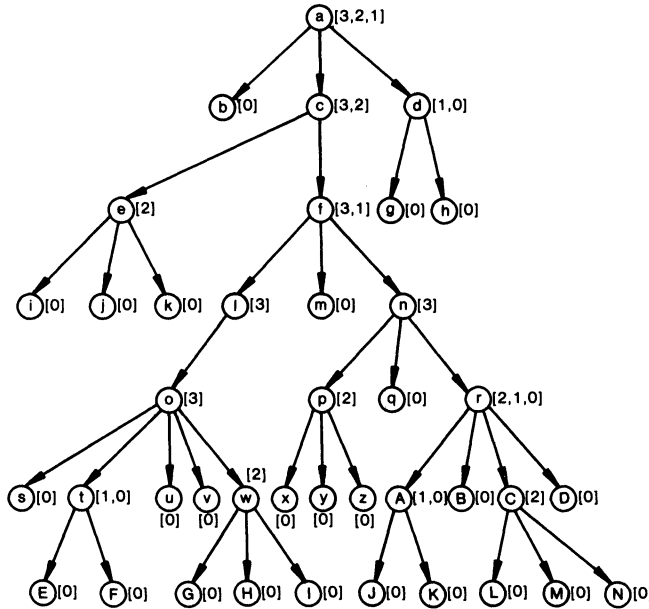


FIG. 11. Example of vertex labeling produced by $\text{Gamma}(\cdot)$.

The following theorem establishes the correctness of the algorithm.

THEOREM 3.2. Let T be a tree with root u having children v_1, \dots, v_d and let $S_i = \Gamma(T[v_i])$, for all i ($1 \leq i \leq d$). $\text{Gamma}(S_1, \dots, S_d) = \Gamma(T)$.

The proof of Theorem 3.2 requires a technical result given in Theorem 3.3. Let T_1 and T_2 be trees. The notation $T_1 : T_2$ denotes the tree obtained by making T_2 a subtree of the root of T_1 . This operation is illustrated in Fig. 12.

THEOREM 3.3. Let R, S, S' be trees with roots u, v, v' and let $T = R \cdot S, T' = R \cdot S'$. $\Gamma(S) \leq \Gamma(S') \Rightarrow \Gamma(T) \leq \Gamma(T')$.

The situation described in Theorem 3.3 is shown in Fig. 13. The proof is given in the appendix.

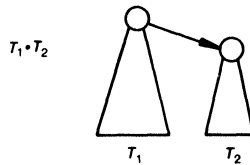


FIG. 12. Definition of $T \cdot T_2$.

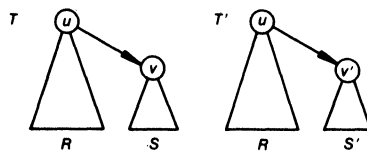


FIG. 13. Illustration for Theorem 3.3.

The following corollaries to Theorem 3.3 are used in the proof of Theorem 3.2.

Let $T_1 \cdot T_2 \cdots T_r$ denote $(\cdots ((T_1 \cdot T_2) \cdot T_3) \cdots) \cdot T_r$.

COROLLARY 3.1. *Let $R, S_1, \dots, S_r, S'_1, \dots, S'_r$ be trees. If $\Gamma(S_i) \cong \Gamma(S'_i)$ for $1 \leq i \leq r$ then $\Gamma(R \cdot S_1 \cdots S_r) \cong \Gamma(R \cdot S'_1 \cdots S'_r)$.*

Proof. By successive applications of Theorem 3.3,

$$\Gamma(R \cdot S_1 \cdots S_r) \cong \Gamma(R \cdot S'_1 \cdot S_2 \cdots S_r) \cong \cdots \cong \Gamma(R \cdot S'_1 \cdots S'_r). \quad \square$$

Let R, S be trees and let u be a vertex in R . Let $R \cdot S|_u$ denote the tree formed by making the root of S a child of u as shown in Fig. 14.

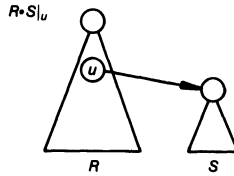


FIG. 14. $R \cdot S$.

COROLLARY 3.2. *Let R, S, S' be trees and let u be a vertex in R and let $T = R \cdot S|_u$, $T' = R \cdot S'|_u$. $\Gamma(S) \cong \Gamma(S') \Rightarrow \Gamma(T) \cong \Gamma(T')$.*

Proof. Let v be the root of R . The proof is by induction on the length of the path from v to u . (Basis) Assume $v = u$. In this case the statement reduces to Theorem 3.3. (Induction) If $v \neq u$ then let w be the child of v that is on the path from v to u . By induction we can assume that $\Gamma(T[w]) \cong \Gamma(T'[w])$. We can then apply Theorem 3.3 with $R[v, w]$, $T[w]$ and $T'[w]$ and the result follows. \square

COROLLARY 3.3. *Let T be a tree with root u having children v_1, \dots, v_a , $d \geq 1$, such that $\Gamma(T[v_1]) \cong \cdots \cong \Gamma(T[v_a])$. $\delta(T) = \max \{ \gamma(T[v_1]), \gamma(T[u, v_1]) \}$.*

Proof. The result is trivially true for $d = 1$. Assume then that $1 < i \leq d$. Since $T[v_1]$ is a subtree of $T[u, v_i]$, $\gamma(T[v_1]) \leq \gamma(T[u, v_i])$. We can now apply Theorem 3.3 (with $R = T[u, v_1, v_i]$, $S = T[v_i]$ and $S' = T[v_1]$), yielding $\gamma(T[u, v_1]) \leq \gamma(T[u, v_i])$. Hence $\max \{ \gamma(T[v_1]), \gamma(T[u, v_1]) \} \leq \max \{ \gamma(T[v_i]), \gamma(T[u, v_i]) \}$. \square

COROLLARY 3.4. *Let T be a tree with $\delta(T) = k$ and let u be a vertex with child x such that for all children y of u , $\Gamma(T[x]) \cong \Gamma(T[y])$. Then u is k -critical $\Leftrightarrow \gamma(T[u, x]) \geq k$.*

Proof. The forward implication is immediate. For the converse, let y be any child of u , let $S = T[y]$, $S' = T[x]$, $R = T[u, x, y]$ and note that $R \cdot S = T[u, x]$ and $R \cdot S' = T[u, y]$. By Corollary 3.2, $\gamma(T[u, x]) \leq \gamma(T[u, y])$. \square

COROLLARY 3.5. *Let T be a tree with root u , let $\gamma(T) = k$ and let x be a k -critical vertex. If y, z are children of x such that for all children w of x ($w \neq y$), $\Gamma(T[y]) \cong \Gamma(T[z]) \cong \Gamma(T[w])$ then $\gamma(T[u, y, z]) < k$ and $\Gamma(T) = [k] \cup \Gamma(T[u, y, z])$.*

Proof. By definition of $\Gamma(T)$, x has children v, w such that $\Gamma(T) = [k] \cup \Gamma(T[u, v, w])$. By Corollaries 3.1 and 3.2, $\Gamma(T[u, y, z]) \cong \Gamma(T[u, v, w])$. Hence $\Gamma(T) = [k] \cup \Gamma(T[u, y, z])$. \square

Proof of Theorem 3.2. Let T be a tree with root u having children v_1, \dots, v_a , let $S_i = \Gamma(T[v_i])$, for all i ($1 \leq i \leq d$), and let $S_1 \cong \cdots \cong S_d$. We want to show that $\text{Gamma}(S_1, \dots, S_d) = \Gamma(T)$. We first prove the correctness for the special cases ($d \leq 1$) and ($d \geq 2 \wedge \delta(T) = 1$). When $d \leq 1$, $\text{Gamma}(\cdot)$ is given by one of lines [3], [5] or [7].

line [3]. $d = 0$. This means that T consists of a single vertex and by definition $\Gamma(T) = [0]$.

line [5]. $d = 1 \wedge \min S_1 \neq 0$. Let $S_1 = \Gamma(T[v_1]) = [a_1, \dots, a_r]$ and let x_i, y_i, z_i ($1 \leq i < r$) be the vertices referred to in the definition of $\Gamma(\cdot)$. Let $H =$

$T[u, y_1, z_1, \dots, y_{r-1}, z_{r-1}]$ and note that $\delta(H) = a_r$. Since $a_r > 0$, H has no a_r -critical vertex, and by Theorem 3.1 $\gamma(H) = a_r$. Now, let $J = T[u, y_1, z_1, \dots, y_{r-2}, z_{r-2}]$ and note that $\delta(J) = a_{r-1}$. Since $a_{r-1} > 0$, the only possible a_{r-1} -critical vertex in J is x_{r-1} , and since $\gamma(H) = a_r < a_{r-1}$, $\gamma(J) = a_{r-1}$ by Theorem 3.1. Continuing in this fashion yields $\Gamma(T) = \Gamma(T[v_1])$.

line [7]. $d = 1 \wedge \min S_1 = 0$. Let $S_1 = \Gamma(T[v_1]) = [a_1, \dots, a_r]$ and let x_i, y_i, z_i ($1 \leq i < r$) be the vertices referred to in the definition of $\Gamma(\cdot)$. Let $w = \min \{x > 0 \mid x \notin S_1\}$ and let $a_i = w - 1$. Since $a_r = 0$, $T[v_1, y_1, z_1, \dots, y_{r-1}, z_{r-1}]$ consists of a single vertex. Obviously $\gamma(T[u, y_1, z_1, \dots, y_{r-1}, z_{r-1}]) = 1$. For all j ($i \leq j \leq r$), $\gamma(T[u, y_1, z_1, \dots, y_{j-1}, z_{j-1}]) = a_{j-1} + 1$, by repeated applications of Theorem 3.1. Since $w > 0$ and $w \notin S_1$, $T[u, y_1, z_1, \dots, y_{i-1}, z_{i-1}]$ has no w -critical vertex. For $1 \leq j < i$, x_j is an a_j -critical vertex in $T[u, y_1, z_1, \dots, y_{j-1}, z_{j-1}]$. Thus $\Gamma(T) = [a_1, \dots, a_{i-1}, w]$ as claimed.

When ($d \geq 2 \wedge \delta(T) = 1$) T is a subtree of the tree shown in Fig. 15. If $d = 2$ then the values of \bar{S}_1 and \bar{S}_2 computed in lines [10] and [11] are correct, since we have established the correctness of *Gamma* (\cdot) for $d < 2$. In particular $\bar{S}_1 = [1]$, $\bar{S}_2 = [0]$ and hence the value of k computed in line [12] is 1. If $\Gamma(T[v_1]) = [1]$ then v_1 has one child and *Gamma* will return [1, 0] at line [16] which is correct. If $\Gamma(T[v_1]) = [1, 0]$ then v_1 has two children and *Gamma* will return [2] at line [20] which is correct. Thus *Gamma* is correct if $d = 2$ and $\delta(T) = 1$. Consequently, when $d = 3$ and $\delta(T) = 1$, the values of \bar{S}_1 and \bar{S}_2 computed in lines [10] and [11] are correct. In particular $\bar{S}_1 = [1, 0]$ and $\bar{S}_2 = [1]$. In this situation, *Gamma* returns [2] at line [14] which is correct.

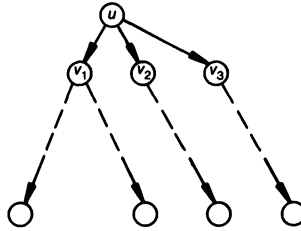


FIG. 15. Special case of Theorem 3.2— $d \geq 2 \wedge \delta(T) = 1$.

The proof now proceeds by induction on d and $\delta(T)$. Assume that *Gamma* is correct for all trees T' in which either the root has fewer than d children or the root has d children but $\delta(T) < k$. Thus, the values of \bar{S}_1 and \bar{S}_2 computed in lines [10] and [11] are equal to $\Gamma(T[u, v_1])$ and $\Gamma(T[u, v_1, v_2])$ respectively, and the value of k computed in line [12] is equal to $\delta(T)$ by Corollary 3.3. Now considering each of the return statements in lines [14], [16], [17], [20].

line [14]. $d \geq 2 \wedge k = d(T) \wedge k = \gamma(T[u, v_1, v_2])$. Since $k = \gamma(T[u, v_1, v_2])$, $k = \gamma(T[u, v_1])$ and by Corollary 3.4 u is k -critical. By Theorem 3.1, $\gamma(T) = k + 1$ and since T contains no $(k + 1)$ -critical vertex, $\Gamma(T) = [k + 1]$.

line [16]. $d \geq 2 \wedge k = \delta(T) \wedge k > \gamma(T[u, v_1, v_2]) \wedge T[v_1]$ does not contain a k -critical vertex $\wedge u$ is k -critical. By Theorem 3.1, $\gamma(T) = k$, and by Corollary 3.5, $\Gamma(T) = [k] \cup \Gamma(T[u, v_1, v_2])$.

line [17]. $d \geq 2 \wedge k = \delta(T) \wedge k > \gamma(T[u, v_1, v_2]) \wedge T$ does not contain a k -critical vertex. By Theorem 3.1, $\gamma(T) = k$, and since there is no k -critical vertex, $\Gamma(T) = [k]$.

line [20]. $d \geq 2 \wedge k = \delta(T) \wedge k > \gamma(T[u, v_1, v_2]) \wedge T[v_1]$ contains a k -critical vertex $\wedge u$ is k -critical. By Theorem 3.1, $\gamma(T) = k + 1$, and since T contains no $(k + 1)$ -critical vertex, $\Gamma(T) = [k + 1]$.

At line [21], we have $d \geq 2 \wedge \Gamma(T[u, v_1]) < k \wedge T[v_1]$ contains a k -critical vertex. Let x be the k -critical vertex in $T[v_1]$ and let y, z be children of x such that $\Gamma(T[v_1, y, z]) = S_1 - [k]$. Since $\delta(T[u, y, z]) < k$, the value of H computed in line 21 is $\Gamma(T[u, y, z])$ by the induction hypothesis. Considering the return statements in lines [22] and [23].

line [22]. $\Gamma(T[u, y, z]) = k$. By Theorem 3.1, $\gamma(T) = [k + 1]$, and since T contains no $(k + 1)$ -critical vertex, $\Gamma(T) = [k + 1]$.

line [23]. $\Gamma(T[u, y, z]) < k$. By Theorem 3.1, $\gamma(T) = [k]$ and by Corollary 3.5, $\Gamma(T) = [k] \cup \Gamma(T[u, y, z])$.

Thus, we have shown that the procedure *Gamma* returns the correct value, $\Gamma(T[u])$. \square

The time required to execute *Label* (\cdot) is proportional to n times the time required to execute *Gamma* (\cdot). Excluding the recursive calls to *Gamma* at lines [10], [11] and [21], the time required to execute *Gamma* is $O(d \log n)$. (This follows from Corollary 2.4.) Note that the recursive call at line [11] can be ignored since the computation performed there is actually a subset of the computation made at line [10]. Consequently we can express the number of recursive calls required to compute *Gamma* (S_1, \dots, S_d) using the recurrence $M(k, d) \leq M(k, d - 1) + M(k - 1, d)$ where $k = \max S_1 \cup \dots \cup S_d$. This follows from the observation that in line [10] the number of subtrees is reduced by one and in line [21] the $\max S_1 \cup \dots \cup S_d$ is reduced by at least one. Of course this is the defining recurrence for the binomial coefficients. Using the boundary conditions $M(k, 1) = M(0, d) = 1$ yields

$$M(k, d) \leq \binom{k + d - 1}{d - 1}.$$

Using Corollary 2.4 we can show that for fixed d , $M(k, d) \leq O((\log n)^{d-1})$. This means that the time complexity of *Gamma* is $O((\log n)^d)$. If we select a degree one vertex as the root of T then $d \leq D - 1$ where D is the maximum vertex degree for the entire tree. Hence the time required to execute *Label* is $O(n(\log n)^{D-1})$.

4. A cutwidth minimization algorithm for degree three trees. Megiddo et al. [25] describe an $O(n \log n)$ algorithm for determining the search number of a tree. Megiddo [24] shows how to reduce this time bound to $O(n)$. As a direct consequence of Corollary 2.3 these algorithms can be used to determine the cutwidth of a degree three tree, although of course they will not directly yield the optimal layout. In this section we describe an $O(n \log n)$ algorithm for degree three trees that is based on the general algorithm presented in § 3. This algorithm is simpler than the search number algorithm given in [25] and can be used to obtain an optimal layout. Furthermore, we can use it to reduce the complexity of the degree d algorithm by a factor of $\log n$.

Let T be a tree with root u having children v_1, v_2 . Figure 16 gives a procedure *Gamma2* for computing $\Gamma(T)$ from $\Gamma(T[v_1])$ and $\Gamma(T[v_2])$. Notice that there is no degree restriction on $T[v_1]$ or $T[v_2]$. The correctness of the algorithm is established by the following theorem.

THEOREM 4.1. *Let T be a tree with root u having children v_1, v_2 and let $S_1 = \Gamma(T[v_1])$, $S_2 = \Gamma(T[v_2])$. *Gamma2*(S_1, S_2) = $\Gamma(T)$.*

Proof. The correctness of the assertion at line [13] of Fig. 16 follows from Theorem 3.2. Let $k = \max H_1$ and note that $k \leq \gamma(T) \leq k + 1$. Consider three cases.

Case 1. $H_1 \cap H_2 = \emptyset$. Let $h = \max \{\min H_1, \min H_2\}$. We claim that $\Gamma(T) = [i \geq h | i \in H_1 \cup H_2]$. The proof is by induction on $|H_1| + |H_2|$. (Basis) Assume $H_1 = [k]$. Then $T[v_1]$ contains no k -critical vertex and since $H_1 \cong H_2$ and $H_1 \cap H_2 = \emptyset$, $\max H_2 <$

```

[1] procedure Gamma2( $S_1, S_2$ )
[2]   (Relabel if necessary so that  $S_1 \supseteq S_2$ )
[3]   if  $\min S_1 \neq 0$  then  $H_1 \leftarrow S_1$ 
[4]   else begin
[5]      $j \leftarrow \min \{i > 0 \mid i \in S_1\}$ 
[6]      $H_1 \leftarrow \{j\} \cup \{i > j \mid i \in S_1\}$ 
[7]   end
[8]   if  $\min S_2 \neq 0$  then  $H_2 \leftarrow S_2$ 
[9]   else begin
[10]     $j \leftarrow \min \{i > 0 \mid i \in S_2\}$ 
[11]     $H_2 \leftarrow \{j\} \cup \{i > j \mid i \in S_2\}$ 
[12]  end
[13]  ( $H_1 \leftarrow \Gamma(T[u, v_2]), H_2 \leftarrow \Gamma(T[u, v_1])$ )
[14]  if  $H_1 \cap H_2 = \emptyset$  then begin
[15]     $h \leftarrow \max \{\min H_1, \min H_2\}$ 
[16]    return  $\{i \geq h \mid i \in H_1 \cup H_2\}$ 
[17]  end
[18]   $h \leftarrow \max H_1 \cap H_2$ 
[19]  if  $h = \min H_1 = \min H_2$  then
[20]    return  $\{0\} \cup \{i \geq h \mid i \in H_1 \cup H_2\}$ 
[21]   $i \leftarrow \min \{j > h \mid j \in H_1 \cup H_2\}$ 
[22]  return  $\{i\} \cup \{j > i \mid j \in H_1 \cup H_2\}$ 
[23] end

```

FIG. 16. Procedure for computing $\Gamma(T)$ from $\Gamma(T[v_1])$ and $\Gamma(T[v_2])$.

k . Hence u is not k -critical either and $\Gamma(T) = [k]$ as claimed. (Induction) If $|H_1| > 1$ then $T[v_1]$ contains a k -critical vertex x with children y, z such that $\Gamma(T[v_1, y, z]) = H_1 - [k]$. By the induction hypothesis $\Gamma(T[u, y, z]) = [i \geq h \mid i \in (H_1 - [k]) \cup H_2]$. Since the largest integer in this sequence is less than k it follows from Corollary 3.5 that $\Gamma(T) = [k] \cup \Gamma(T[u, y, z])$.

Case 2. $H_1 \cap H_2 = [\min H_1] = [\min H_2]$. Let $h = \min H_1$. We claim that $\Gamma(T) = [0] \cup [i \geq h \mid i \in H_1 \cup H_2]$. The proof is by induction on $|H_1| + |H_2|$. (Basis) Assume $H_1 = H_2 = [h]$. Then neither $T[v_1]$ nor $T[v_2]$ contains a h -critical vertex. However u is h -critical and hence by Theorem 3.2 $\Gamma(T) = [h, 0]$ as claimed. (Induction) If $|H_1| + |H_2| > 2$ then $T[v_1]$ has a k -critical vertex x with children y, z such that $\Gamma(T[v_1, y, z]) = H_1 - [k]$. By the induction hypothesis

$$\Gamma(T[u, y, z]) = [0] \cup [i \geq h \mid i \in (H_1 - [k]) \cup H_2].$$

Since the largest integer in this sequence is less than k it follows from Corollary 3.5 that $\Gamma(T) = [k] \cup \Gamma(T[u, y, z])$.

Case 3. $H_1 \cap H_2 \neq \emptyset \wedge (H_1 \cap H_2 \neq [\min H_1] \vee (H_1 \cap H_2 \neq [\min H_2]))$. Let $h = \max H_1 \cap H_2$ and $i = \min \{j > h \mid j \notin H_1 \cup H_2\}$. We claim that $\Gamma(T) = [i] \cup [j > i \mid j \in H_1 \cup H_2]$. We consider two subcases.

Subcase 3a. $i > k$. In fact, in this case, $i = k + 1$ since $k = \max H_1 > \max H_2$. We claim that $\Gamma(T) = [k + 1]$. The proof is by induction on $k - h$. (Basis) Assume $k - h = 0$. Note that $T[v_1]$ contains a k -critical vertex. Since $k \in H_2$ it follows that u is also k -critical, and by Theorem 3.2, $\Gamma(T) = [k + 1]$ as claimed. (Induction) Assume $k - h > 0$. Note that H_1 contains a k -critical vertex x with children y, z such that $\Gamma(T[v_1, y, z]) = H_1 - [k]$. By the induction hypothesis $\Gamma(T[u, y, z]) = [k]$ and thus by Theorem 3.2, $\Gamma(T) = [k + 1]$.

Subcase 3b. $i < k$. The proof is by induction on $k - i$. (Basis) Assume that $k - i = 1$. Then $T[v_1]$ contains a k -critical vertex with children y, z such that $\Gamma(T[v_1, y, z]) = H_1 - [k]$. By Subcase 3a, $\Gamma(T[u, y, z]) = [i]$. Applying Theorem 3.2 yields $\Gamma(T) = [k, i]$. (Induction) Again $T[v_1]$ contains a k -critical vertex x with children y, z such that $\Gamma(T[v_1, y, z]) = H_1 - [k]$. By the induction hypothesis $\Gamma(T[u, y, z]) = [i] \cup [j > i \mid j \in (H_1 - [k]) \cup H_2]$. By Theorem 3.2 then $\Gamma(T) = [k] \cup \Gamma(T[u, y, z])$ as claimed. \square

For any degree three tree T we can determine the cutwidth by applying *Gamma2* from the bottom up. Each call to *Gamma2* requires time $O(|S_1| + |S_2|) \leq O(k)$ where $k = \gamma(T)$. By Corollary 2.4 $k = O(\log n)$, hence the cutwidth of T can be determined in time $O(n \log n)$.

The procedure *Gamma2* can also be used to speed up the degree d algorithm. By calling *Gamma2* whenever the current vertex has two children, one step of recursion is eliminated, reducing the time required to execute *Gamma* by a factor of $\log n$. This yields an $O(n (\log n)^{d-2})$ algorithm for determining the cutwidth of a degree d tree. In [24], Megiddo gives a linear time algorithm for determining the search number of a tree. The technique used there can also be used to give a linear time version of procedure *Gamma2*. This in turn, can be used to reduce the complexity of the general algorithm by another factor of $\log n$.

It is also worth noting in passing that almost all random trees have a maximum vertex degree that is $O(\log n / \log \log n)$ [29]. Using this one can show that the general cutwidth minimization algorithm runs in time $O(n^{2+\epsilon})$ on random trees where ϵ is any positive constant.

5. A layout algorithm. The labeling algorithms described in the previous sections produce a label $\lambda(x) = \Gamma(T[x])$ for every vertex x in T . Using these labels one can produce an optimal layout of T . The basic idea is contained in the proof of Theorem 2.1. If $\gamma(T) = k$ and T has no k -critical vertex then every vertex x has a child y such that $\gamma(T[x, y]) < k$. Consequently one can construct a path v_1, \dots, v_s where v_1 is the root of T , v_s is a leaf and $\gamma(T[v_i, v_{i+1}]) < k$ for $1 \leq i < s$. This is illustrated in Fig. 17. Once we have found this path we apply the layout algorithm recursively to the subtrees $T[v_i, v_{i+1}]$. If T does have a k -critical vertex x then one can construct a similar path v_1, \dots, v_s . In this case v_1 and v_s are both leaves and x is contained in the path.

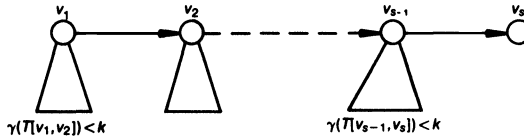


FIG. 17. Motivation for layout algorithm.

```

[1] procedure Layout( $T, \lambda, k, \sigma, pos$ )
[2]    $r \leftarrow$  the root of  $T$ 
[3]   if  $r$  is the only vertex in  $T$  then begin
[4]      $\sigma(r) \leftarrow pos$ 
[5]      $pos \leftarrow pos + 1$ 
[6]     return
[7]   end
[8]   if  $r$  has a child  $x$  such that  $\lambda(x) > [k]$  then begin
[9]      $y \leftarrow$  the vertex that satisfies  $\lambda(y) > [k]$ 
[10]    and for all children  $z$  of  $y$ ,  $\lambda(z) \leq [k]$ 
[11]  else  $y \leftarrow r$ 
[12]   $\langle v_1, \dots, v_s \rangle \leftarrow chain(T, y, \lambda)$ 
[13]  for each vertex  $x$  in  $T$  do
[14]     $\lambda(x) \leftarrow \lambda(x) - [k]$ 
[15]  Layout( $T(v_1, v_2), \lambda, k-1, \sigma, pos$ )
[16]  for  $j = 2$  to  $s-1$  do
[17]    Layout( $T(v_j, v_{j-1}, v_{j+1}), \lambda, k-1, \sigma, pos$ )
[18]  Layout( $T(v_s, v_{s-1}), \lambda, k-1, \sigma, pos$ )
[19]  return
[20]  end
    
```

FIG. 18. Layout procedure.

A procedure for computing the layout from the labels is shown in Fig. 18. T is a tree for which a cutwidth k layout is required and λ is the set of labels produced by one of the labeling algorithms from the previous sections. Upon return from *Layout* (T, λ, k, σ, i) each vertex in T is assigned a unique position in the layout σ starting at position i . More precisely $\forall x \in T, i \leq \sigma(x) < i + |T|$ and $\forall x, y \in T, \sigma(x) \neq \sigma(y)$. Note that the notation for undirected trees is used to specify the subtrees in the recursive calls to *Layout*. The procedure *chain* (\cdot) locates the path discussed above, and is shown in Fig. 19. Figure 20 shows the layout obtained for the tree whose labels were given in Fig. 20.


```

[1] procedure chain(T, v, λ)
[2]   Let  $v_0 = v$  and let  $v_0, \dots, v_r$  be a maximum length
[3]   path in  $T$  satisfying the following conditions for  $0 \leq i < r$ .
[4]     (a)  $v_i$  is the parent of  $v_{i+1}$  and
[5]     (b) for all children  $x$  of  $v_i, \lambda(v_{i+1}) \geq \lambda(x)$ .
[6]   If  $v$  has less than two children then
[7]     return  $\langle v_1, \dots, v_r \rangle$ 
[8]   Let  $u_1$  be a child of  $v$  that satisfies  $u_1 \neq v_1$  and
[9]    $\lambda(u_1) \geq \lambda(x)$  for all children  $x$  of  $v$  such that  $x \neq v_1$ .
[10]  Let  $u_1, \dots, u_s$  be a maximum length path in  $T$  satisfying
[11]  the following conditions for  $1 \leq i < s$ .
[12]    (a)  $u_i$  is the parent of  $u_{i+1}$  and
[13]    (b) for all children  $x$  of  $u_i, \lambda(u_{i+1}) \geq \lambda(x)$ .
[14]  return  $\langle v_r, \dots, v_1, v, u_1, \dots, u_s \rangle$ 
[15] end
    
```

FIG. 19. Procedure for finding a chain.

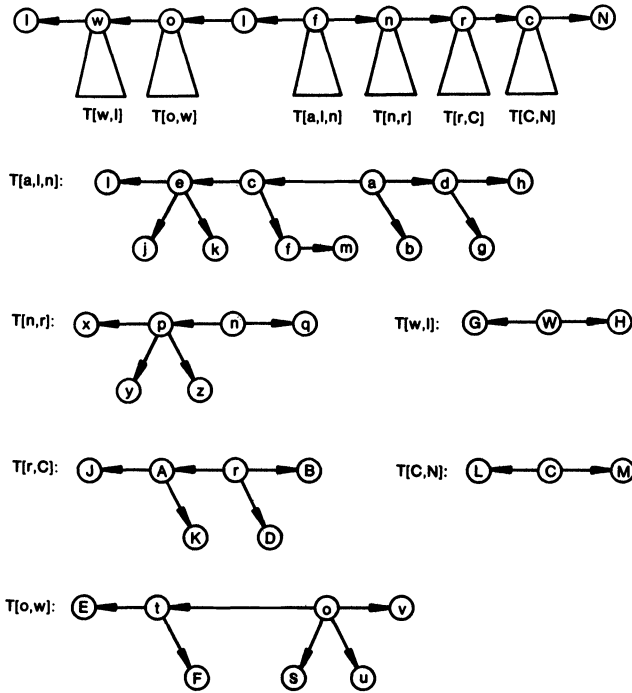


FIG. 20. Example of layout procedure.

6. Open problems. Until recently, the main open question was whether the MIN CUT problem could be solved in polynomial time for unrestricted trees. Yannakakis [42] has now reported a polynomial time algorithm for this problem.

To our knowledge, no good approximation algorithms have been proposed for the MIN CUT problem on graphs. At the same time there is no evidence that such algorithms do not exist. No effort has been made to bound the cutwidth of random graphs. This is a necessary first step in understanding the probabilistic performance of cutwidth minimization algorithms.

A natural extension of the MIN CUT problem is to edge weighted graphs; the weight of a cut being the sum of the weights of all edges in the cut. There is a straightforward log space reduction from the PARTITION problem to this weighted MIN CUT problem on trees. Hence even for trees the problem is NP-complete. However it is not known to be strongly NP-complete. In particular the complexity of the MIN CUT problem on trees with edge weights in $\{1, 2\}$ is open.

Perhaps the most surprising aspect of current research on the MIN CUT problem is the connections that have been found to seemingly unrelated problems. The equivalence of cutwidth and black/white pebble demand on degree three trees in which the sink has in-degree one was completely unexpected and we thank Nick Pippenger for pointing it out to us. As we noted above, this connection not only yields an exact pebbling algorithm for degree three trees, it also yields approximation algorithms for pebbling degree d trees. A closer study of this relationship could deepen our understanding of both problems. Another surprise was the connection between cutwidth and search number. We have shown their equivalence for degree three trees. Makedon and Sudborough [22] have strengthened this to degree three graphs. Recently Chung [4] has shown a connection between the topological bandwidth and the cutwidth of trees. This could lead to efficient algorithms for the topological bandwidth problem.

Appendix. The proof of Theorem 3.3 requires the following two lemmas.

LEMMA 3.1. *Let R be the tree consisting of the single vertex u , let S and S' be trees with roots v, v' and let $T = R \cdot S, T' = R \cdot S'$. $\Gamma(S) \leq \Gamma(S') \Rightarrow \gamma(T) \leq \gamma(T')$.*

Proof. If $\gamma(S) < \gamma(S')$ then the result is immediate. Assume then that $\gamma(S) = \gamma(S') = k$. Note that $k = \delta(T) = \delta(T')$. The proof proceeds by induction on $|\Gamma(S)|$. If $\Gamma(S) = [0]$, then $\gamma(T) = 1 \leq \gamma(T')$. Assume then that $k > 0$. If $\Gamma(S) = [k]$ then S has no k -critical vertex. Consequently T has no k -critical vertex, and by Theorem 3.1, $\gamma(T) = k \leq \gamma(T')$. This establishes the basis of the induction.

Now assume that S and S' violate the lemma, where $|\Gamma(S)| > 1$. Also, assume that there is no pair of trees H, H' that also violate the lemma where $|\Gamma(H)| < |\Gamma(S)|$. We continue to let $\gamma(S) = \gamma(S') = k$. Note that since $\Gamma(S) \leq \Gamma(S')$, $|\Gamma(S')| > 1$. Hence, S contains a k -critical vertex x with children y, z such that $\Gamma(S[v, y, z]) = \Gamma(S) - [k]$ and S' contains a k -critical vertex x' with children y', z' such that $\Gamma(S'[v', y', z']) = \Gamma(S') - [k]$. This is illustrated in Fig. 21. Since S and S' violate the lemma, $\gamma(T) > \gamma(T')$. In fact, we must have $\gamma(T) = k + 1$ and $\gamma(T') = k$. Now, since x is the only k -critical vertex in T , $\gamma(T) = k + 1 \Rightarrow \gamma(T[u, y, z]) \geq k$, by Theorem 3.1. Also, since x' is k -critical, $\gamma(T') = k \Rightarrow \gamma(T'[u, y', z']) < k$. Thus $\gamma(T[u, y, z]) > \gamma(T'[u, y', z'])$. Since $\Gamma(S) \leq \Gamma(S')$, $\Gamma(S[v, y, z]) \leq \Gamma(S'[v', y', z'])$. Hence $S[v, y, z]$ and $S'[v', y', z']$ also violate the lemma, giving the desired contradiction. \square

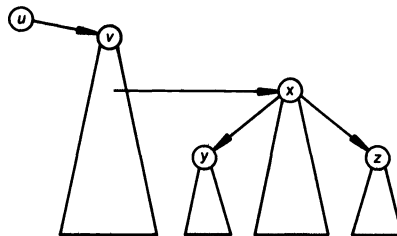


FIG. 21. Illustration for Lemma 3.1.

LEMMA 3.2. *Let R, S, S' be trees with roots u, v, v' and let $T = R \cdot S, T' = R \cdot S'$. $\Gamma(S) \leq \Gamma(S') \Rightarrow \gamma(T) \leq \gamma(T')$.*

Proof. If $\gamma(S) < \gamma(S')$ then we can take any optimal layout of T' and substitute S for S' without increasing the cutwidth of the layout. Assume then that $\gamma(S) = \gamma(S') = k$ and let $\gamma(R) = m$. The proof is by induction on $|R \cdot S|$. The lemma is clearly true if S consists of a single vertex. This together with Lemma 3.1 provides the basis of the induction. Assume that R, S, S' violate the lemma and that there are no H, J, J' that also violate the lemma, where $|H \cdot J| < |R \cdot S|$. We consider three cases.

Case 1. $m < k$. Note that $\delta(T) = k$, and since R, S, S' violate the lemma, $\gamma(T) = k + 1$, $\gamma(T') = k$. If S has no k -critical vertex then neither does T and by Theorem 3.1 $\gamma(T) = k$, which is a contradiction. If, on the other hand, S does have a k -critical vertex then so does S' . Assume then that S has a k -critical vertex x with children y, z such that $\Gamma(S[v, y, z]) = \Gamma(S) - [k]$ and S' has a k -critical vertex x' with children y', z' such that $\Gamma(S'[v', y', z']) < \Gamma(S') - [k]$. Now, since x is the only k -critical vertex in T , $\gamma(T) = k + 1 \Rightarrow \gamma(T[u, y, z]) \geq k$ by Theorem 3.1. Also, since x' is k -critical, $\gamma(T') = k \Rightarrow \gamma(T'[u, y', z']) < k$. Thus, $\gamma(T[u, y, z]) > \gamma(T'[u, y', z'])$. Since $\Gamma(S[v, y, z]) = \Gamma(S) - [k]$ and $\Gamma(S'[v', y', z']) = \Gamma(S') - [k]$, $R, S[v, y, z], S'[v', y', z']$ also violate the lemma giving a contradiction.

Case 2. $m > k$. Note that $\delta(T) \leq m$, and since R, S, S' violate the lemma, $\gamma(T) = m + 1$ and $\gamma(T') = m$. If R has no m -critical vertex then the only possible m -critical vertex in T is u . But if u is not m -critical in R then u has two children r, s in T such that $\gamma(T[u, r, s]) < m$. Then, by Theorem 3.1, $\gamma(T) = m$, which is a contradiction. Thus R must contain an m -critical vertex x . Since x is also m -critical in T' , $\gamma(T') = m \Rightarrow$ that x has children y, z such that $\gamma(T'[u, y, z]) < m$. If x is the only m -critical vertex in T , then by Theorem 3.1, $\gamma(T) = m + 1 \Rightarrow \gamma(T[u, y, z]) \geq m$. If x is not the only m -critical vertex in T , then the other must be u , and by the definition of m -criticality, $\gamma(T[u, y, z]) \geq m$ (since y and z are both in the same subtree of u). Thus $R[u, y, z], S, S'$ violate the lemma giving a contradiction.

Case 3. $m = k$. Note that $\delta(T) = k$. Since R, S, S' violate the lemma, $\gamma(T) = k + 1$, $\gamma(T') = k$. Now, if u is k -critical in R , then by Theorem 3.1, $\gamma(T') = k + 1$, a contradiction. If neither R nor S has a k -critical vertex then the only possible k -critical vertex in T is u . But since u is not k -critical in R , u has children r, s in T such that $\gamma(T[u, r, s]) < k$. Then, by Theorem 3.1, $\gamma(T) = k$, which is a contradiction. Thus, if neither R nor S has a k -critical vertex, then neither does T , and again by Theorem 3.1, $\gamma(T) = k$. Thus either R or S must have a k -critical vertex. By Theorem 3.1, they cannot both have a k -critical vertex since, that would imply $\gamma(T') > k$. If S contains a k -critical vertex x then S' contains a k -critical vertex x' and we proceed as in Case 1. If R contains a k -critical vertex we proceed as in Case 2. \square

Proof of Theorem 3.3. The proof is by induction on $|\Gamma(T)|$. Lemma 3.2 provides the basis. Assume then that R, S, S' violate the theorem and that there are no H, J, J' that violate the theorem such that $|\Gamma(H \cdot J)| < |\Gamma(T)|$. By Lemma 3.2, $\gamma(T) \leq \gamma(T')$, thus for $\Gamma(T) > \Gamma(T')$ to be true we must have $\gamma(T) = \gamma(T') = k$ and T must have some k -critical vertex x . We consider three cases.

Case 1. $x \in S$. In this case x has children y, z such that $\Gamma(S[v, y, z]) = \Gamma(S) - [k]$. Since $\Gamma(S) \leq \Gamma(S')$, S' must have a k -critical vertex x' with children y', z' such that $\Gamma(T'[u, y', z']) = \Gamma(T') - [k]$. Hence $\Gamma(T'[u, y', z']) < \Gamma(T) - [k] \leq \Gamma(T[u, y, z])$ and $\Gamma(S[v, y, z]) \leq \Gamma(S') - [k] < \Gamma(T[u, y, z]) \leq \Gamma(S'[v', y', z'])$. Thus $R, S[v, y, z]$ and $S'[v', y', z']$ violate the theorem giving a contradiction.

Case 2. $x \in R - \{u\}$. In this case x is k -critical in both T and T' and has children y, z, y', z' such that $\Gamma(T[u, y, z]) = \Gamma(T) - [k]$ and $\Gamma(T'[u, y', z']) = \Gamma(T') - [k]$. Since $\Gamma(T') < \Gamma(T)$, $\Gamma(T'[u, y', z']) < \Gamma(T[u, y, z]) \leq \Gamma(T[u, y', z'])$. Thus $R[u, y', z'], S, S'$ violate the theorem giving a contradiction.

Case 3. $x = u$. Since u is k -critical in T , it has no child y such that $\gamma(T[u, y]) < k$. If u is not k -critical in T' then u has a child y' in T' such that $\gamma(T'[u, y']) < k$. If $y' \in R$ then $R[u, y'], S, S'$ violate Lemma 3.2. If $y' = v'$ then $\gamma(T[u, v]) = k > \gamma(T'[u, v'])$, but this is clearly absurd, since $T[u, v] = R = T'[u, v']$. Thus u is k -critical in both T and T' . Further u has children y, z, y', z' such that $\Gamma(T[u, y, z]) = \Gamma(T) - [k]$ and $\Gamma(T'[u, y', z']) = \Gamma(T') - [k]$. Consider two subcases.

Subcase 3a. $y' \neq v' \neq z'$. Since $\Gamma(T') < \Gamma(T)$, $\Gamma(T'[u, y', z']) < \Gamma(T[u, y, z]) \cong \Gamma(T[u, y', z'])$. Hence $R[u, y', z']$, S , S' violate the theorem giving a contradiction.

Subcase 3b. $y' = v'$. Since $\Gamma(T') < \Gamma(T)$, $\Gamma(T'[u, v', z']) < \Gamma(T[u, y, z]) \cong \Gamma(T[u, v, z'])$, but this is contradictory since $T'[u, v', z'] = T[u, v, z']$. \square

Acknowledgments. The authors wish to thank John Ellis and Manrique Mata, participants in the algorithms and complexity seminar at Northwestern University for their valuable suggestions. We would also like to thank F. R. K. Chung for pointing out an error in an earlier version of Theorem 2.2.

REFERENCES

- [1] M. A. BREUER, *Min-cut placement*, J. Design Automation and Fault Tolerant Computing, 1 (1977), pp. 343–362.
- [2] F. R. K. CHUNG, *On linear arrangements of trees*, Bell Laboratories TM-80-1216-31, Murray Hill, NJ, 1980.
- [3] ———, *Some problems and results on labelings of graphs*, in The Theory and Applications of Graphs, G. Chartrand, ed., John Wiley, New York, 1981, pp. 255–264.
- [4] ———, *On the cutwidth and the topological bandwidth of a tree*, SIAM J. Alg. Disc. Meth., 6 (1985), to appear.
- [5] D. DOLEV AND H. TRICKEY, *Embedding a tree on a line*, IBM Technical Report RJ3368, 1982.
- [6] A. FELLER, *Automatic layout of low-cost quick turnaround random-logic custom LSI devices*, Proc. Thirteenth Design Automation Conference, 1976, pp. 79–85.
- [7] M. J. FOSTER AND H. T. KUNG, *Recognizing regular languages with programmable building-blocks*, in VLSI-81 Conference, Aug. 1981.
- [8] MICHAEL R. GAREY, DAVID S. JOHNSON AND L. J. STOCKMEYER, *Some simplified NP-complete graph problems*, Theoret. Computer Sci., 1 (1976), pp. 237–267.
- [9] MICHAEL R. GAREY, R. L. GRAHAM, DAVID S. JOHNSON AND D. E. KNUTH, *Complexity results for bandwidth minimization*, SIAM J. Appl. Math., 34 (1978), pp. 477–495.
- [10] MICHAEL R. GAREY AND DAVID S. JOHNSON, *Computers and Intractability—A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.
- [11] F. GAVRIL, *Some NP-complete problems on graphs*, Proc. 11th Conference on Information Sciences and Systems, John Hopkins Univ., Baltimore, MD, pp. 91–95.
- [12] M. K. GOLDBERG AND I. A. KLIPKER, *Minimal placing of trees on a line*, Technical Report, Physico-Technical Institute of Low Temperatures, Academy of Sciences of Ukrainian SSR, 1976. (In Russian.)
- [13] EITAN M. GURARI AND I. H. SUDBOROUGH, *Improved dynamic programming algorithms for bandwidth minimization and the min-cut linear arrangement Problem*, J. Algorithms, to appear.
- [14] A. S. LAPAUGH, *Recontamination does not help*, Technical Report, Computer Science Dept., Princeton Univ., Princeton, NJ, 1983.
- [15] M. A. IORDANSKII, *Minimal numberings of the vertices of trees*, Soviet Math. Doklady (1974), pp. 1311–1315.
- [16] T. LENGAUER, *Black-white pebbles and graph separation*, Acta Inform., 16 (1981), pp. 465–475.
- [17] T. LENGAUER AND R. E. TARJAN, *The space complexity of pebble games on trees*, Inform. Processing Lett., 10 (1980), pp. 184–188.
- [18] ———, *Asymptotically tight bounds on time-space trade-offs in a pebble game*, J. Assoc. Comput. Mach., 29 (1982), pp. 1087–1130.
- [19] THOMAS LENGAUER, *Upper and lower bounds on the complexity of the min-cut linear arrangement problem on trees*, SIAM J. Alg. Disc. Meth., 3 (1982), pp. 99–113.
- [20] A. D. LOPEZ AND H-F. S. LAW, *A dense gate matrix layout method for MOS VLSI*, IEEE Trans. Electronic Devices, ED-27 (1980), pp. 1671–1675.
- [21] M. C. LOUI, *The space complexity of two pebble games on trees*, MIT Technical Report, MIT/LCS/TM-133, Massachusetts Institute of Technology, Cambridge, 1979.
- [22] F. S. MAKEDON AND I. H. SUDBOROUGH, *Minimizing width in linear layouts*, Lecture Notes in Computer Science, 154, Springer-Verlag, New York, 1983, pp. 478–490.
- [23] FILLIA S. MAKEDON, I. HAL SUDBOROUGH AND C. H. PAPADIMITRIOU, *Topological bandwidth*, in Proc. 8th Colloquium on Trees in Algebra and Programming, 1983; SIAM J. Alg. Disc. Meth., 6 (1985), to appear.

- [24] N. MEGIDDO, *Linear time algorithm for search number in trees*, unpublished manuscript, April 1981.
- [25] N. MEGIDDO, S. L. HAKIMI, MICHAEL R. GAREY, DAVID S. JOHNSON AND CHRISTOS H. PAPADIMITRIOU, *The complexity of searching a graph*, Proc. IEEE Foundations of Computer Science Symposium, 1981, pp. 376–385.
- [26] F. MEYER AUF DER HEIDE, *A comparison of two variations of a pebble game on graphs*, Theoret. Comput. Sci., 13 (1981), pp. 315–322.
- [27] T. OHTSUKI, H. MORI, E. S. KUH, T. KASHIWABARA AND T. FUJISAWA, *One-dimensional logic gate assignments and interval graphs*, IEEE Trans. Circuits and Systems, CAS-26 (1979), pp. 675–684.
- [28] BURKHARD MONIEN AND I. H. SUDBOROUGH, *Bandwidth constrained NP-complete problems*, Proc. 11th ACM Symposium on Theory of Computing, 1981, pp. 207–217.
- [29] JOHN W. MOON, *Counting labeled trees*, Canadian Mathematical Monographs 1, 112 (1970).
- [30] CHRISTOS H. PAPADIMITRIOU, *The NP-completeness of the bandwidth minimization problem*, Computing, 16 (1976), pp. 263–270.
- [31] T. D. PARSONS, *The search number of a connected graph*, Proc. Ninth Southeastern Conference on Combinatorics, Graph Theory, and Computing, 1978, pp. 549–554.
- [32] ———, *Pursuit-evasion in a graph*, in Theory and Application of Graphs, Y. Alavi and D. R. Lick, eds., Springer-Verlag, New York, 1976, pp. 426–441.
- [33] G. PERSKY, D. DEUTSCH AND D. SCHWEIKERT, *LTX—A minicomputer-based system for automated LSI layout*, J. Design Automation and Fault Tolerant Computing, 1 (1977), pp. 217–255.
- [34] N. PIPPENGER, *Pebbling*, IBM Research Report RC8258, 1980.
- [35] ———, *Advances in pebbling*, IBM Research Report RJ3466, 1982.
- [36] ———, private communication.
- [37] R. R. REDZIEJOWSKI, *On arithmetic expressions and trees*, Comm. ACM, 12 (1969), pp. 81–84.
- [38] YOSSI SHILOACH, *A minimum linear arrangement algorithm for undirected trees*, this Journal, 8 (1979), pp. 15–32.
- [39] L. STOCKMEYER, private communication to M. R. Garey and D. S. Johnson, 1974; see [10, p. 201].
- [40] S. TRIMBERG, *Automating chip layout*, in IEEE Spectrum, 1982, pp. 38–45.
- [41] A. WEINBERGER, *Large scale integration of MOS complex logic: a layout method*, IEEE J. Solid State Circuits, 2 (1967), pp. 182–190.
- [42] MIHALIS YANNAKAKIS, *A polynomial algorithm for the min-cut linear arrangement of trees*, Proc. IEEE Symposium on the Foundations of Computer Science, 1983.
- [43] H. YOSHIZAWA, H. KAWANISHI AND K. KANI, *A heuristic procedure for ordering MOS arrays*, Proc. Design Automation Conference, 1975, pp. 384–393.

ADDITIVE COMPLEXITY AND ZEROS OF REAL POLYNOMIALS*

J. J. RISLER†

Abstract. Let $P \in R[X]$ be a polynomial of additive complexity k (the additive complexity is the minimal number of \pm operations needed to compute P over R). It is shown that there exists a constant C (independent of P) such that the number of distinct real zeros of P is $\leq C^{k^2}$.

This is an improvement on a result of Borodin and Cook (SIAM J. Comput., 5 (1970), pp. 146-157). This result is then generalized to polynomials in several variables, the number of zeros being replaced by the number of connected components of the zero set.

Key words. real polynomials, additive complexity, real roots of polynomials

Introduction. Let $P \in R[X]$ be a polynomial with coefficients in the field R of real numbers. The additive complexity of P , noted $L_+^R(P)$, is by definition the minimum number of additions and subtractions required to evaluate P over R (beginning with the constants and the polynomial X), using the four arithmetical operations (addition, subtraction, multiplication and division). The number of multiplications and divisions is therefore not counted in $L_+^R(P)$.

Let $Z(P)$ be the real zeros of P , $\# Z(P)$ the number of distinct real zeros of P , $R(k)$ the set of polynomials P such that $L_+^R(P) \leq k$, and $\rho(k)$ the least upper bound of $\# Z(P)$ for $P \in R(k)$.

Borodin and Cook ([B-C]) have shown that there exists a constant C such that

$$\rho(k) \leq 2^{2^{\dots 2^{Ck}}},$$

where this expression has $k - 1$ exponentiations.

On the other hand, Borodin and Cook [B-C] and Van De Wiele [V-W] have shown that $\rho(k) \geq 3^k$, and it is conjectured that $\rho(k) = 3^k$ (or at least that there exists $C > 0$ such that $\rho(k) \leq C^k$).

I will show here that there exists $C > 0$ such that $\rho(k) \leq C^{k^2}$. This result is then generalized to several variables, the number of distinct real zeros being replaced by the number of connected components of the zero set.

1. Additive complexity of polynomials in one variable over the real field. By definition, a polynomial $P \in R[X]$ is in $R(k)$ if there exists a system of $k + 1$ equations:

$$(1) \quad \begin{aligned} S_1 &= c_1 X^{m_{0,1}} + d_1 X^{m'_{0,1}}, \\ &\vdots \\ S_k &= c_k \prod_{i=0}^{k-1} S_i^{m_{i,k}} + d_k \prod_{i=0}^{k-1} S_i^{m'_{i,k}}, \\ P_* &= c_{k+1} \prod_{i=0}^k S_i^{m_{i,k+1}} \quad (\text{with } S_0 = X), \end{aligned}$$

with $m_{i,j}$ and $m'_{i,j}$ in Z , c_i and d_i in R , and $P(X)$ evaluated from P_* by successive elimination of the S_i ($1 \leq i \leq k$).

We will show:

THEOREM 1.1. *There exists a constant C such that $\rho(k) \leq C^{k^2}$ (i.e. every polynomial $P \in R[X]$ which can be evaluated over R with $k \pm$ operations has less than C^{k^2} real zeros).*

* Received by the editors July 5, 1983 and in revised form August 15, 1983.

† University of Paris VII, UER de Mathématique, 75 251 Paris Cedex 05, France.

The proof is based on a result by Hovansky which uses a variant of ‘‘Descartes’ Lemma,’’ which says that any $P \in R[X]$ with n nonzero monomials has less than $n - 1$ distinct zeros in R_+^* . (R_+^* is the set of positive real numbers, and R^* is the set of nonzero real numbers.)

THEOREM 1.2 ([H₁], and [H₂]). *Let $P_1, \dots, P_n = 0$ be a system of n polynomial equations in real variables X_1, \dots, X_n , the total number of nonconstant monomials (in the polynomials P_1, \dots, P_n) being k . The number of nondegenerate solutions in $(R_+^*)^n$ of this system is less than $2^{(k(k-1)/2)}(n+1)^k$.*

Let us recall that a system of n equations in n unknowns is nondegenerate at a point if the Jacobian determinant of the system is nonzero at this point.

Remarks 1.3. 1) The notions of a nondegenerate solution and an isolated solution are not the same (for a system of n equations in n unknowns).

2) Hovansky’s proof of his theorem shows that the same result is true if the exponents of the X_i ($1 \leq i \leq n$) are in Z or in Q . We will use this fact because in the system (2), below, the exponents can be in Z so as to accommodate division as a basic operation.

3) One can conjecture that Hovansky’s result is true for all the isolated solutions of a system of equations, however, for the special systems we consider here (cf. Lemma 1.4 and Proposition 1.5), the distinction between isolated and nondegenerate zeros is not important.

We shall apply this result to the system of $k+1$ equations in $k+1$ unknowns $X = S_0, S_1, \dots, S_k$:

$$\begin{aligned}
 (2) \quad & S_1 = c_1 X^{m_{0,1}} + d_1 X^{m'_{0,1}}, \\
 & \vdots \\
 & S_k = c_k \prod_{i=0}^{k-1} S_i^{m_{i,k}} + d_k \prod_{i=0}^{k-1} S_i^{m'_{i,k}}, \\
 & 0 = c_{k+1} \prod_{i=0}^k S_i^{m_{i,k+1}}.
 \end{aligned}$$

LEMMA 1.4. *The real solutions of (2) are nondegenerate if and only if the real roots of the polynomial P are simple.*

Proof. It is enough to prove that if (X, S_1, \dots, S_k) is a solution of (2), then $P'(X)$ is equal to $J(X, S_1, \dots, S_k)$, where J is the Jacobian determinant of the system (2). We will do this by induction on k .

Let

$$\begin{aligned}
 T_1(X) &= S_1(X), \\
 T_2(X) &= S_2(X, T_1(X)), \\
 T_k(X) &= S_k(X, T_1(X), \dots, T_{k-1}(X)).
 \end{aligned}$$

We have then:

$$P'(X) = \frac{\partial P_*}{\partial X} + T'_1(X) \frac{\partial P_*}{\partial S_1} + \dots + T'_k(X) \frac{\partial P_*}{\partial S_k},$$

and by the induction hypothesis, $T'_i(X)$ is the Jacobian determinant of the system

$$\begin{aligned}
 & S_1 = c_1 X^{m_{0,1}} + d_1 X^{m'_{0,1}}, \\
 & \vdots \\
 & S_i = c_i \prod_{j=0}^{i-1} S_j^{m_{j,i}} + d_i \prod_{j=0}^{i-1} S_j^{m'_{j,i}} \quad (\text{with } S_0 = X),
 \end{aligned}$$

for $1 \leq i \leq k - 1$.

To prove the lemma, it is now enough to look at the Jacobian determinant of the system (2), which has the form

$$\begin{vmatrix} \frac{\partial S_1}{\partial X} & -1 & 0 & \cdots \\ \frac{\partial S_2}{\partial X} & \frac{\partial S_2}{\partial S_1} & -1 & 0 & \cdots \\ \frac{\partial P_*}{\partial X} & \frac{\partial P_*}{\partial S_1} & \cdots & \frac{\partial P_*}{\partial S_k} \end{vmatrix}. \quad \text{Q.E.D.}$$

So, if we want to apply Theorem 1.2, we must suppose that all the real roots of P are simple. This will be possible by the next proposition.

PROPOSITION 1.5. *There exists ε (with $|\varepsilon|=1$) such that for small enough t ($t \in \mathbb{R}$):*

- (a) *all the roots of the polynomial $P(X) - \varepsilon|t|$ are simple;*
- (b) *the number of roots of $P(X) - \varepsilon|t|$ is greater than the number of roots of $P(X)$ (i.e. $\text{card}(P(X) - \varepsilon|t|^{-1}(0)) \geq \text{card } P^{-1}(0)$).*

Proof. (a) is true if t is not equal to some $P(X_\alpha)$, where the X_α are the roots of the derivative $P'(X)$. (This is an elementary version of Sard's Theorem, which is essential in the proof of Theorem 1.2.)

For (b), one has to count the number of intersection points of the line $y = \varepsilon|t|$ with the curve defined by $y = P(X)$.

Let us assume that the curve $y = P(X)$ cuts the line $y = 0$ at n points such that among them there are:

- n_1 points where the intersection is transverse (in these points the derivative of P is not zero and corresponds to simple roots of P),
- n_2 local minima (for $P(X)$);
- n_3 local maxima (for $P(X)$);
- n_4 points where $P'(X) = 0$, and $P(X)$ changes of sign in a neighborhood.

We have then $n = n_1 + n_2 + n_3 + n_4$.

If we assume for instance that $n_2 \geq n_3$, the next lemma shows that $\varepsilon = +1$ is convenient for condition (b) of Proposition 1.5.

LEMMA 1.6. *Under the above hypothesis, the number of real roots of $P(X) = t$ is $n_1 + 2n_2 + n_4$ for t small enough.*

Proof. This fact is elementary and is illustrated by Fig. 1. It is seen that $P(X)$ has six roots and $P(X) = t$ has seven for small enough t . Q.E.D.

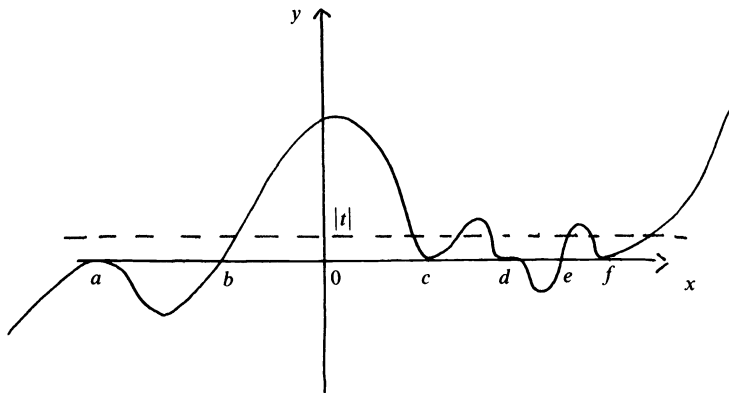


FIG. 1. $n_1 = 2$ (points b and e); $n_2 = 2$ (points c and f); $n_3 = 1$ (point a); $n_4 = 1$ (point d).

Proof of Theorem 1.1. By Proposition 1.5, it is enough to majorize the number of roots of the equation $P(X) = C$, where C is a suitably chosen constant.

We will thus apply Theorem 1.2 to the system (2') (deduced from (2) by replacing the last line with $C = c_{k+1} \prod_{i=0}^k S_i^{m_i, k+1}$).

This system has $3k + 1$ nonconstant monomials. There are then less than

$$2^{k+1}(k+2)^{3k+1}2^{3k(3k+1)/2} = (k+2)^{3k+1}2^{(9k^2+5k+2)/2}$$

solutions in $(R^*)^{k+1}$. But Hovansky's Theorem 1.2 does not depend on the number of constant monomials in the equations in (2'). It is then possible to make a perturbation of the system (adding little constants to the equations) and assume that all the real solutions of (2') are in $(R^*)^{k+1}$. This operation does not change the total number of solutions of (2) because all the roots of P are simple.

We have then

$$\rho(k) \leq (k+2)^{3k+1}2^{(9k+5k+2)/2} \leq C^{k^2}$$

for k large enough, and with $C < 32$.

Theorem 1.1 is now completely proved.

Remark 1.7. Making a direct generalization of Descartes' lemma, one can conjecture like Kuchnirenko that if $P_1 = \dots = P_n = 0$ is a system of equations such that P_i has k_i monomials ($1 \leq i \leq n$), then this system has less than $\prod_{i=1}^n (k_i - 1)$ nondegenerate solutions in $(R_+^*)^n$.

The proof of this conjecture for the system (2') would imply a bound of C^k for $\rho(k)$ (with $C < 5$).

2. Additive complexity of polynomials in several variables. If $P \in R[X_1, \dots, X_n]$, let $C(P)$ be the number of connected components of $Z(P)$. (It is well known that $C(P)$ is finite; anyway, this fact will be a consequence of the following proof.)

THEOREM 2.1. *There is a function $\psi(k, n): N \times N \rightarrow N$ such that for all $P \in R[X_1, \dots, X_n]$ with $L_+^R(P) \leq k$, one has $C(P) \leq \psi(k, n)$.*

Proof. The proof is made by induction on n , the case $n = 1$ having been resolved in § 1.

Let $C_b(P)$ be the number of bounded (i.e. compact) components of $Z(P)$, and $C_n(P)$ the number of nonbounded components.

LEMMA 2.2. *There exists an affine hyperplane $H \subset R^n$ intersecting at least $C_n(P)/2$ unbounded components of $Z(P)$.*

Proof. (A similar argument is in [H₁], § 2.) Let Z_i be a nonbounded component of $Z(P)$, and C_i a smooth connected and unbounded curve such that $C_i \subset Z_i$. In the sphere S^{n-1} , let P_i be a limit point (for $\|x\| \rightarrow +\infty$) of the set $x/\|x\|$, with $x \in C_i$.

If H_1 is a hyperplane (with $0 \in H_1$) such that $P_i \notin H_1$ for all i , and one of the half-spheres delimited by H_1 contains more than half of the points P_i , then an affine hyperplane H , parallel to H_1 and far enough in the direction of this half-sphere, will intersect half of the C_i and half of the Z_i . Q.E.D.

Now let P_H be the restriction of P to H . If $L_+^R(P) \leq k$, one can look at P_H as a polynomial in $n - 1$ variables with $L_+^R(P_H) \leq k + n - 1$ (because the equation of H has less than $n - 1 \pm$ operations).

We have then $C_n(P) \leq 2\psi(k + n - 1, n - 1)$ by the induction hypothesis and Lemma 2.2.

We must now majorize $C_b(P)$. As in § 1 we may now assume that $Z(P)$ is smooth, eventually increasing $L_+^R(P)$ by one. (The proof is the same as the proof of Proposition 1.5.)

PROPOSITION 2.3. *There exists $\varepsilon \in \mathbb{R}$ (with $|\varepsilon| = 1$) such that for small enough t :*

- (a) *$Z(P - \varepsilon|t|)$ is smooth;*
- (b) *the number of compact components of $Z(P - \varepsilon|t|)$ is greater than $C_b(P)$.*

Proof. The proof is similar to the proofs of Proposition 1.5 and Lemma 1.6. The only different thing to do is to verify that if $P^{-1}(0)$ has n_1 compact connected components such that P takes positive values in a neighborhood, then $P^{-1}(t)$ (with $t > 0$) has more than n_1 compact connected components. This is an easy exercise in general topology. Q.E.D.

LEMMA 2.4. *Let r be the number of distinct solutions of the system*

$$(3) \quad \begin{aligned} P(X_1, \dots, X_n) &= 0, \\ \frac{\partial P}{\partial X_1}(X_1, \dots, X_n) &= 0, \\ &\vdots \\ \frac{\partial P}{\partial X_n}(X_1, \dots, X_n) &= 0. \end{aligned}$$

Then $2C_b(P) \leq r$.

(In this lemma we do not assume that r is finite; let us recall that $C_b(P)$ is the number of compact connected components of $Z(P) \subset \mathbb{R}^n$.)

Proof. The solutions of (3) are exactly the critical points of the function X_n restricted to $Z(P)$. But if C is a compact connected component of $Z(P)$, then X_n restricted to C has more than two critical points.

LEMMA 2.5. *The polynomial P being fixed (with $Z(P)$ smooth), there exists a linear change of coordinates, $X_i = M(X'_i)$ ($1 \leq i \leq n$) (where $M \in Gl(n, \mathbb{R})$), such that in the new coordinates, the system*

$$(4) \quad P = 0, \quad \frac{\partial P}{\partial X'_1} = 0, \quad \dots, \quad \frac{\partial P}{\partial X'_n} = 0$$

has a finite number of solutions, all nondegenerate.

Proof. This is a result of ‘‘Bertini type,’’ classical in algebraic geometry; cf. for example [K1, Lemma 1].

LEMMA 2.6. *Let $P(X_1, \dots, X_n) \in \mathbb{R}[X_1, \dots, X_n]$ such that $L_+^R(P) \leq k$. Then $L_+^R(\partial P / \partial X'_i) \leq 3k(k+1)/2$ ($1 \leq i \leq n$).*

Proof. We may assume here that P is a one-variable polynomial. Let us then look at the system of § 1:

$$(1) \quad \begin{aligned} S_1 &= c_1 X^{m_{0,1}} + d_1 X^{m_{0,1}}, \\ &\vdots \\ S_k &= c_k \prod_{i=0}^{k-1} S_i^{m_{i,k}} + d_k \prod_{i=0}^{k-1} S_i^{m_{i,k}}, \\ P_* &= c_{k+1} \prod_{i=0}^k S_i^{m_{i,k+1}} \quad (\text{with } S_0 = X), \end{aligned}$$

and again let $T_1(X) = S_1(X)$, $T_2(X) = S_2(X, T_1(X))$, \dots , $T_k(X) = S_k(X, T_1(X), \dots, T_{k-1}(X))$.

We have then

$$T'_k(X) = \frac{\partial S_k}{\partial X} + T'_1(X) \frac{\partial S_k}{\partial S_1} + \dots + T'_{k-1} \frac{\partial S_k}{\partial S_{k-1}}$$

and

$$P'(X) = \frac{\partial P_*}{\partial X} + T'_1 \frac{\partial P_*}{\partial S_1} + \cdots + T'_k \frac{\partial P_*}{\partial S_k}.$$

By construction one has $L_+^R(T_{k-1}) \leq k-1$. We may hence assume, by induction on k , that we can evaluate T_{k-1} and T'_{k-1} with less than $3k(k-1)/2 \pm$ operations. Then the evaluation of T_k will take one \pm operation more, the evaluation of T'_k will take $2k-1$ operations more (because in the expression of T'_k there are $k-1 \pm$ signs, and each $\partial S_k/\partial S_i$ has two monomials), and the evaluation of $P'(X)$ will take $k \pm$ operations more.

We have then $L_+^R(P') \leq 3k(k-1)/2 + 3k = 3k(k+1)/2$. Q.E.D.

Remark 2.7. Baur and Strassen [B-S] have proved that for the multiplicative complexity L_*^F (or total complexity L^F) over any field F , we have $L_*^F(P, \partial P/\partial X_1, \dots, \partial P/\partial X_n) \leq 3 \cdot L_*^F(P)$ (respectively, $L^F(P, \partial P/\partial X_1, \dots, \partial P/\partial X_n) \leq 5 \cdot L^F(P)$). For the additive complexity, such a strong result is not possible; consider $P = X_1 X_2 \cdots X_n$. However, it is possible that Lemma 2.6 can be substantially improved, and there may be some analogue of the Baur-Strassen result for additive complexity.

We can now finish the proof of Theorem 2.1. If we want to majorize $C_b(P)$, it is enough by Lemma 2.4 to majorize the number of solutions of the system (4), and we will be able to use Hovansky's Theorem 1.2 because of Lemma 2.5.

We shall in fact apply Theorem 1.2 to the system (similar to (2)) from which we can evaluate P and the $\partial P/\partial X_i$ ($1 \leq i \leq n$) from the constants and the X_i . This system has $2n(k+1) + k(k-3)/2$ variables and the same number of equations. The variables are: X'_i ($1 \leq i \leq n$), X_i ($1 \leq i \leq n$), S_i ($1 \leq i \leq k$), $\partial S_i/\partial S_j$ ($1 \leq i \leq k$, $1 \leq j \leq i$), $\partial S_i/\partial X_j$ ($1 \leq i \leq k$, $1 \leq j \leq n-1$) and $\partial T_i/\partial X_j$ ($1 \leq i \leq k$, $1 \leq j \leq n-1$). The total number of monomials of the system is bounded by a third-degree polynomial $q(n, k)$ in n and k , namely $n(n+1) + 3k(k+1)/2 + (3n-1)(k(k+3)/2)$ (see Lemma 2.6, where it is shown how to evaluate $\partial P/\partial X_i$ from the system evaluating P). The application of Theorem 1.2 gives a bound for $C_b(P)$ of $C^{q(n,k)^2}$, where C is an appropriate constant.

The proof of Theorem 2.1 is now complete.

Note added in proof. Prof. Volker Strassen told me after the impression of this paper that Dima Grigoryev independently found similar results (*Lower bounds in algebraic computational complexity*, Notes of Scientific Seminars of LOMI, 118 (1982), pp. 25-82, USSR Academy of Sciences, Steklov Math. Inst., Leningrad Dept.).

REFERENCES

- [B-C] A. BORODIN AND S. COOK, *On the number of additions to compute specific polynomials*, this Journal, 5 (1970), pp. 146-157.
- [B-S] W. BAUR AND V. STRASSEN, *The computational complexity of partial derivatives*, Theoret. Comput. Sci., 22 (1983), pp. 317-330.
- [H₁] A.-G. HOVANSKY, *On a class of systems of transcendental equations*, Soviet Math. Dokl., 22, 3 (1980), pp. 762-765.
- [H₂] ———, *Théorème de Bézout pour les fonctions de Liouville*, I.H.E.S., preprint 1981.
- [K1] KLEIMANN, *Transversality of general translates*, Composition Math., 28 (1974).
- [V-W] J.-P. VAN DE WIELE, *Complexité additive et zéros des polynômes à coefficients réels et complexes*, Rapport IRIA no. 292 1978.

FACTORING POLYNOMIALS OVER ALGEBRAIC NUMBER FIELDS*

SUSAN LANDAU†

Abstract. We show that if $f(x)$ is a polynomial in $Z[\alpha][x]$, where α satisfies a monic irreducible polynomial over Z , then $f(x)$ can be factored over $Q(\alpha)[x]$ in polynomial time. We also show that the splitting field of $f(x)$ can be determined in time polynomial in $([\text{Splitting field of } f(x): Q], \log |f(x)|)$.

Key words. algebraic number fields, norm, polynomial factorization

1. Introduction. Mathematicians have long sought efficient algorithms for factoring polynomials over the rationals. In 1793 Frederick von Schubert showed that the problem of factoring over the integers was decidable [Kn]. If $f(x)$ is the polynomial one desires to factor, von Schubert's idea was to compute $f(1), f(2), \dots, f(n)$, where n is the degree of $f(x)$. Consider a possible sequence $d(1), \dots, d(n)$, where $d(i)$ divides $f(i)$. A sequence defines a potential divisor of $f(x)$, which can be found by interpolation. All divisors of $f(x)$ can be found in this way—if one has enough time. The algorithm is exponential.

If one raises questions of efficiency, one must begin by asking how much space is required to write down the factors of $f(x) = x^n + a_{n-1}x^{n-1} + \dots + a_0$. It is not difficult to show that the roots of $f(x)$ are polynomially bounded in size, which implies the factors of $f(x)$ can be written down in polynomial space. (Mignotte [Mi1], [Mi2] demonstrates tight bounds on the size of factors.)

Algorithms which were developed for factoring polynomials over the integers had exponential running time. An important one which worked well on average was created by Zassenhaus in 1969 [Za.] His idea was to factor $f(x) \bmod p$, for a carefully chosen prime p , and then to lift the factorization to p^k for a large integer k . (In 1969 Berlekamp [Be] discovered an algorithm which factored a polynomial of degree n over Z/pZ in $O(n^3 p)$ steps.) The factorization mod p^k is examined to give a factorization over the integers.

Zassenhaus' algorithm has the problem that its worst case running time is exponential in the degree of the polynomial to be factored. For a time it seemed it might be easier to check polynomial irreducibility than to factor. In 1979 Weinberger [Wei] showed that under the Generalized Riemann Hypothesis, testing irreducibility of polynomials is in polynomial time. In 1981 Cantor [Can] proved that irreducible polynomials had succinct certificates. Still the problem of factoring univariate polynomials over the integers remained stubbornly exponential.

In 1982 Lenstra, Lenstra and Lovász announced an algorithm to factor $f(x) = a_n x^n + \dots + a_0 \in Z[x]$ into irreducible factors over $Z[x]$ in

$$O(n^{9+\varepsilon} + n^{7+\varepsilon} \log^{2+\varepsilon} (\sum a_i^2))$$

steps, for all $\varepsilon > 0$. The L^3 algorithm brings many important problems into polynomial time. It is natural to ask if their algorithm can be extended to larger fields, in particular, are there polynomial time algorithms for factoring polynomials over algebraic and

* Received by the editors August 19, 1982, and in final revised form July 9, 1983. This work was partially supported by the Office of Naval Research under grant N00014-80-C-622.

† Mathematics Department and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

transcendental extensions of the rationals? In this paper we answer the first question affirmatively.

We do so by using norms, a technique proposed by Kronecker in 1882. Assume that α satisfies $g(t) = t^m + a_{m-1}t^{m-1} + \cdots + a_0$ which is irreducible over Q . If $f(x) = x^n + \beta_{n-1}x^{n-1} + \cdots + \beta_0$, where $\beta_i = \sum_{j=0}^{m-1} b_{ij}\alpha^j$, $b_{ij} \in \mathbb{Z}$, then we can factor $f(x)$ over $Q(\alpha)[x]$ in time polynomial in the size and degree of $g(t)$ and $f(x)$. (Chistov and Grigoriev [CG] have also observed that Kronecker's technique produces a polynomial time algorithm.) By generalizing the algorithm in [L³], A. K. Lenstra [AKL] has found a different method for factoring polynomials over algebraic number fields in polynomial time.

This paper is organized as follows: § 2 Background, § 3 The norm, § 4 Computations in algebraic number fields, § 5 An algorithm, and § 6 Ramifications and open questions.

2. Background. It is a simple matter to show that if $g(x)$ divides $f(x)$ in $\mathbb{Z}[x]$, then $g(x)$ is polynomial size as a function of $f(x)$ to write down. The situation is only slightly more complex in the case of algebraic number fields. We begin with some definitions. An element α is *algebraic over a field K* iff α satisfies a polynomial with coefficients in K . An extension field L is *algebraic over a field K* iff every element in L is algebraic over K . It is well known that every finite extension of a field is algebraic; the finite extensions of Q are called the *algebraic number fields*.

Every algebraic number field is expressible as $Q(\alpha)$ for a suitable α . The field $Q(\alpha)$ is isomorphic to $Q[t]/g(t)$, where $g(t)$ is the minimal (irreducible) polynomial for α . In our algorithms we will work with the number field in its formulation as $Q[t]/g(t)$, although certain of our proofs will be in terms of $Q(\alpha)$. Let the degree of $g(t)$ be m . The conjugates of α are the remaining roots of $g(t)$: $\alpha_2 \cdots \alpha_m$, α can be thought of as α_1 . Since $Q(\alpha)$ is of characteristic 0, by the minimality of $g(t)$ we know the α_i 's are all distinct. (Note that the fields $Q(\alpha_i)$ are all isomorphic.)

It is convenient for us to consider a special class of algebraic numbers, the algebraic integers. A number α is an *algebraic integer* iff it is a root monic polynomial over \mathbb{Z} . For the remainder of this paper we assume $\alpha = \alpha_1, \alpha_2, \dots, \alpha_m$ are algebraic integers satisfying $g(t)$, a monic irreducible polynomial over \mathbb{Z} . The set of algebraic integers of $K = Q(\alpha)$ form a ring, frequently written O_K . This ring is a natural extension of the integers, and in particular, Gauss' lemma generalizes:

PROPOSITION 1.1. *Let $f(x) \in O_K[x]$ be monic. Then $f(x)$ factors as the product of two polynomials with coefficients in K iff $f(x)$ factors as the product of two polynomials with coefficients in O_K .*

If we factor $f(x)$, a polynomial in a number ring, the factors of $f(x)$ also lie in the number ring. It is somewhat more complicated than it was in the case of the integers to show that factors of $f(x)$ over O_K will have short descriptions. First we need to know what the ring of integers of an algebraic number field looks like. In general, computing a basis for the ring of integers of an algebraic number field is at least as hard as determining the squarefree part of an integer [Mar], and it may be as difficult as factoring. Fortunately it is not necessary to do so:

PROPOSITION 1.2. *Let α be an algebraic integer satisfying $g(t)$, a monic irreducible polynomial over \mathbb{Z} . The ring of algebraic integers of $Q(\alpha)$ is contained in $(1/d)\mathbb{Z}[\alpha]$, where*

$$d^2 |\text{disc}(g(t))| = \prod_{i < j} (\alpha_i - \alpha_j)^2.$$

If we factor a polynomial over $Z[\alpha][x]$, we are guaranteed that the coefficients of the factors lie in $(1/d)Z[\alpha]$. If we show that an integer coefficient of a factor of a polynomial in a number field is less than the integer “ a ” say, then the coefficient can be written as b/d where $|b| < |a|/|d|$. Thus bounding a coefficient in absolute value bounds it in length of description.

We consider this question in greater detail. If $g(t) = t^m + a_{m-1}t^{m-1} + \dots + a_0$, a_i in Z , then we define the *size* of $g(t)$, $|g(t)| = (\sum_{i=0}^{m-1} a_i^2)^{1/2}$. Following Weinberger and Rothschild, we define the *size* of β , $\| \beta \|$ to be the maximum of the absolute values of the conjugates of β . If $f(x) = \beta_n x^n + \beta_{n-1} x^{n-1} + \dots + \beta_0$, with $\beta_i = \sum_{j=0}^{m-1} b_{ij} \alpha^j$, then the *size* of $f(x)$, $\| f(x) \|$, is defined to be $\max_i (\sum_{j=0}^{m-1} b_{ij}^2)^{1/2}$. Then we have the following theorem, whose proof appears in the Appendix.

THEOREM 1.3 (Weinberger and Rothschild). *Let β be a root of $f(x) \in Z[\alpha][x]$, notation as above. Then $\| \beta \| \leq 1 + \| f(x) \|$. Assume that $f(x)$ is monic, and let*

$$h(x) = h_r x^r + h_{r-1} x^{r-1} + \dots + h_0$$

be a factor of $f(x)$ in $(1/d)Z[\alpha][x]$ which is primitive. If $h_i = (1/d)(c_{im-1} \alpha^{m-1} + \dots + c_{i0})$, then $|c_{ij}| \leq d^{(m)} \| f(x) \| |g(t)|^m \text{disc}(g(t))^{-1/2}$.

The result of Theorem 1.3 is somewhat cumbersome to express. As we are concerned with asymptotic running times, we will assume that $|c_{ij}| \leq \| f(x) \| (m |g(t)|)^m$.

It is often easier to compute in the rationals than in the algebraic number fields, because of the rationals’ simpler structure. A useful tool is the norm, which relates elements in the number fields to elements in Q . Let $\beta = a_0 + a_1 \alpha + \dots + a_{m-1} \alpha^{m-1} \in Q(\alpha)$. Then

$$\text{Norm}(\beta) = N(\beta) = \prod_i (a_0 + a_1 \alpha_i + \dots + a_{m-1} \alpha_i^{m-1}).$$

If σ is an element of the Galois group of $g(t)$ over Q , then $\sigma(\alpha) = \alpha_j$, where α_j is a conjugate of α over Q . Then

$$\begin{aligned} \alpha_j(N(\beta)) &= \sigma_j \left(\prod_i (a_0 + a_1 \alpha_i + \dots + a_{m-1} \alpha_i^{m-1}) \right) \\ &= \prod_i \sigma_j(a_0 + a_1 \alpha_i + \dots + a_{m-1} \alpha_i^{m-1}) \\ &= \prod_i (a_0 + a_1 \alpha_i + \dots + a_{m-1} \alpha_i^{m-1}) \\ &= N(\beta), \end{aligned}$$

and since σ_j just permutes the α_i ’s we conclude that $N(\beta) \in Q$. (In particular, for q in Q , we have $N(q) = q^m$.) Since the norm is the product of multiplicative functions, the norm itself is multiplicative, i.e. $N(\gamma\beta) = N(\gamma)N(\beta)$. We can think of a polynomial $f(x) \in Q(\alpha)[x]$ as a polynomial in two variables x and α , and denote it by $f_\alpha(x)$. It is quite natural to extend the definition of norm to polynomials in $Q(\alpha)[x]$ by

$$N(f(x)) = \prod_i f_{\alpha_i}(x)$$

If $f(x) \in Q(\alpha)[x]$, $N(f(x)) \in Q[x]$. Under appropriate hypotheses, a polynomial in $Q(\alpha)[x]$ can be factored by taking the norm of the polynomial, factoring the norm over the rationals, and raising that to a factorization over the number field. We examine these hypotheses in greater detail in the next section.

3. The norm.

THEOREM 1.4. *Let $f(x) \in Q(\alpha)[x]$ be irreducible. Then $N(f(x))$ is a power of an irreducible polynomial in $Q[x]$.*

Proof. Suppose not. Then $N(f(x)) = C(x)D(x) \in Q[x]$, where $C(x)$ and $D(x)$ are relatively prime. $N(f(x)) = \prod_i f_{\alpha_i}(x)$: therefore $f_{\alpha}(x)$ must divide $C(x)$ or $D(x)$ in $Q(\alpha)[x]$. Without loss of generality, $f_{\alpha}(x)|C(x)$, which implies that there exists $g_{\alpha}(x) \in Q(\alpha)[x]$ such that $f_{\alpha}(x)g_{\alpha}(x) = C(x)$. Let $\sigma: Q(\alpha)[x] \rightarrow Q(\alpha_i)[x]$ be an element of the Galois group which sends α to α_i . Then $\sigma(C(x)) = C(x)$ since $C(x)$ is in $Q[x]$, but $\sigma(f_{\alpha}(x)) = f_{\alpha_i}(x)$ and $\sigma(g_{\alpha}(x)) = g_{\alpha_i}(x)$. Thus we have $f_{\alpha_i}(x)|C(x)$ for all α_i which are conjugates of α . Now $C(x)$ and $D(x)$ are relatively prime (and not both 1.) Therefore for all α_i , $f_{\alpha_i}(x) \nmid D(x)$, which implies that $N(f(x)) = \prod_i f_{\alpha_i}(x) = C(x)$, and consequently $N(f(x))$ is a power of an irreducible polynomial. \square

THEOREM 1.5. *Let $f(x) \in Q(\alpha)[x]$ be such that $N(f(x))$ is squarefree. Then if $N(f(x)) = \prod_i G_i(x)$ is a factorization into irreducible polynomials in $Q[x]$, then $f(x) = \prod_i \gcd(f(x), G_i(x))$ is a factorization into irreducibles in $Q(\alpha)[x]$.*

Proof. Let $g_i(x) = \gcd(f(x), G_i(x))$. Then we need to show that each $g_i(x)$ is irreducible, and that each irreducible factor of $f(x)$ appears in $\prod_i g_i(x)$. Let $h(x)$ be an irreducible factor of $f(x)$ in $Q(\alpha)[x]$. By Theorem 1.4, $N(h(x))$ is a power of an irreducible polynomial. But $N(h(x))|N(f(x))$, and $N(f(x))$ is squarefree; thus $N(h(x)) = G_i(x)$ for some i .

The norm is multiplicative; thus the norm of $f(x)$ equals the product of the norms of the irreducible factors of $f(x)$. Each $G_i(x)$ is the norm of some irreducible factor of $f(x)$. The $G_i(x)$'s are all irreducible and distinct, which implies that the $g_i(x)$'s are all distinct and irreducible. Since all the irreducible factors of $f(x)$ appear as some $\gcd(f(x), G_i(x))$ we are done. \square

Our algorithm should now be clear. We begin with $f(x)$. So long as $N(f(x))$ is squarefree, we factor it over the rationals, then compute gcd's to obtain a factorization over $Q(\alpha)[x]$. These steps—computing the norm, factoring over the rationals, and taking gcd's—are all in polynomial time. The question of what to do if $N(f(x))$ is not squarefree remains. Kronecker [Kr] observed that so long as $f(x)$ has no repeated roots in $Q(\alpha)[x]$, $f(x)$ can be “twiddled” so as to obtain a polynomial with squarefree norm. The proof we present is due to Trager [Tr.]

LEMMA 1.6. *Let $f(x) \in Q(\alpha)[x]$ be a squarefree polynomial of degree n , where $[Q(\alpha): Q] = m$. Then there are at most $(nm)^2/2$ integers s such that $N(f(x - s\alpha))$ is not squarefree.*

Proof. Instead we show that there are at most $n(n-1)m(m-1)/2$ integers s such that $N(f(x - s\alpha))$ has a repeated root; this will immediately imply the result. Suppose that the roots of $f(x)$ are $\{\beta_i\}$, then the roots of $N(f(x - s\alpha))$ are $\{\beta_i + s\alpha_j\}$, where the α_j 's are conjugates of α . Then $N(f(x - s\alpha))$ has a repeated root iff $\beta_i + s\alpha_j = \beta_k + s\alpha_l$, for some $i \neq k$ or $j \neq l$. This would mean $s = (\alpha_l - \alpha_j)/(\beta_k - \beta_i)$. (We can divide, since $f(x)$ squarefree means that $\beta_k \neq \beta_l$, for $k \neq l$.) Clearly there are at most $(n(n-1)m(m-1))/2$ such s . \square

The algorithm we have suggested to factor polynomials requires the computation of norms. In the next section we show how we can do this in polynomial time.

4. Computations in algebraic number fields. The coefficients of the norm are all symmetric functions in the α_i , since $N(f(x)) = \prod_i f_{\alpha_i}(x)$. The straightforward way to calculate them takes exponential time. Fortunately there is a way around this difficulty. Let $h(x) = h_r x^r + h_{r-1} x^{r-1} + \dots + h_0$, $k(x) = k_s x^s + k_{s-1} x^{s-1} + \dots + k_0$, for $h_i, k_j \in K$, a field. We define the *resultant*,

$$\text{Res}_x (h(x), k(x)) = \begin{vmatrix} k_s & 0 & \cdots & 0 & h_r & 0 & \cdots & 0 \\ k_{s-1} & k_s & 0 & \cdots & 0 & h_{r-1} & h_r & 0 & \cdots & 0 \\ k_{s-2} & k_{s-1} & k_s & \cdots & 0 & h_{r-2} & h_{r-1} & h_r & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ k_{s-r} & k_{s-r+1} & \cdots & k_s & h_{r-s} & h_{r-s+1} & \cdots & h_r \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & k_0 & k_1 & 0 & 0 & \cdots & h_0 & h_1 \\ 0 & 0 & 0 & \cdots & k_0 & 0 & 0 & 0 & \cdots & h_0 \end{vmatrix}$$

$\underbrace{\hspace{15em}}_r \qquad \underbrace{\hspace{15em}}_s$

It can be shown that $\text{Res}_x (h(x), k(x)) = h_r^s \prod_i k(\beta_i)$, where the β_i are roots of $h(x)$ [vdW]. Thus $N(f(x)) = \prod_i f_{\alpha_i}(x) = (\text{Res}_t (g(t), f(x, t))) / g_m^n$, where $f(x, t)$ is $f(x)$ with t 's substituted in for α 's.

We introduce the resultant because it is a computationally efficient way to compute the norm. After we examine gcd algorithms, we will be ready to factor polynomials over algebraic number fields.

Algebraic computation has benefitted from the fact that many classical algorithms in algebra and number theory are highly efficient. This includes the Euclidean algorithm; however, a naive implementation runs the problem of coefficient blowup. Collins, and Brown and Traub were able to resolve this difficulty by using the theory of subresultants. In our algorithm, we will need to compute gcd's of polynomials over Q and over algebraic number fields. We generalize Brown's result in Corollary 8. The proof appears in the Appendix.

THEOREM 1.7 (Brown). *Let $f(x)$ and $g(x)$ be polynomials over $Q[x]$, of degree m and n respectively. Then $\text{gcd}(f(x), g(x))$ can be computed in $O(\max^2(\log |f(x)|, \log |g(x)|)(\max^4(m, n)))$ steps.*

COROLLARY 1.8. *Let α satisfy a monic irreducible polynomial $g(t)$ over Z of degree m and with discriminant d . Let $f_1(x), f_2(x)$ be polynomials over $O_K, K = Q[t]/g(t)$ with maximum degree n and $\|f_1(x)\|, \|f_2(x)\|$ bounded by $\|f(x)\|$. Then $\text{gcd}_K(f_1(x), f_2(x))$ can be computed in $O(n^7 m^2 \log^2 |g(t)| \log^2 \|f(x)\|)$ steps.*

5. An algorithm. Let α be a root of $g(t)$, a monic irreducible polynomial with coefficients in Z , and discriminant d , and suppose $f(x)$ of degree n is a polynomial whose coefficients lie in O_K , where $K = Q(\alpha)$. We can think of $f(x)$ as a polynomial in two variables, x and α . (When there is no risk of confusion, we use $f(x)$ and $f(x, t)$ interchangeably.) In § 2, we sketched an algorithm due to Kronecker, for factoring polynomials over an algebraic number field. We find $h(x) = \text{gcd}(f(x), f'(x))$. Then $h(x)$ is squarefree, and all the irreducible factors of $f(x)$ appears as factors of $h(x)$. We compute an integer "c" such that $N_{Q(\alpha)/Q}(h(x - c\alpha)) = F(x)$ is squarefree. Using the L^3 algorithm, we factor $F(x) = \prod_{i=1}^r F_i(x)$ over Q . By computing the $\text{gcd}_{Q(\alpha)}(F_i(x), h(x))$ for $i = 1, \dots, r$, we obtain a factorization of $h(x)$ over $Q(\alpha)$. This allows us to determine a factorization of $f(x)$ over $Q(\alpha)$. Algorithm 2.1, which given $g(t)$, a monic irreducible polynomial over Z , and $f(x)$, a polynomial in O_K , where $K = Q[t]/g(t)$ factors $f(x)$ over $O_K[x]$ appears in the Appendix.

THEOREM 2.1. Algorithm 2.1 computes a factorization of $f(x)$, a polynomial of degree n over $O_K[x]$ into irreducible factors in $O_K[x]$. It does so in $O(m^{9+\epsilon} n^{7+\epsilon} \log^2 |g(t)| \log^{2+\epsilon} (\|f(x)\| (m|g(t)|)^n (mn)^n)$ steps.

Proof. The algorithm has four major steps. Step 1 computes the squarefree part, $h(x)$, of $f(x)$. In order to factor $f(x)$ it suffices to factor $h(x)$. Step 2 computes an integer c such that $\text{Norm}_{(Q[t]/g(t))/Q}(h(x-ct))$ is squarefree. Lemma 1.6 guarantees that there is a c less than $(\text{degree}_Q(g(t))\text{degree}_{Q(\alpha)}(f(x)))^2$ which yields $h(x-ct)$ which has squarefree norm.

In Step 3, we factor $l(x) = N(h(x-ct))$. Theorem 1.6 assures us that if $l(x) = \prod_{i=1}^r F_i(x)$ is a complete factorization of $l(x)$ in $Q[x]$, then

$$h(x-ct) = \prod_{i=1}^r \gcd(F_i(x), h(x-ct)) = \prod_{i=1}^r f_i(x-ct)$$

will be a complete factorization of $h(x-ct)$ in $Q(\alpha)[x]$. We are interested in a factorization of $h(x)$ however; we compute $f_i(x) = \gcd(F_i(x+ct), h(x))$. All that remains to be done is the factorization of $k(x)$. Because all irreducible factors of $k(x)$ appear as factors of $h(x)$, by computing gcd's in Step 5 we obtain a complete factorization of $f(x)$.

By the work of Collins, Brown and Traub on polynomial gcd's, it is clear that all of the above steps can be done in polynomial time. We do a careful analysis to obtain the bounds of the theorem. Note that the work of Weinberger and Rothschild shows that $h(x)$ in Step 1, and the $f_i(x)$ in Steps 4 and 5 are polynomial size in $(\log \|f(x)\|, \log |g(t)|, m, n)$ to write down.

Step 1 requires one gcd over $Q[t]/g(t)$ to obtain $k(x)$ and $h(x)$. The time required for Step 1 is subsumed by the time required for steps 2 and 4.

In Step 2, we must find a c such that $\text{Norm}_{(Q[t]/g(t))/Q}(h(x-ct))$ is squarefree. We compute the norm by resultants. The resultant is the determinant of a $2m \times 2m$ matrix whose entries are coefficients of x in $h(x-ct)$. The integer coefficients of t in $h(x-ct)$ are bounded by $B_1 = n^n \|f(x)\| (m|g(t)|)^m (mn)^n$ in absolute value, and therefore the integer coefficients of the resulting polynomial, the Norm, are bounded by $B_2 = (2m)! B_1^{2m} = (2m)! (n^n \|f(x)\| (m|g(t)|)^m (mn)^n)^{2m}$. We need to determine if $N(h(x-ct))$ is squarefree, and we do this by factoring $N_{Q(\alpha)/Q}(h(x-c\alpha))$ over Z/pZ , where p does not divide $\text{disc}(N_{Q(\alpha)/Q}(h(x-c\alpha)))$. Let p be a prime which does not divide $\text{disc}(N_{Q(\alpha)/Q}(h(x-c\alpha)))$. (We are guaranteed there is such a prime less than $\log(n^n B_2^{2n-1})$ [L³, p. 27.]) Now if p is such a prime, then $N(h(x-c\alpha))$ has a squarefree factorization mod p . The time required to factor a polynomial of degree n over Z/pZ is $O(n^3 p)$ steps [R]; thus the factorization takes no more than a constant times $n^3 \times mn(\log \|f(x)\| + m \log(m|g(t)|)) = O(n^4 m(\log \|f(x)\| + m \log m|g(t)|))$ steps. Since it must be repeated at most mn times, Step 2 requires at most $O(n^5 m^2(\log \|f(x)\| + m \log(m|g(t)|)))$ steps.

Step 3 factors $l(x) = N_{Q(\alpha)/Q}(h(x-c\alpha))$ which is squarefree. The integer coefficients of x in $N(h(x-ct))$ are less than a constant $\times (\|f(x)\| (m|g(t)|)^n (mn)^n)^m$ in absolute value, or require at most $O(m(\log(\|f(x)\| (m|g(t)|)^n)))$ bits to write down. Thus $l(x)$ can be factored in $O(m^{9+\epsilon} n^{7+\epsilon} \log^{2+\epsilon}(\|f(x)\| (m|g(t)|)^n (mn)^n))$ steps to factor.

In Step 4, we compute at most n gcd's of polynomials. The factors determined in Step 3 of the algorithm are of degree at most mn , and have coefficients of length at most $(\log \|f(x)\| + m \log(m|g(t)|))$ bits, while $h(x)$ is of degree at most n , again with integer coefficients requiring at most $(\log \|f(x)\| + m \log(m|g(t)|))$ bits. Thus this step can be done in at most $O((mn)^7 \log^2(m|g(t)|) \log^2(\|f(x)\| + m^2 \log^2(m|g(t)|)))$ steps. We do this at most n times, therefore requiring no more than $O(m^7 n^8 \log^2 |g(t)| (\log^2 \|f(x)\| + m^2 \log(m|g(t)|)))$ steps. Finally the running time of

Step 5 is dominated by that of Step 4. Our total running time is dominated by Steps 3 and 4 of the algorithm, and the theorem is proved. \square

We observed earlier that an algebraic number field can be written as $Q(\alpha)$ for an appropriate α . In our algorithm, we assumed that the number field over which we are factoring was presented as $Q(\alpha)$. Suppose we were asked to factor $f(x) \in Q(\alpha, \beta)[x]$; how would we proceed? We could calculate a primitive element for $Q(\alpha, \beta)$, and apply the Algorithm 2.1 directly. Alternatively, we might observe that

$$N_{Q(\alpha, \beta)/Q}(f(x)) = N_{Q(\alpha)/Q}(N_{Q(\alpha, \beta)/Q(\alpha)}(f(x))).$$

In order to factor $f(x)$ over $Q(\alpha, \beta)$, we could compute $N_{Q(\alpha, \beta)/Q(\alpha)}(f(x))$, and then consider the question of factoring that polynomial over $Q(\alpha)$. Such an approach leads to a bootstrapping technique for factoring which is, in some cases, faster than the method of finding a primitive element. However, we have found it useful, and *not more costly* to obtain a primitive element. Recall that if β satisfies $h(x)$, an irreducible polynomial over $Q(\alpha)$, then whenever $N_{Q(\alpha)/Q}(h(x - c\alpha))$ is squarefree, $Q(\beta + c\alpha) = Q(\alpha, \beta)$. We remind the reader that Lemma 1.6 guarantees that such a c can be quickly determined.

6. Ramifications and open questions. The ability to factor allows many other computations. Questions whose solutions were infeasible are now in polynomial time. We list several consequences of Algorithm 2.1 before we suggest some open questions.

COROLLARY 1. *Factoring multivariate polynomials over algebraic number fields is polynomial time reducible to factoring multivariate polynomials over the rationals.*

Proof. The algebraic property necessary for the proofs of Theorems 2 and 3 is that $Q(\alpha)[x]$ is a unique factorization domain. Since $Q(\alpha)[x_1, \dots, x_n]$ is also, Theorems 2 and 3 extend to these domains. To prove Lemma 4, we consider $f(x_1, \dots, x_n) \in Q(\alpha)[x_1, \dots, x_n]$ as a polynomial in x_1 with coefficients in $Q(\alpha)[x_2, \dots, x_n]$. (Note that since we can factor $n+1$ variable polynomials over Q , we can compute the gcd of n variable polynomials over $Q(\alpha)$.) Let $\deg_{x_1}(f(x_1, \dots, x_n)) = n_1$, and $[Q(\alpha) : Q] = m$. As before, we assume $f(x_1, \dots, x_n)$ is squarefree; otherwise we take the gcd to obtain the square free part of $f(x_1, \dots, x_n)$. Then $N(f(x_1, \dots, x_n))$ has no repeated roots. Viewing $f(x_1, \dots, x_n)$ as a polynomial in x_1 with coefficients in $Q(\alpha)[x_2, \dots, x_n]$, it has n_1 roots. The proof of the lemma goes through as before, and we obtain our reduction. \square

Kaltofen [Ka1], [Ka2], and A. Lenstra [Lpc] have independently shown that factoring a polynomial with a bounded number of variables over the rationals is polynomial time equivalent to factoring a univariate polynomial over the rationals. In light of Corollary 2.3 and the earlier [L³] result, we conclude that factoring a polynomial with a bounded number of variables over an algebraic number field presented as $Q(\alpha)$ can be done in polynomial time.

COROLLARY 2. *Let α satisfy $g(t)$, an irreducible polynomial of degree m over Z , and let β satisfy $f(x)$, an irreducible polynomial of degree n over $Z[\alpha]$. Then determining if the intersection of $Q(\alpha)$ and $Q(\beta)$ is Q can be done in time polynomial in $(\log |g(t)|, m, \log \|f(x)\|, n)$.*

Proof. Let $h(x)$ be the minimal polynomial of β over Q . If α does not satisfy $h(x)$, (i.e. α and β are not conjugates over Q), then $Q(\alpha) \cap Q(\beta) = Q$ iff $h(x)$ remains irreducible over $Q(\alpha)$. If α is a root of $h(x)$, then $Q(\alpha) \cap Q(\beta) = Q$ iff $h(x)/x - \alpha$ is irreducible over $Q(\alpha)$. \square

Those number fields, $Q(\alpha)$, which are distinguished by the fact that α may be expressed as a combination of several m th roots are called the *radical number fields*.

COROLLARY 3. *Finding bases for radical number fields can be done in polynomial time.*

COROLLARY 4. *Finding bases for algebraic number fields can be done in polynomial time.*

For a long time normal polynomials—polynomials which factor completely upon adjoining a single root—were most difficult to factor. However we would like to note the following corollary:

COROLLARY 5. *Let $f(x) \in Z[x]$ be of degree n . Then $f(x)$ can be checked for normality in time polynomial in $(\log |f(x)|, n)$. Furthermore, if $f(x)$ is normal, computing its Galois group can be done in time polynomial in $(\log |f(x)|, n)$.*

The algorithm proposed by Galois for computing splitting fields involved factoring a polynomial of degree $n!$ “Bootstrapping”, factoring $f(x)$ in $Q[x, y]/f(y)$, adjoining another root of $f(x)$, etc., until a splitting field is reached, gives:

COROLLARY 6. *Let $f(x)$ be a polynomial in $Z[x]$. The splitting field of $f(x)$ can be determined in time polynomial in $([Splitting\ field\ ((f(x))/Q):\ Q], \log |f(x)|)$.*

Proof. Our algorithm is as follows: we begin with a field $Q(\alpha_1, \dots, \alpha_i) = Q(\beta_i)$ of degree n_i over Q , in which $f(x)$ has been factored. Let α_{i+1} be a root of $f_{i+1}(x)$, an irreducible factor of $f_{i+1}(x)$ in $Q(\beta_{i+1})$. Then whenever $N(f_{i+1}(x - k_{i+1}\beta_i))$ is squarefree, $\alpha_{i+1} + k_{i+1}\beta_i = \beta_{i+1}$ generates $Q(\alpha_1, \dots, \alpha_{i+1}) = Q(\beta_{i+1})$. Then we factor $f(x)$ in $Q(\beta_{i+1})$, and repeat the process. We must show:

- (1) The k_i can be determined in time polynomial in $(n_i, \log |f(x)|)$.
- (2) The minimum polynomials for β_i over Q have small integer coefficients, i.e. the number of bits needed to write down the integer coefficients of the minimum polynomial of β_i is polynomial in $(n_i, \log |f(x)|)$.
- (3) The number of bits needed to write down the integer coefficients of the factors of $f(x)$ in $Q(\beta_i)$ are bounded by a polynomial function of $(n_i, \log |f(x)|)$.

We prove Corollary 6 by induction on i . When $i = 1$, $\beta_1 = \alpha_1$, and $k_1 = 1$, so (1) and (2) are true, and (3) is a consequence of Theorem 2.1. We assume that the theorem is true for i , i.e. that (1), (2) and (3) hold for i , and that we know β_i (i.e. we know the minimum polynomial for β_i); we now prove it for $i + 1$. Now we know that $k_{i+1} < n_{i+1}^2$, by the discussion that follows the proof of Theorem 2.1. Thus

$$\begin{aligned} \llbracket \beta_{i+1} \rrbracket &= \llbracket k_{i+1}(k_i(\dots(k_2\alpha_1 + \alpha_2)\dots) + \alpha_i) + \alpha_{i+1} \rrbracket \\ &\leq \llbracket k_{i+1}(k_i(\dots(k_2 + 1)\alpha_1 + \alpha_2)\dots) + \alpha_1 + \alpha_{i+1} \rrbracket \\ &\leq \llbracket (k_{i+1} + 1)(k_i + 1)\dots(k_2 + 1)\alpha_1 \rrbracket \\ &\leq (k_{i+1} + 1)(k_i + 1)\dots(k_2 + 1)\llbracket \alpha_1 \rrbracket \\ &\leq (k_{i+1} + 1)(k_i + 1)\dots(k_2 + 1)\llbracket f(x) \rrbracket \\ &\leq (n_{i+1}n_i \dots n_1)^4 |f(x)|. \end{aligned}$$

Note that $(n_{i+1}n_i \dots n_1)^4 < n_{i+1}^{i+1} < n_{i+1}^{n_i+1}$. We also know that the degree of β_{i+1} over $Q(\beta_i)$ is n_{i+1} . Now β_{i+1} is an algebraic integer, since it is the sum and product of various algebraic integers. The coefficients of the minimal polynomial for β_{i+1} are bounded by $(n_{i+1}n_i \dots n_1)^{4n_i+1} |f(x)|^{n_{i+1}}$ which is less than $n_{i+1}^{4n_i+1^2}$. Thus they require at most $4n_{i+1}^2 \log n_{i+1} + n_{i+1}^2 \log |f(x)|$ bits to write down. Thus (2) is proved.

Now if $f_{i+2}(x)$ is an irreducible factor of $f(x)$ in $Q(\beta_{i+1})$, then $f_{i+2}(x) = \prod_{i_1, \dots, i_j} (x - \alpha_i)$. If $f_{i+2}(x) = x^r + \gamma_{r-1}x^{r-1} + \dots + \gamma_0$, then $\llbracket \gamma_i \rrbracket < 2^r \llbracket f(x) \rrbracket^r$. We wish however, to bound the integer coefficients of $f_{i+2}(x)$, where $\gamma_i = \sum_{j=0}^{n_{i+1}-1} c_{ij}\beta_{i+1}^j$; to do so we apply Weinberger–Rothschild. Since $f_{i+2}(x)$ is a divisor of $f(x)$ in $Q(\beta_{i+1})$, we

have $|c_{ij}| \leq \|f(x)\| (n_{i+1} h_{i+1}(t))^{n_i+1}$, where $h_{i+1}(t)$ is the minimal polynomial for β_{i+1} over Q . Observations of the previous paragraph demonstrate that the coefficients of $h_{i+1}(t)$ are polynomially bounded in size, and that the number of bits needed to write down c_{ij} is polynomial in $(n_{i+1}, \log |f(x)|)$.

Finally we observe that the arguments used in part (2) of the proof of the main theorem combined with the discussion on primitive elements yields a proof of (1). \square

A polynomial of degree n over Q usually has splitting field of degree $n!$ over Q , so Corollary 6 gives a bound which is exponential in n and $|f(x)|$. But there are many subtleties about Galois groups to be exploited which indicate Corollary 6 can be greatly improved. We include Corollary 6 in order to emphasize the fact that the ability to factor has put many tantalizing problems within our grasp. We can test normality. We can compute the Galois group if it is abelian. We can even test primitivity [LM]. All of these can be done in polynomial time.

Appendix. We include here the proofs of Theorem 1.3 and Corollary 1.8, as well as Algorithm 2.1.

THEOREM 1.3 (Weinberger and Rothschild). *Let β be a root of $f(x) \in Z[\alpha][x]$, notation as above. Then $\|\beta\| \leq 1 + \|f(x)\|$. Assume that $f(x)$ is monic, and let*

$$h(x) = h_r x^r + h_{r-1} x^{r-1} + \dots + h_0$$

be a factor of $f(x)$ in $(1/d)Z[\alpha][x]$ which is primitive. If $h_i = (1/d)(c_{im-1} \alpha^{m-1} + \dots + c_{i0})$, then $|c_{ij}| \leq d \binom{m}{i} \|f(x)\|(t)^m \text{disc}(g(t))^{-1/2}$.

Proof. It is not difficult to see that if α and γ are algebraic numbers, then $\|\alpha + \gamma\| \leq \|\alpha\| + \|\gamma\|$, and that $\|\alpha\gamma\| \leq \|\alpha\| \|\gamma\|$. We have noted previously that for α a root of $g(x)$, $\|\alpha\| \leq 1 + \max_i |a_i| \leq 1 + |g(t)|$. A similar argument shows that $\|\beta\| \leq 1 + \max_i \|\beta_i\| \leq 1 + \|f(x)\|$.

Suppose $h(x)|f(x)$ in $Q(\alpha)[x]$. By Proposition 1.1, $h(x) \in (1/d)Z[\alpha][x]$. Now $h(x) = x^i + h_{i-1} x^{i-1} + \dots + h_0 = \prod_{i \in S} (x - \beta_i)$, for some $S \subseteq \{1, \dots, n\}$. Then $\|h_i\| \leq (i) \|f(x)\|$ by [Mi1.] Now we seek to bound the integer coefficients of h_i .

If $\gamma \in Q(\alpha)$, $\gamma = \sum_{j=0}^{m-1} r_j \alpha^j$, $r_j \in Q$. Define $\gamma_i = \sum_{j=0}^{m-1} r_j \alpha_i^j$, and define a map $L: C^n \rightarrow C^n$ by $L(r_0, \dots, r_{m-1}) = (\gamma_1, \dots, \gamma_m)$. Note that this map is invertible and linear. It is invertible because it is a Vandermonde matrix formed from $\alpha_1 \dots \alpha_m$. We have $\det(L) = \text{disc}(g(t))^{1/1}$. Let $|\gamma|_\infty = \max_i |\gamma_i|$, and $|r|_\infty = \max_i |r_i|$. Since all of the $r_i \in Q$, $\gamma \in Q(\alpha)$, and $|\gamma|_\infty = \| \gamma \|$. The action of L is multiplication by a matrix, which, by abuse-of-notation, we also call L , $rL = \gamma$. Thus $r = \gamma L^{-1}$, and $|r|_\infty \leq |\gamma|_\infty |L^{-1}|_\infty$, where $|L^{-1}|_\infty = \max_j (\sum_{i=1}^m |l_{ij}|)$. If $r_j = c_j/d$, then $|c_j| < d \binom{m}{i} \|f(x)\| |L^{-1}|_\infty$.

Next we bound $|L^{-1}|_\infty$. The entries of L^{-1} are cofactors of L , divided by the $\text{disc}(g(t))^{1/2}$. By Hadamard's inequality, we obtain an upper bound of

$$\prod_{j=1}^{m-1} \left(\sum_{i=1}^{m-1} |\alpha_j|^{2i} \right)^{1/2}$$

for the determinant. Then:

$$\prod_{|\alpha_j| \leq 1} (m-1)^{1/2} \prod_{|\alpha_j| > 1} (m-1)^{1/2} |\alpha_j|^{m-1} = (m-1)^{(m-1)/2} \left(\prod_{|\alpha_j| > 1} |\alpha_j|^{m-1} \right).$$

Recall that $g(t)$ is monic; by Mignotte we find that $\prod_{|\alpha_j| \leq 1} |\alpha_j| \leq |g(t)|$, so that $(m-1)^{(m-1)/2} |g(t)|^{m-1} \text{disc}(g(t))^{-1/2}$ bounds the absolute value of the entries of L . Therefore

$$|L^{-1}|_\infty \leq m(m-1)^{(m-1)/2} |g(t)|^{m-1} |\text{disc}(g(t))|^{-1/2}.$$

Thus

$$|c_{ij}| \leq d \binom{m}{i} \ll f(x) m(m-1)^{(m-1)/2} |g(t)|^{m-1} \text{disc}(g(t))^{-1/2}.$$

COROLLARY 1.8. *Let α satisfy a monic irreducible polynomial $g(t)$ over Z of degree m and with discriminant d . Let $f_1(x), f_2(x)$ be polynomials over $O_K, K = Q[t]/g(t)$ with maximum degree n and $\ll f_1(x), \ll f_2(x)$ bounded by $\ll f(x)$. Then $\text{gcd}_K(f_1(x), f_2(x))$ can be computed in $O(n^7 m^2 \log^2 |g(t)| \log^2 \ll f(x))$ steps.*

Proof. We assume that $f_1(x), f_2(x)$ are primitive. In the gcd algorithm presented by Brown [Br2], he computes polynomials $G_i(x)$, where $G_1(x) = f_1(x), G_2(x) = f_2(x)$,

$$G_3(x) = (-1)^{\delta_1+1} \text{pseudoremainder}(G_1(x), G_2(x))$$

and

$$G_i(x) = \frac{(-1)^{\delta_{i-2}+1} \text{pseudoremainder}(G_{i-2}(x), G_{i-1}(x))}{g_{i-2} h_{i-2}^{\delta_{i-2}}},$$

with δ_i being the degree of $G_i(x)$, g_i the leading coefficient of $G_i(x)$, and $h_i = g_i^{\delta_{i-1}} h_{i-1}^{1-\delta_{i-1}}$.

We perform the same algorithm with a minor modification. Each time we compute the pseudoremainder $G_i(x)$, we reduce the coefficients modulo $(g(t))$. Then $S(l, m)$, the size of a product of m such determinants, each of order l , changes accordingly; we find:

$$S(l, k) = lk \log \ll f(x) (n + m \log |g(t)|).$$

Then we find that:

$$C_i^{(1)} < 2(\delta_{i-2} + 1)n_{i-2}S(l_{i-1}, 1)S(l_i, \delta_{i-2} + 2)$$

and

$$C_i^{(2)} < (n_i + 1)S(l_i, 1)S(l_{i-1}, \delta_{i-2} + 1).$$

Then

$$C_i < 2(2\delta_{i-2} + 3)n_{i-2}S(l_{i-1}, 1)S(l_i, \delta_{i-2} + 2).$$

This implies that $C < \text{constant} \times n^7 m^2 \log^2 |g(t)| \log^2 \ll f(x)$, which was to be shown. \square

ALGORITHM 2.1. FACTOR.

input: $g(t) \in Z[t]$, monic, irreducible
 $f(x) \in Q[x, t]; f(x)$ with coefficients in $O_K, K = Q[t]/(g(t))$

Step 1. $c \leftarrow 1$
 $j \leftarrow 0$
 $k(x) \leftarrow \text{gcd}_{O[t]/g(t)}(f(x), f'(x))$
 $h(x) \leftarrow f(x)/k(x)$

Step 2. $l(x) \leftarrow \text{Res}_t(g(t), h(x-ct))$
 While $(\text{gcd}(l(x), l'(x)) \neq 1)$, do:
 $c \leftarrow c + 1$
 $l(x) \leftarrow \text{Res}_t(g(t), h(x-ct))$

Step 3. Factor $l(x) = \prod_{i=1}^r F_i(x)$

Step 4. For $i = 1, \dots, r$, do:
 $f_i(x) \leftarrow \text{gcd}_{O[t]/g(t)}(F_i(x+ct), h(x))$

Step 5. If $(k(x) = 1)$, then return $\{f_i(x)\}, c(t)$
 Else for $i = 1, \dots, r$, do:
 While $\gcd(F_i(x + ct), k(x)) \neq 1$, do:
 $j \leftarrow j + 1$
 $f_{j+r}(x) \leftarrow \gcd(F_i(x + ct), k(x))$
 $k(x) \leftarrow k(x) / f_{j+r}(x)$
return: $\{f_i(x)\}$, where $f_i(x)$ is irreducible over $O_K[x]$, where $K = Q[t]/g(t)$,
 and $f(x) = \prod_{i=1}^{j+r} f_i(x)$

Acknowledgments. Warm thanks to Rich Zippel, in whose class this result was first observed, and who has tirelessly answered my many questions. Conversations with Arjen Lenstra and Michael Ben-Or led to improved bounds in Weinberger–Rothschild (Theorem 1.3), and the main algorithm respectively. Gary Miller contributed to the proof of Corollary 6.

REFERENCES

- [B-O] M. BEN-OR, *Probabilistic algorithms in finite fields*, Proc. Twenty-second Annual IEEE Symposium on Foundations of Computer Science, 1981, pp. 394–398.
- [Be] E. R. BERLEKAMP, *Algebraic Coding Theory*, McGraw-Hill, New York, 1968.
- [Be2] ———, *Factoring polynomials over large finite fields*, in Math. Comp., 24, 1970, pp. 713–735.
- [Br1] W. S. BROWN, *On Euclid’s algorithm and the computation of polynomial greatest common divisors*, J. Assoc. Comput. Mach., 18 (1971), pp. 478–504.
- [Br2] ———, *The subresultant PRS algorithm*, ACM Trans. Math. Software, 4 (1978), pp. 241–249.
- [BT] W. S. BROWN AND J. F. TRAUB, *On Euclid’s algorithm and the theory of subresultants*, J. Assoc. Comput. Mach., 18 (1971), pp. 505–514.
- [Can] D. G. CANTOR, *Irreducible polynomials with integral coefficients have succinct certificates*, J. Algorithms, to appear.
- [CG] A. L. CHISTOV AND D. Y. GRIGORIEV, *Polynomial-time factoring of the multivariable polynomials over a general field*, USSR Academy of Sciences, Steklov Mathematical Institute, Leningrad, 1982.
- [Co] G. COLLINS, *The calculation of multivariate polynomial resultants*, J. Assoc. Comput. Mach., 18 (1971), pp. 515–532.
- [Edm] J. EDMONDS, *Systems of distinct representations and linear algebra*, J. National Bureau of Standards, Ser. B, 71B (1967), pp. 241–5.
- [Ka1] E. KALTOFEN, *A polynomial reduction from multivariate to bivariate polynomial factorization*, Proc. Fourteenth Annual ACM Symposium on Theory of Computing, 1982, pp. 261–266.
- [Ka2] ———, *A polynomial-time reduction from bivariate to univariate integral polynomial factorization*, Proc. Twenty-third Annual IEEE Symposium on Foundations of Computer Science (1982), pp. 57–64.
- [KMS] E. KALTOFEN, D. R. MUSSER AND B. D. SAUNDERS, *A generalized class of polynomials which are hard to factor*, Proc. 1981 ACM Symposium on Symbolic and Algebraic Computation, pp. 188–194.
- [Kn] D. KNUTH, *The Art of Computer Programming, Vol. II: Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 1969.
- [Kr] L. KRONECKER, *Gründzüge Einer Arithmetischen Theorie der Algebraischen Grössen*, Druck und Verlag von G. Riemeier, Berlin, 1882.
- [LM] S. LANDAU AND G. L. MILLER, *Solvability by radicals is in polynomial time*, Proc. Fifteenth Annual ACM Symposium on Theory of Computing, 1983, pp. 140–151.
- [L³] A. K. LENSTRA, H. W. LENSTRA AND L. LOVASZ, *Factoring polynomials with rational coefficients*, Tech. Report 82–05, Dept. Mathematics, Univ. Amsterdam, 1982.
- [AKL] A. K. LENSTRA, *Factoring polynomials over algebraic number fields*, Tech. Report IW213/82, Dept. Computer Science, Stichting Mathematisch Centrum, Amsterdam, 1982.
- [Lpc] H. W. LENSTRA, *private communication*.
- [Mar] D. MARCUS, *Number Fields*, Springer-Verlag, New York, 1977.
- [Mi1] M. MIGNOTTE, *An inequality about factors of polynomials*, Math. Comp., 28 (1974), pp. 1153–1157.

- [Mi2] M. MIGNOTTE, *Some inequalities about univariate polynomials*, Proc. 1981 ACM Symposium on Symbolic and Algebraic Computation, pp. 195–99.
- [Mu] D. R. MUSSER, *Multivariate polynomial factorization*, J. Assoc. Comput. Mach., 22 (1975), pp. 291–308.
- [Sa] P. SAMUEL, *Algebraic Theory of Numbers*, Kershaw Publishing, London, 1972.
- [Tr] B. TRAGER, *Algebraic factoring and rational function integration*, Proc. 1976 ACM Symposium on Symbolic and Algebraic Computation, pp. 219–226.
- [vdW] B. L. VAN DER WAERDEN, *Modern Algebra*, Frederick Ungar, New York, 1941.
- [Wei] P. J. WEINBERGER, *Finding the number of factors of a polynomial*, unpublished manuscript.
- [WR] P. WEINBERGER AND L. ROTHSCILD, *Factoring polynomials over algebraic number fields*, J. Assoc. Comput. Mach, (1976), pp. 335–350.
- [Weyl] H. WEYL, *Algebraic Theory of Numbers*, Princeton Univ. Press, Princeton, NJ, 1940.
- [Za1] H. ZASSENHAUS, *On Hensel factorization*, I, J. Number Theory, 1 (1969), pp. 291–311.

THE COMPUTATIONAL COMPLEXITY OF SIMULTANEOUS DIOPHANTINE APPROXIMATION PROBLEMS*

J. C. LAGARIAS†

Abstract. Simultaneous Diophantine approximation in d dimensions deals with the approximation of a vector $\alpha = (\alpha_1, \dots, \alpha_d)$ of d real numbers by vectors of rational numbers all having the same denominator. This paper considers the computational complexity of algorithms to find good simultaneous approximations to a given vector α of d rational numbers. We measure the goodness of an approximation using the sup norm. We show that a result of H. W. Lenstra, Jr. produces polynomial-time algorithms to find sup norm best approximations to a given vector α when the dimension d is fixed. We show that a recent algorithm of A. K. Lenstra, H. W. Lenstra, Jr. and L. Lovasz to find short vectors in an integral lattice can be used to find a good approximation to a given vector α in d dimensions with a denominator Q^* satisfying $1 \leq Q^* \leq 2^{d/2} N$ which is within a factor $\sqrt{5d} 2^{(d+1)/2}$ of the best approximation with denominator Q with $1 \leq Q \leq N$. This algorithm runs in time polynomial in the input size, independent of the dimension d .

We also prove complementary results showing that certain natural simultaneous Diophantine approximation problems are NP-hard. We show that the problem of deciding whether a given vector α of rational numbers has a simultaneous approximation of specified accuracy with respect to the sup norm with denominator Q in a given interval $1 \leq Q \leq N$ is NP-complete. (Here the dimension d is allowed to vary.) We prove two other complexity results, which suggest that the problem of locating best (sup norm) simultaneous approximations is harder than this NP-complete problem.

Key words. NP-completeness, simultaneous Diophantine approximation, public key cryptography, lattices, integer programming

1. Introduction. Simultaneous Diophantine approximation is the study of the approximation properties of a vector of several real numbers $\alpha = (\alpha_1, \dots, \alpha_d)$ by a vector of rational numbers $\mathbf{v} = (p_1/q, \dots, p_d/q)$ all having the same denominator. Here d is called the *dimension* of the problem. There has been considerable recent work on algorithms to locate such approximations ([2], [4], [14], [15], [17]). Such algorithms have a number of applications, e.g. to factor univariate polynomials with rational coefficients [22] and to find units in number fields [2]. In addition, good one-dimensional Diophantine approximations ([1], [6], [21]) and simultaneous Diophantine approximations ([12], [20]) have been used in breaking several proposed public key cryptosystems of knapsack type.

This paper studies the computational complexity of finding good simultaneous Diophantine approximations. We first describe how recent results of H. W. Lenstra, Jr. [15] and A. K. Lenstra, H. W. Lenstra, Jr. and L. Lovasz [14] give fast algorithms for finding good simultaneous approximations. Then we complement these results by proving that certain natural simultaneous Diophantine approximation problems are NP-hard.

We use the sup norm to measure the goodness of a simultaneous Diophantine approximation. The *sup norm* measure $\{\{q\alpha\}\}$ for the degree of approximation to α by rationals with denominator q is given by

$$\{\{q\alpha\}\} = \max_{1 \leq i \leq d} \{q\alpha_i\}$$

* Received by the editors June 11, 1982, and in final form October 10, 1983. A preliminary version of this paper appeared in the Proceedings of the 23rd Annual IEEE Conference on Foundations of Computer Science, 1982.

† AT & T Bell Laboratories, Murray Hill, New Jersey 07974.

where $\{\beta\}$ denotes the distance to the nearest integer, i.e.

$$\{\beta\} = \min_{n \in \mathbf{Z}} |\beta - n|.$$

We associate to a vector α its (*sup norm*) *best simultaneous approximation denominators* (BSAD's), which are exactly those q for which $\{\{q\alpha\}\}$ is smaller than any of the $\{\{q'\alpha\}\}$ with $1 \leq q' < q$.

In order to obtain a well posed computational problem, the vector α must be described by a finite input. This constraint leads us to consider α which are vectors of rational numbers, i.e., $\alpha = (a_1/b_1, \dots, a_d/b_d)$ where the a_i are integers and the b_i are positive integers. Here the $\{a_i, b_i: 1 \leq i \leq d\}$ are the input data describing α .

For a long time it has been known that the ordinary continued fraction algorithm provides a fast method of finding the best approximations to a single real number α . No such algorithms were known which are guaranteed to find even relatively good approximations in any dimension $d \geq 2$ until very recently, cf. [2], [4], [17]. In 1981, H. W. Lenstra, Jr. [15] found a polynomial-time algorithm for solving integer programming problems in a fixed number of variables. This easily implies the following result.¹

THEOREM A. *For any fixed dimension d , there exist algorithms to solve the following two problems whose worst case running time is bounded by a polynomial in the length of the input.*

(1) FINDING A GOOD APPROXIMATION.

INPUT. A d -dimensional vector of rational numbers $\alpha = (a_1/b_1, \dots, a_d/b_d)$ and positive integers N, s_1 and s_2 .

OUTPUT. A denominator Q with $1 \leq Q \leq N$ such that $\{\{Q\alpha\}\} \leq s_1/s_2$, provided at least one exists, and $Q = 0$ otherwise.

(2) FINDING BEST APPROXIMATIONS.

INPUT. A d -dimensional vector of rational numbers $\alpha = (a_1/b_1, \dots, a_d/b_d)$ and a positive integer N .

OUTPUT. A complete list of all best simultaneous approximation denominators Q to α for which $Q \leq N$.

The running time bounds for these algorithms increase as the dimension increases. Lenstra's running time bound for problems with input length L is of the form $O(L^{c(d)})$, where $c(d)$ grows exponentially with d .

More recently A. K. Lenstra, H. W. Lenstra, Jr. and L. Lovasz (see [14]) found a polynomial time algorithm, which we will call the L^3 -algorithm, for finding short vectors in a lattice $\Lambda \subseteq \mathbf{Z}^d$. An important feature of the L^3 -algorithm is that it has a running time polynomial in the input size, even when the dimension d of the lattice is allowed to vary. Lenstra, Lenstra and Lovasz observed that the L^3 -algorithm can be used to find simultaneous Diophantine approximations nearly as good as those guaranteed to exist by Dirichlet's theorem.

DIRICHLET'S THEOREM. *For any vector $\alpha \in \mathbf{R}^d$ and any positive integer N there exists a denominator q with $1 \leq q \leq N^d$ such that*

$$\{\{q\alpha\}\} \leq N^{-1}.$$

In particular, for $\alpha \in \mathbf{Q}^d$ Lenstra, Lenstra and Lovasz [14, Prop. 1.39] showed that the L^3 -algorithm can be directly used to find a denominator q with $1 \leq q \leq$

¹ Part (1) of Theorem A has been proved independently by A. Shamir and P. van Emde Boas.

$2^{n(n+1)/4}N^d$ such that

$$\{\{q\alpha\}\} \leq N^{-1}.$$

We prove the following result, which shows that one can find in polynomial time a simultaneous approximation denominator Q^* for $\alpha \in \mathbf{Q}^d$ with $1 \leq Q^* \leq 2^{d/2}N$ which has $\{\{Q^*\alpha\}\}$ within a constant factor (depending on d) of $\{\{Q\alpha\}\}$ for the best possible denominator Q with $1 \leq Q \leq N$.

THEOREM B. *Let $\alpha = (a_1/b_1, \dots, a_d/b_d)$ be given, and for a given positive integer N set*

$$\delta_N = \delta_N(\alpha) = \min_{1 \leq q \leq N} \{\{q\alpha\}\}.$$

There is an algorithm which when given α and N as input will produce a denominator Q^ with $1 \leq Q^* \leq 2^{d/2}N$ such that*

$$\{\{Q^*\alpha\}\} \leq \sqrt{5d} 2^{(d-1)/2} \delta_N.$$

This algorithm runs to completion in $O(d^6(d \log_2 M + \log_2 N)^4)$ bit operations, where $M = \max\{|a_i|, |b_i|: 1 \leq i \leq d\}$.

Note that if the input to the algorithm of Theorem B is $\{(a_i, b_i): 1 \leq i \leq d\}$ together with N , expressed as integers in binary, then the input length Σ satisfies $\Sigma \geq \max(d, \log_2 M, \log_2 N)$ so that the running time to the algorithm is at most $O(\Sigma^{14})$ bit operations.

Theorem B follows by applying the L^3 -algorithm to an appropriately chosen series of lattices in \mathbf{Z}^{d+1} . We give the precise details in § 2.

Now we describe the main results of this paper, which are complementary results showing that certain natural simultaneous Diophantine approximation problems are NP-hard. We consider the following set recognition problems.

(1) **GOOD SIMULTANEOUS APPROXIMATION (GSA).**

INSTANCE. A finite vector of rationals $\alpha = (a_1/b_1, \dots, a_d/b_d)$ and positive integers N, s_1 , and s_2 .

QUESTION. Is there an integer Q with $1 \leq Q \leq N$ such that $\{\{Q\alpha\}\} \leq s_1/s_2$?

(2) **LARGEST BEST SIMULTANEOUS APPROXIMATION (LBSA).**

INSTANCE. A finite vector of rationals $\alpha = (a_1/b_1, \dots, a_d/b_d)$ and positive integers Q, N with $Q \leq N$.

QUESTION. Is Q the largest best simultaneous approximation denominator to α with $Q < N$?

(3) **BEST SIMULTANEOUS APPROXIMATION IN AN INTERVAL (BSAI).**

INSTANCE. A finite vector of rational numbers $\alpha = (a_1/b_1, \dots, a_d/b_d)$ and positive integers N_1, N_2 with $N_1 \leq N_2$.

QUESTION. Is there a best simultaneous approximation denominator Q with $N_1 \leq Q \leq N_2$?

In these problems we allow the input to vary over all dimensions d . If the dimension d were fixed, Theorem A implies all three problems could be solved in polynomial time.

Our main result is as follows.

THEOREM C. *GOOD SIMULTANEOUS APPROXIMATION is NP-complete.*

Theorem C is proved by a method inspired by Manders and Adleman [16]. We use Theorem C to derive two consequences, concerning the problem of locating best simultaneous approximations.

THEOREM D. LARGEST BEST SIMULTANEOUS APPROXIMATION is in co-NP. If $NP \neq \text{co-NP}$ then it is not in NP.

This result implies that one cannot improve the algorithm of Theorem B to find the best sup norm approximation in an interval $[1, 2^{d/2}N]$ (instead of merely a good one satisfying (1.1)) while retaining the polynomial running time, unless $NP = \text{co-NP}$. Theorem D is proved by means of a nondeterministic conjunctive truth table reduction of GSA^c to LBSA.

THEOREM E. BEST SIMULTANEOUS APPROXIMATION IN AN INTERVAL is in the polynomial hierarchy complexity class Δ_2^P . It is NP-hard.

Theorems D and E together suggest that the problem of locating best simultaneous approximations is computationally harder than that of locating good simultaneous approximations.

In passing, we note that GSA problems are instances of a special type of integer programming problem.

INTEGER PROGRAMMING WITH AT MOST TWO VARIABLES IN EACH CONSTRAINT (2-IP).

INSTANCE. Given a finite set of integer vectors $\{(a_{1,k}, a_{2,k}, a_{3,k}) : 1 \leq k \leq M\}$, and two functions $\sigma, \tau : \{1, 2, \dots, M\} \rightarrow \{1, 2, \dots, N\}$.

QUESTION. Does the integer program $a_{1,k}x_{\sigma(k)} + a_{2,k}x_{\tau(k)} \leq a_{3,k}, 1 \leq k \leq M$ have an integer feasible solution $\{x_i : 1 \leq i \leq N\}$?

Theorem B immediately implies the following NP-completeness result.²

THEOREM F. 2-IP is NP-complete.

Theorems C through F are proved in § 3.

We propose two conjectures for further work. These conjectures, if true, would more precisely delimit the computational complexity of finding good approximations and best approximations.

CONJECTURE 1. If $NP \neq \text{co-NP}$, then BSAI is not in $NP \cup \text{co-NP}$.

CONJECTURE 2. Let $f(d)$ be a polynomial in d . Let $\alpha = (a_1/b_1, \dots, a_d/b_d)$ and N be given and let $\delta_N = \min_{1 \leq q \leq N} \{q\alpha\}$. Suppose there exists an algorithm which when given α and N as input will produce a denominator Q^* with $1 \leq Q^* \leq f(d)N$ such that

$$\{\{Q^*\alpha\}\} \leq f(d)\delta_N$$

whose running time is bounded by a polynomial of the input size not depending on d . Then $P = NP$.

2. Polynomial time algorithms for finding good approximations.

Proof of Theorem A. The problem FINDING A GOOD APPROXIMATION can be formulated as an integer programming problem in $d + 1$ variables and $2d + 2$ constraints:

$$(2.1) \quad -s_1B \leq s_2 \left(\frac{a_i B}{b_i} y - Bx_i \right) \leq s_1B, \quad 1 \leq i \leq d$$

where $B = b_1 b_2 \dots b_d$, together with

$$1 \leq y \leq N.$$

Then solve this integer program using H. W. Lenstra Jr.'s algorithm [15].

To solve FINDING BEST APPROXIMATIONS, we first describe an algorithm which finds the largest BSAD q to α in the interval $1 \leq q \leq N$, where N is given. We

²I owe this observation to R. Kaltofen.

let $[s, M_1, N]$ denote the integer programming problem

$$(2.2a) \quad -s \leq \frac{a_i B}{b_i} y - x_i \leq s,$$

$$(2.2b) \quad M_1 \leq y \leq N,$$

where (2.2a) is obtained from (2.1) by setting $s_2 = B$. We use H. W. Lenstra, Jr.'s algorithm [15] to solve a series of problems of the form $[s, M_1, N]$, varying s in the interval $1 \leq s \leq B$, using a bisection strategy to find the minimal $s = s_0$ for which the system (2.2) is solvable. This takes at most $O(\log B)$ iterations. Then we fix $s = s_0$ and vary M_1 , using a bisection strategy to locate the largest M_1 for which the system (2.2) is solvable. This takes at most $O(\log N)$ iterations. The value of M_1 obtained is the desired BSAD q . Second, we use this algorithm to successively locate one by one the largest BSAD not yet found, until a complete list of BSAD's is produced. There are at most $O(2^{d+1} \log B)$ BSAD's in this list, using [11, Thm. 2.2] (see also [10]). Consequently this algorithm halts in polynomial time. \square

Proof of Theorem B. The L^3 algorithm finds a short vector $\mathbf{x} \neq \mathbf{0}$ in a lattice $\Lambda \subseteq \mathbf{Z}^d$ where a basis $\mathbf{v}_1, \dots, \mathbf{v}_d$ of the lattice is given as input. This algorithm is described in [14], where the following results are proved ([14, Props. 1.11, 1.26]). Here $\|\mathbf{x}\|$ denotes the Euclidean length of a vector \mathbf{x} , i.e. if $\mathbf{x} = (x_1, \dots, x_d)$ then

$$\|\mathbf{x}\|^2 = x_1^2 + x_2^2 + \dots + x_d^2.$$

PROPOSITION 2.1. (i) *The L^3 algorithm produces a short vector $\mathbf{x} \neq \mathbf{0}$ which satisfies*

$$(2.3) \quad \|\mathbf{x}\|^2 \leq 2^{d-1} \|\mathbf{w}\|^2,$$

for all nonzero $\mathbf{w} \in \Lambda$.

(ii) *Let the largest entry in the input basis $\langle \mathbf{v}_1, \dots, \mathbf{v}_d \rangle$ of Λ have absolute value L . Then the L^3 algorithm produces a short vector \mathbf{x} in $O(d^6(\log L)^3)$ bit operations.*

We consider the following algorithm. Let $\alpha = (a_1/b_1, \dots, a_d/b_d)$.

Step 1. Determine whether or not there is some q^* with $1 \leq q^* \leq N$ such that $\{q^* \alpha\} = 0$. If so, halt. If not, proceed to Step 2.

Step 2. Let $B = b_1 b_2 \dots b_d$. For each j in the range $1 \leq j \leq d + \log B + \log N$ apply the L^3 algorithm to a lattice $\Lambda_j \subseteq \mathbf{Z}^{d+1}$ where $\Lambda_j = \langle \mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_d, \mathbf{v}_{d+1}^{(j)} \rangle$ where

$$\mathbf{v}_i = (0, \dots, 0, NB, 0, \dots, 0)$$

is nonzero only in the i th coordinate, for $1 \leq i \leq d$, and

$$\mathbf{v}_{d+1}^{(j)} = \left(BN \frac{a_1}{b_1}, \dots, BN \frac{a_d}{b_d}, 2^j \right).$$

Let $\mathbf{x}^{(j)} = (x_1^{(j)}, \dots, x_{d+1}^{(j)})$ be the short vector in Λ_j produced by the L^3 algorithm. Set $q_j^* = x_{d+1}^{(j)}$ and check whether or not

$$1 \leq q_j^* \leq 2^{d/2} N.$$

If so, call q_j^* *admissible* and compute

$$\eta_j = \max_{1 \leq i \leq d} |x_i^{(j)}|.$$

Step 3. Select an admissible q_j^* for which η_j is minimal and halt.

We show that this algorithm has the required properties. We first prove the algorithm is correct. We must show that whenever the algorithm reaches Step 2, it then produces at least one admissible q_j^* for which

$$\{\{q_j^* \alpha\}\} \leq \sqrt{5d} 2^{(d-1)/2} \delta_N.$$

Assume we are in Step 2, and let q be a denominator which minimizes $\{\{q \alpha\}\}$ for $1 \leq q \leq N$, so that $\delta_N = \{\{q \alpha\}\}$. Then $\delta_N > 0$ since the algorithm did not halt at Step 1, so that

$$(2.4) \quad \delta_N \geq \frac{1}{B}.$$

Now consider that k for which

$$(2.5) \quad 2^{2k-2} q^2 \leq dB^2 N^2 \delta_N < 2^{2k} q^2.$$

We claim that

$$(2.6) \quad 1 \leq k \leq d + \log B + \log N.$$

Using (2.4) we have

$$dB^2 N^2 \delta_N \geq dN^2 \geq q^2,$$

so that (2.5) forces $k \geq 1$. The right side of (2.6) follows similarly from the condition $\delta_N \leq 1$.

Now (2.6) shows that $\Lambda = \Lambda_k$ is examined in Step 2 of the algorithm. Let $\mathbf{x} = \mathbf{x}^{(k)}$ be the vector produced by the L^3 algorithm applied to Λ_k . We may write

$$\mathbf{x} = -p_1^* \mathbf{v}_1 - p_2^* \mathbf{v}_2 - \dots - p_d^* \mathbf{v}_d + q^* \mathbf{v}_{d+1}^{(k)}.$$

If $\mathbf{x} = (x_1, \dots, x_{d+1})$, then

$$x_i = BN \left(q^* \frac{a_i}{b_i} - p_i^* \right)$$

for $1 \leq i \leq d$, hence

$$\max_{1 \leq i \leq d} |x_i| \geq \{\{q^* \alpha\}\} BN.$$

Consequently

$$(2.7) \quad \|\mathbf{x}\|^2 \geq \{\{q^* \alpha\}\}^2 B^2 N^2 + 2^{2k} (q^*)^2.$$

Let p_i be the nearest integer to $q(a_i/b_i)$ and consider the vector

$$\mathbf{w} = -(p_1 \mathbf{v}_1 + \dots + p_d \mathbf{v}_d) + q \mathbf{v}_{d+1}^{(k)}$$

in Λ . Now

$$|w_i| \leq BN \delta_N, \quad 1 \leq i \leq d,$$

and

$$|w_{d+1}| = 2^k q.$$

Hence

$$\|\mathbf{w}\|^2 \leq dB^2 N^2 \delta_N + 2^{2k} q^2.$$

Now (2.3) yields

$$\|\mathbf{x}\|^2 \leq 2^{d-1}(dB^2N^2\delta_N + 2^{2k}q^2).$$

Combining this with (2.5) and (2.7), we obtain the inequalities

$$(2.8) \quad (q^*)^2 \leq \frac{1}{2^{2k}}(2^{d-1}dB^2N^2\delta_N) + q^2 \leq (2^{d-1} + 1)q^2,$$

and

$$(2.9) \quad \{\{q^*\alpha\}\}^2 \leq 2^{d-1}d\delta_N + 2^{d-1+2k} \frac{q^2}{B^2N^2} \leq 5d2^{d-1}\delta_N.$$

Since $q \leq N$, (2.8) proves that \mathbf{x} is admissible. Finally (2.9) implies (1.1), so the algorithm is correct.

Next we bound the running time of the algorithm. To carry out Step 1, we observe that the least positive q^{**} such that $\{\{q^{**}\alpha\}\} = 0$ is

$$q^{**} = \text{l.c.m.} \left\{ \frac{b_i}{(a_i, b_i)} : 1 \leq i \leq d \right\}.$$

One can compute q^{**} in polynomial time by repeatedly using the Euclidean algorithm. Then check whether or not $q^{**} \leq N$. Using this procedure Step 1 requires at most $O(d(d \log M + \log N)^3)$ bit operations. Step 2 applies the L^3 algorithm to $O(d \log M + \log N)$ lattices. The lattice vectors in each Λ_j have maximum coordinate

$$L \leq 2^d M^{2d} N^2.$$

This bound together with Proposition 2.1 implies that the L^3 algorithm takes $O(d^6(d \log M + \log N)^3)$ bit operations applied to each Λ_j separately. Thus Step 2 requires $O(d^6(d \log M + \log N)^4)$ bit operations. Step 3 requires $O((d \log M + \log N)^2)$ bit operations. Consequently the algorithm halts after at most $O(d^6(d \log M + \log N)^3)$ bit operations. \square

3. NP-hardness of certain simultaneous Diophantine approximation problems.

Proof of Theorem C. Our goal is to show that the set

$$\text{GSA}^* = \left\{ (\alpha, N, s_1, s_2) : \text{There is some } Q \text{ with } 1 \leq Q \leq N \text{ for which } \{\{Q\alpha\}\} \leq \frac{s_1}{s_2} \right\}$$

is NP-complete. The set GSA^* is clearly in NP, since we may “guess” a good Q if one exists. We will, in fact, show that the subset

$$\text{GSA}_1^* = \left\{ (\alpha, N, s_2) : \text{There is some } Q \text{ with } 1 \leq Q \leq N \text{ for which } \{\{Q\alpha\}\} \leq \frac{1}{s_2} \right\}$$

is NP-complete. The NP-completeness of GSA^* follows since GSA_1^* is a polynomial time recognizable subset of GSA^* .

To prove GSA_1^* is NP-complete, we polynomial time many-one reduce (i.e. Karp-reduce) the NP-complete problem WEAK PARTITION to the problem GSA_1^* .
WEAK PARTITION.

INSTANCE. A finite vector (a_1, \dots, a_d) of integers.

QUESTION. Is the equation $\sum_{i=1}^d a_i x_i = 0$ solvable with all $x_i = -1, 0$ or 1 and with some $x_i \neq 0$?

WEAK PARTITION was shown to be NP-complete by Shamir [19], and later by van Emde Boas [3] and Rubin [18].

In the remainder of the proof, we first describe the reduction, then prove it is correct, and finally check that it runs to completion in polynomial time.

Reduction to WEAK PARTITION.

Step 1. We are given $\mathbf{a} = \{a_i: 1 \leq i \leq d\}$ and are asked to recognize those \mathbf{a} for which

$$(2.10a) \quad \sum_{j=1}^d a_j y_j = 0,$$

$$(2.10b) \quad y_j \in \{-1, 0, 1\}, \quad \text{not all zero,}$$

has a nonzero solution $y = (y_1, \dots, y_d)$.

We encode (2.10a) as a congruence. Set

$$(2.11a) \quad A = \sum_{j=1}^d |a_j|.$$

Next let p_0 be the smallest prime such that $p_0 \nmid a_1 \cdots a_d$. Define R by

$$(2.11b) \quad p_0^{R-1} \leq A < p_0^R.$$

Next we find a set of primes Q_1, \dots, Q_d and an integer T that satisfy the following size conditions.

SIZE CONDITIONS.

- (i) $Q_1 < Q_2 < \dots < Q_d$.
- (ii) $(Q_i, p_0 a_1 a_2 \cdots a_d) = 1$.
- (iii) $Q_1^{T-1} \geq 4(d+1)p_0^R$.
- (iv) $Q_d < 2^{1/T} Q_1$.

(We will prove such Q_1, \dots, Q_d and T can always be found in the analysis of the running time of this reduction.)

Using the Chinese remainder theorem, we find $(\theta_1, \dots, \theta_d)$ such that θ_j is the smallest positive integer satisfying

$$(2.12a) \quad \theta_j \equiv 0 \pmod{\left(\frac{Q_1 \cdots Q_d}{Q_j}\right)^T},$$

$$(2.12b) \quad \theta_j \equiv a_j \pmod{p_0^R},$$

$$(2.12c) \quad \theta_j \not\equiv 0 \pmod{Q_j}.$$

In fact, if θ_j^0 is the smallest positive solution to (2.12a), (2.12b), then

$$(2.13) \quad \theta_j = \begin{cases} \theta_j^0 & \text{if } \theta_j^0 \not\equiv 0 \pmod{Q_j}, \\ \theta_j^0 + p_0^R \left(\frac{Q_1 \cdots Q_N}{Q_j}\right)^T & \text{otherwise,} \end{cases}$$

since one or both of these satisfy (2.12c). It follows that the system (2.10) is equivalent to:

$$(2.14a) \quad \sum_{j=1}^d \theta_j y_j \equiv 0 \pmod{p_0^R},$$

$$(2.14b) \quad y_j \in \{-1, 0, 1\}, \quad \text{not all zero,}$$

in view of (2.11) and (2.12b).

Step 2. We next encode (2.14b) as a search over a bounded interval. We introduce a new variable Z defined by

$$(2.15) \quad Z = \sum_{j=1}^d \theta_j y_j.$$

We set

$$(2.16) \quad H = \sum_{j=1}^d |\theta_j|,$$

and note that (2.14b), (2.15) imply that

$$(2.17) \quad |Z| \leq H.$$

We define $B = (\prod_{j=1}^d Q_j)^T$ and observe that

$$(2.18) \quad H < \frac{1}{2}B,$$

using the inequalities

$$|\theta_j| \leq |\theta_j^0| + p_0^R \left(\frac{Q_p \cdots Q_d}{Q_j} \right)^T \leq 2p_0^R \left(\frac{Q_1 \cdots Q_d}{Q_j} \right)^T \leq \frac{1}{2(d+1)} B,$$

which follow from (2.12a), (2.12b), (2.13) and the size condition (iii) for T .

CLAIM 1. *The system (2.14) is solvable if and only if the following system (2.19) is solvable.*

$$(2.19a) \quad -H \leq Z \leq H,$$

$$(2.19b) \quad Z \equiv 0 \pmod{p_0^R},$$

$$(2.19c) \quad Z \equiv 0, \pm \theta_j \pmod{Q_j^T}, 1 \leq j \leq d,$$

$$(2.19d) \quad Z \neq 0.$$

(We defer verifying this claim to the proof of correctness.)

Step 3. We now convert (2.19) to a GSA problem. First, we find the unique θ_j^* satisfying

$$(2.20a) \quad \theta_j \theta_j^* \equiv 1 \pmod{Q_j^T},$$

$$(2.20b) \quad 1 \leq \theta_j^* < Q_j^T,$$

for $1 \leq j \leq d$. This is always possible by the relative primality condition (ii) of the size conditions. We choose as our GSA vector

$$(2.21a) \quad \alpha_0 = \frac{1}{p_0^R},$$

$$(2.21b) \quad \alpha_j = \frac{\theta_j^*}{Q_j^T}, \quad 1 \leq j \leq d,$$

and we choose

$$(2.21c) \quad s_2 = Q_1^T.$$

CLAIM 2. *The system (2.19) is solvable if and only if the following GSA problem (2.22) is solvable.*

$$(2.22a) \quad -H \leq Z \leq H,$$

$$(2.22b) \quad \{Z\alpha_j\} \leq \frac{1}{Q_1^T}, \quad 0 \leq j \leq d,$$

$$(2.22c) \quad Z \neq 0.$$

Proof of correctness. It suffices to verify the two claims of equivalence.

Claim 1. Suppose (2.14) has a solution. Then (2.17) implies (2.19a), (2.14a) implies (2.19b), (2.12a) and (2.14b) imply (2.19c). To show (2.19d) holds, note that (2.14b) guarantees that some $y_j \neq 0$. Then (2.12a), (2.12c) imply that

$$Z \equiv \theta_j y_j \neq 0 \pmod{Q_j}.$$

Hence $Z \neq 0$, so (2.19) is solvable.

Suppose (2.19) has a solution. We first show that the y_j can be uniquely recovered from the data (2.19). In fact, using (2.12a) and (2.15), we have

$$(2.23) \quad Z \equiv \theta_j y_j \pmod{Q_j^T}$$

hence we can recover $y_j \pmod{Q_j^T}$ using the invertibility of $\theta_j \pmod{Q_j}$ given by (2.12c). In particular, (2.19c) now implies

$$(2.24) \quad y_j \equiv -1, 0 \text{ or } +1 \pmod{Q_j^T}.$$

Next, the Chinese remainder theorem guarantees the system of congruences (2.19c) has exactly 3^d solutions in the interval

$$-\frac{1}{2}B < Z \leq \frac{1}{2}B,$$

since $B = (\prod_j Q_j)^T$. Using (2.18), we conclude that the system (2.19a) and (2.19c) has $\leq 3^d$ solutions. However, we can explicitly exhibit 3^d such solutions, i.e. those with all $y_j = -1, 0$ or $+1$, (see (2.16), (2.18)) provided we can show they are all distinct. And all 3^d of them are distinct by (2.23), for if $y_j^{(0)} \neq y_j^{(1)}$ then $Z^{(0)} \not\equiv Z^{(1)} \pmod{Q_j^T}$. Consequently, we have found all such solutions and they all have $y_j = -1, 0, 1$. Also (2.23) implies that $Z = 0$ if and only if all $y_j = 0$. Together with (2.19d), this proves (2.14b) holds. Finally, (2.19b) and (2.15) imply (2.14a). We have shown (2.14) is solvable in this case, and Claim 1 is proved.

Claim 2. Suppose (2.19) is solvable. Certainly (2.19a), (2.19d) imply (2.22a), (2.22c), respectively. Now (2.19b) forces $\{Z\alpha_0\} = 0$. And, using (2.12a), (2.19c), we must have

$$\{Z\alpha_j\} = \left\{ \frac{\theta_j \theta_j^* y_j}{Q_j^T} \right\} = \left\{ \frac{\pm y_j}{Q_j^T} \right\} = 0 \text{ or } \frac{1}{Q_j^T},$$

for $1 \leq j \leq d$. Since $1/Q_j^T \leq 1/Q_1^T$, (2.22b) follows. Hence, (2.22) is solvable.

Now suppose (2.22) is solvable. Then (2.19a), (2.19d) hold. Also the size condition (iii) for T implies that

$$\frac{1}{p_0^R} > \frac{1}{Q_1^T}.$$

Since (2.22b) holds for α_0 , this forces $\{Z\alpha_0\} = 0$ using (2.21a). Hence

$$Z \equiv 0 \pmod{p_0^R},$$

which is (2.19b). Next we use the size condition (iv) for T ,

$$\frac{2}{Q_j^T} > \frac{1}{Q_1^T},$$

which together with $\{Z\alpha_j\} \leq 1/Q_1^T$ forces $\{Z\alpha_j\} = 0$ or $1/Q_j^T$ since α_j has denominator Q_j^T , for $1 \leq j \leq d$. This can only occur if

$$Z \equiv 0, \pm \theta_j \pmod{Q_j^T},$$

for $1 \leq j \leq d$, which is (2.19c). Thus (2.19) is solvable and Claim 2 is proved. \square

Polynomial running time bound. It suffices to show we can find in polynomial time a set of primes p_0, Q_1, \dots, Q_d and an exponent T satisfying the side conditions. If so, then the lengths of the binary expansions of the Q_j^T are bounded by a polynomial in the input length. The other operations of the reduction procedure (Chinese remainder theorem, solving linear congruences, etc.) can all be carried out in time polynomial in the input length.

Let Σ denote the input length, so in particular $\Sigma \geq d$. The product $\prod_{i=1}^d a_i$ has length at most Σ^2 , hence has at most Σ^2 distinct prime factors. Hence p_0 is one of the first $\Sigma^2 + 1$ prime numbers. We find it by trial division in $O(\Sigma^4)$ bit operations.

Finding the Q_j is the main problem. The relations (2.11a), (2.11b) imply

$$p_0^R \leq 2^{\Sigma^2+1} A \leq 2^{\Sigma^2+\Sigma(\log d)+1} \leq 2^{2\Sigma^2},$$

consequently, we choose

$$T = 3\Sigma^2.$$

For this choice of T we can be sure that the size condition (iii) holds, even if $Q_1 = 3$.

We now use the following simple search subroutine to locate suitable primes Q_j . Test for each $x = 1, 2, 3 \dots$ whether the interval $I_x = [x, (2)^{1/T}x]$ contains at least $d + \Sigma^2 + 1$ distinct prime numbers. Halt when such an x is found. We are guaranteed that at least d of these primes satisfy the relative primality condition (ii) of the size conditions, since there are at most $\Sigma^2 + 1$ distinct prime divisors of $p_0 a_1 \dots a_d$. The size conditions (i), (iv) are automatically satisfied by such a set of d primes. In this subroutine we test each integer y in I_x for primality by trial division by all integers $\leq \sqrt{y}$, and for each prime y that we find we test whether size condition (ii) holds by division. The number of bit operations involved in testing each interval I_x is $O((x^{3/2} + x\Sigma^2)(\log x)^2)$.

It suffices to obtain an upper bound which is polynomial in Σ for the value of x at which the search algorithm halts. We use the following result of Heath-Brown and Iwaniec [7].

THEOREM. *For each $\delta > \frac{11}{20}$ there is a constant $x_0(\delta)$ such that for all $x > x_0(\delta)$ the interval $[x, x + x^\delta]$ contains a prime.*

We choose $\delta = \frac{3}{5}$ and $x = \Sigma^{15}$. Now

$$(2)^{1/T} \geq 2^{1/3\Sigma^2} \geq 1 + \frac{1}{6\Sigma^2}.$$

Hence

$$I_x \subseteq \left[x, \left(1 + \frac{1}{6\Sigma^2} \right) x \right] = \left[\Sigma^{15}, \Sigma^{15} + \frac{1}{6} \Sigma^{13} \right].$$

However, each interval $[\Sigma^{15}, \Sigma^{15} + \Sigma^9]$ contains a prime. We have $\gg \Sigma^4$ such subintervals in (2.24) and so we produce $\gg \Sigma^4$ primes in the interval (2.24), which is enough. Thus, we can locate the required Q_i by a trial-divide process in $O(\Sigma^{23})$ bit operations. \square

Remarks. (1) P. van Emde Boas has observed that this proof can be modified to avoid the use of the deep result of Heath-Brown and Iwaniec [7]. His key idea is that

the reduction of GSA_1^* to WEAK PARTITION remains valid if the prime powers Q_i^T used in the reduction are replaced by pairwise relatively prime integers R_i satisfying conditions like the size conditions (with $T = 1$). To find such R_i , he notes that all the numbers of the form $U + p_k$ where $U = \prod_{j=1}^v p_j$ and with $p_v < p_k < 3p_v$ are pairwise relatively prime. Using a form of the prime number theorem with an explicit error term, one can prove that for $v \geq c_0 \Sigma^2$ for a suitable constant c_0 the existence of a subset of these integers with the properties required of the R_j is guaranteed. (Condition (iii) of the size conditions must be replaced by $U > p_v^R(v+1)2(d+1)$.)

(2) In any case the full strength of the Heath-Brown and Iwaniec result [7] is not needed in the proof above. A result like theirs with any $\delta < 1$ suffices, e.g. the original result of Hoheisel [8].

Proof of Theorem D. We study the complexity of the set:

$$LBSA^* = \{(\alpha, Q, N) : Q \text{ is the largest BSAD to } \alpha \text{ with } Q \leq N\}.$$

$LBSA^*$ is in co-NP, because its complement

$$\overline{LBSA^*} = \{(\alpha, Q, N) : Q \text{ is not the largest BSAD to } \alpha \text{ with } Q \leq N\}$$

is in NP. (More precisely, the complement of $LBSA^*$ is the union of $\overline{LBSA^*}$ and a P-time recognizable set of ill-formed words.) To see this, we use the nondeterministic Turing machine that “guesses” the largest BSAD $Q' \leq N$, and then checks that $Q' \neq Q$ and $\{\{Q'\alpha\}\} < \{\{Q\alpha\}\}$.

To show that $NP \neq co-NP$ implies $LBSA^*$ is not in NP, we will use the notion of nondeterministic conjunctive truth table reducibility, which we denote by \leq_{NP}^c (see [13], p. 37; [9], [5], p. 161). We say $A \leq_{NP}^c B$ if there is *nondeterministic oracle Turing machine* (NOTM) having B as an oracle, which accepts the set A , “YES” answers to the oracle are permitted, the NOTM never halting if a “NO” answer occurs. (An oracle Turing machine has a special oracle tape onto which an element x can be written, and a special oracle state which will provide a “YES” or “NO” answer to the query “Is $x \in B$?”.) We use results of Leggett ([13], Theorem 3.1, Corollary 2.29.1, cf. [5, p. 165]).

THEOREM (Leggett). *NP is closed downwardly under \leq_{NP}^c , i.e. if $A \leq_{NP}^c B$ and B is in NP, then A is in NP. In particular, if $A \leq_{NP}^c B$ and A is co-NP complete, then $NP \neq co-NP$ implies that B is not in NP.*

We use the set:

$$\overline{GSA^*} = \left\{ (\alpha, N, s_1, s_2) : \text{There exists no } Q \text{ with } 1 \leq Q \leq N \text{ and } \{\{Q\alpha\}\} \leq \frac{s_1}{s_2} \right\}.$$

Theorem C implies that $\overline{GSA^*}$ is the complement of an NP-complete set, hence is co-NP complete.

Theorem D then follows from the following fact.

Fact. $\overline{GSA^*} \leq_{NP}^c \overline{LBSA^*}$.

To prove the fact, we consider a nondeterministic oracle Turing machine (NOTM) having $LBSA^*$ as an oracle set. Now given (α, N, s_1, s_2) the NOTM “guesses” the Q which is the largest BSAD for α with $Q \leq N$, calls the oracle, gets a “yes” answer, and then checks if $\{\{Q\alpha\}\} > s_1/s_2$. If so, the NOTM halts and accepts (α, N, s_1, s_2) . Thus, this NOTM recognizes $\overline{GSA^*}$, proving the Fact. \square

Proof of Theorem E. We consider the set:

$$BSAI^* = \{(\alpha, N_1, N_2) : \text{There is a BSAD } Q \text{ to } \alpha \text{ with } N_1 \leq \alpha \leq N_2\}.$$

Theorem C showed GSA^* is NP-complete. To show BSAI^* is NP-hard, we exhibit a polynomial time Turing reduction of GSA^* to BSAI^* . To do this we use a deterministic oracle Turing machine (DOTM) which provides both “YES” and “NO” answers to queries of the form “Is $x \in \text{BSAI}^*$?” Given input (α, N, s_1, s_2) this DOTM locates the largest BSAD Q with $Q \leq N$ by an interval bisection strategy starting with the interval $[1, N]$, using at most $O(\log N)$ queries of the BSAI^* oracle. It then checks if $\{\{Q\alpha\}\} \leq s_1/s_2$. If so, it accepts the input and halts, otherwise rejects the input and halts.

To show that BSAI^* is in Δ_2^P , it suffices to show that BSAI^* can be recognized by a DOTM having a set in NP as an oracle [5, pp. 161–163]. We show a DOTM with oracle set GSA^* will do. Given (α, N_1, N_2) where $\alpha = (a_1/b_1, \dots, a_n/b_n)$, set $D = b_1 \cdots b_n$. Then every $\{\{Q\alpha\}\} = m/D$ for some m , $0 \leq m \leq D$. Our DOTM locates the minimal value m_0/D of $\{\{Q\alpha\}\}$ for $1 \leq Q \leq N_2$ by a sequence of at most $O(\log D)$ GSA^* oracle calls using $s_1 = m$, $s_2 = D$ and a bisection strategy on m . Then it locates the smallest Q with $\{\{Q\alpha\}\} = m_0/D$ by a bisection strategy on the interval $[1, N_2]$, using at most $O(\log N_2)$ oracle calls. If $N_1 \leq Q \leq N_2$, it accepts (α, N_1, N_2) and halts, otherwise it rejects (α, N_1, N_2) and halts. \square

Proof of Theorem F. Any GSA problem can be polynomial time encoded as a 2-IP problem, using the encoding (2.1) of Theorem A. \square

Acknowledgment. I am indebted to P. van Emde Boas for suggesting improvements to a draft of this paper, and to L. Lovasz for informing me of the details of the L^3 algorithm and pointing out its relation to simultaneous approximation.

REFERENCES

- [1] L. M. ADLEMAN AND R. L. RIVEST, *How to break the Lu-Lee (COMSAT) public-key cryptosystem*, MIT Laboratory for Computer Science, Cambridge, MA, July 1979.
- [2] A. BRENTJES, *A two-dimensional continued fraction algorithm for best approximations with an application to cubic number fields*, J. Reine Angew. Math., 326 (1981), pp. 18–44.
- [3] P. VAN EMDE BOAS, *Another NP-complete partition problem and the complexity of computing short vectors in a lattice*, Math. Dept. Report 81–04, Univ. Amsterdam, April 1981.
- [4] H. R. P. FERGUSON AND R. N. FORCADE, *Generalization of the Euclidean algorithm for real numbers to all dimensions higher than two*, Bull. Amer. Math. Soc., NS. 1 (1979), pp. 912–914.
- [5] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.
- [6] J. M. GOETHALS AND C. COUVREUR, *A cryptanalytic attack on the Lu-Lee public key cryptosystem*, Philips J. Research, 35 (1980), pp. 301–306.
- [7] R. HEATH-BROWN AND H. IWANIEC, *On the difference between consecutive primes*, Inventiones Math., 55 (1979), pp. 49–69.
- [8] G. HOHEISEL, *Primzahlprobleme in der Analysis*, Sitz. Preuss. Akad. Wiss., 33 (1930), pp. 3–11.
- [9] R. E. LADNER, N. A. LYNCH AND A. L. SELMAN, *A comparison of polynomial time reducibilities*, Theoret. Comput. Sci., 1 (1975), pp. 103–123.
- [10] J. C. LAGARIAS, *Some new results in simultaneous Diophantine approximation*, in Proc. the Queen’s Number Theory Conference 1979, P. Ribenboim, ed., Queen’s Papers in Pure and Applied Mathematics No. 54, 1980, pp. 453–474.
- [11] ———, *Best simultaneous diophantine approximations I. Growth rates of best approximation denominators*, Trans. Amer. Math. Soc., 272 (1982), pp. 545–554.
- [12] ———, *Knapsack-type public key cryptosystems and Diophantine approximation*, in Advances in Cryptology, Proc. of Crypto-83, D. Chaum, ed, Plenum, New York, 1984, pp. 3–24.
- [13] E. W. LEGGETT, JR., *Tools and techniques for classifying NP-complete problems*, Ph.D. Thesis, Dept. Computer and Information Sciences, Ohio State Univ., Columbus, OH, 1977.
- [14] A. K. LENSTRA, H. W. LENSTRA, JR. AND L. LOVASZ, *Factoring polynomials with rational coefficients*, Math. Annal., 261 (1982), pp. 515–534.
- [15] H. W. LENSTRA, JR., *Integer programming with a fixed number of variables*, Math. Oper. Res., to appear.
- [16] K. L. MANDERS AND L. ADLEMAN, *NP-complete decision problems for binary quadratics*, J. Comput. Systems Sci., 16 (1978), pp. 168–184.

- [17] A. PAZ, *Generating best approximations of vectors of real numbers with a preassigned tolerance*, preprint.
- [18] A. RUBIN, *A note on sum-distinct sets and a problem of Erdős*, Abstracts of the A.M.S. 2 (1981), p. 42.
- [19] A. SHAMIR, *On the cryptocomplexity of knapsack systems*, Proc. 11th ACM Symposium on the Theory of Computing, ACM, 1979, pp. 118–129.
- [20] —, *A polynomial time algorithm for breaking the basic Merkle-Hellman cryptosystem*, Proc. 23rd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1982, pp. 145–152.
- [21] A. SHAMIR AND R. E. ZIPPEL, *On the security of the Merkle-Hellman cryptographic scheme*, IEEE Trans. Information Theory, IT-26 (1980), pp. 339–340.
- [22] H. ZASSENHAUS, *A new polynomial factorization algorithm*, unpublished manuscript, 1981.

ARBORICITY AND SUBGRAPH LISTING ALGORITHMS*

NORISHIGE CHIBA† AND TAKAO NISHIZEKI†

Abstract. In this paper we introduce a new simple strategy into edge-searching of a graph, which is useful to the various subgraph listing problems. Applying the strategy, we obtain the following four algorithms. The first one lists all the triangles in a graph G in $O(a(G)m)$ time, where m is the number of edges of G and $a(G)$ the arboricity of G . The second finds all the quadrangles in $O(a(G)m)$ time. Since $a(G)$ is at most three for a planar graph G , both run in linear time for a planar graph. The third lists all the complete subgraphs K_l of order l in $O(la(G)^{l-2}m)$ time. The fourth lists all the cliques in $O(a(G)m)$ time per clique. All the algorithms require linear space. We also establish an upper bound on $a(G)$ for a graph G : $a(G) \leq \lceil (2m+n)^{1/2}/2 \rceil$, where n is the number of vertices in G .

Key words. arboricity, clique, complete subgraph, independent set, quadrangle, subgraph listing algorithm, triangle

1. Introduction. The problems to list certain kinds of subgraphs of a graph arise in many practical applications [2], [3], [4], [6], [8], [10], [11]. In this paper we introduce a new simple strategy into edge-searching of a graph, which is useful to the various subgraph listing problems. We choose a vertex v in a graph and scan the edges of the subgraph induced by the neighbors of v to find the pattern subgraphs containing v . The feature of the strategy is to repeat the searching above for each vertex v in nonincreasing order of degree and to delete v after v is processed so that no duplication occurs. We will show in the succeeding section that the procedure above requires $O(a(G)m)$ time. Throughout this paper m is the number of edges of a graph G , n is the number of vertices of G , and $a(G)$ is the *arboricity* of G , that is, the minimum number of edge-disjoint spanning forests into which G can be decomposed [5]. We use the rather unfamiliar graph invariant $a(G)$ as a parameter in bounding the running time of algorithms.

The strategy yields simple algorithms for the problems to list certain kinds of subgraphs of a graph. The kinds of these subgraphs include "triangle," "quadrangle," "complete subgraph of a fixed order," and "clique." Our algorithms are as fast as the known ones if any, and a factor n is often reduced to $a(G)$ in the time complexity.

In § 2 we give an upper bound on $a(G)$ for a general graph G : $a(G) \leq \lceil (2m+n)^{1/2}/2 \rceil$, which implies $a(G) \leq O(m^{1/2})$ for a connected graph G . In § 3 we give a simple algorithm which lists all the triangles in an arbitrary graph G in $O(a(G)m)$ time. In § 4 we present an $O(a(G)m)$ time algorithm for finding all the quadrangles (i.e. C_4) in G , which does not actually list C_4 but finds a representation of all the C_4 . If G is planar, $a(G) \leq 3$, so these two algorithms run in linear time for planar graphs. Because of the bound on $a(G)$, they run in at most $O(m^{3/2})$ time for general graphs. In § 5, extending the triangle listing algorithm, we present an $O(la(G)^{l-2}m)$ time algorithm for listing all the complete subgraphs of order l (i.e. K_l) in G , where l is an arbitrary number. Finally in § 6 we present an algorithm for listing all the cliques in G in $O(a(G)m)$ time per clique. All our algorithms require linear space and exceed the known algorithms [3], [6], [9] for the same purposes in running time, space, or simplicity.

* Received by the editors December 15, 1982.

† Department of Electrical Communications, Faculty of Engineering, Tohoku University, Sendai 980, Japan.

2. Preliminaries. We first define some terms. Let $G = (V, E)$ be a *simple graph* with vertex set V and edge set E . The edge set of graph G is often denoted by $E(G)$. The edge joining vertices u and v is denoted by (u, v) . Throughout this paper we denote by n the number of vertices and by m the number of edges of a graph. Let $d(v)$ denote the *degree* of a vertex v , that is, the number of edges incident to v . A graph is *planar* if it is embeddable on the plane without edge crossing. It is well-known that $m \leq 3n - 3$ if G is planar [5]. A *triangle* in a graph is a *cycle* of length three (i.e. C_3), in other words, a *complete subgraph* of three vertices (i.e. K_3). An *independent set* is a set of pairwise nonadjacent vertices in a graph. A *clique* is a maximal complete subgraph in a graph. We denote by $\lceil x \rceil$ the smallest integer not less than x .

We next present two results; the first is concerned with the arboricity of a graph and the other with the time required by scanning edges in a way of our strategy.

LEMMA 1. *Let a graph G have n vertices and m edges. Then*

- (1) (a) $a(G) \leq \lceil (2m + n)^{1/2}/2 \rceil$;
 (b) $a(G) \leq \lceil n/2 \rceil$; and
 (c) $a(G) \leq 3$ if G is planar [5, p. 124].

Proof. (a) Nash-Williams [7] showed that

$$(2) \quad a(G) = \max_{H \subset G} \lceil q/(p-1) \rceil,$$

where H runs over all nontrivial subgraphs of G , p is the number of vertices and q the number of edges of H . Suppose that the maximum in the right-hand side of (2) is achieved by a subgraph H having p vertices and q edges. Let k be the number of edges of a complete graph with p vertices, that is, $k = p(p-1)/2$. Consider the following two cases.

Case 1. $k \leq m$.

$$\begin{aligned} a(G) &= \lceil q/(p-1) \rceil \leq \lceil k/(p-1) \rceil = \lceil p/2 \rceil \\ &= \lceil (2k+p)^{1/2}/2 \rceil \leq \lceil (2m+n)^{1/2}/2 \rceil. \end{aligned}$$

Case 2. $k \geq m$.

$$\begin{aligned} a(G) &= \lceil q/(p-1) \rceil \leq \lceil m/(p-1) \rceil \leq \lceil \{mk/(p-1)^2\}^{1/2} \rceil \\ &= \lceil \{(m(p-1)+m)/2(p-1)\}^{1/2} \rceil \\ &\leq \lceil \{m/2+k/2(p-1)\}^{1/2} \rceil \\ &= \lceil (2m+p)^{1/2}/2 \rceil \\ &\leq \lceil (2m+n)^{1/2}/2 \rceil. \end{aligned}$$

(b) Immediate from (2).

(c) If G is planar, (2) implies that

$$a(G) \leq \max_{H \subset G} \lceil (3p-3)/(p-1) \rceil = 3. \quad \text{Q.E.D.}$$

Since $a(K_n) = \lceil n/2 \rceil = \lceil (2m+n)^{1/2}/2 \rceil$ where $m = n(n-1)/2$, there exist an infinite number of graphs attaining the upper bound in (1). In this sense the bound is best possible. It should be noted that $a(G) = O(1)$ for a large class of graphs including (i) planar graphs, (ii) graphs of bounded genus, and (iii) graphs of bounded maximum degree.

LEMMA 2. *If graph $G = (V, E)$ has n vertices and m edges, then*

$$\sum_{(u,v) \in E} \min \{d(u), d(v)\} \leq 2a(G)m.$$

Proof. Let F_i ($1 \leq i \leq a(G)$) be the edge-disjoint spanning forests of G such that $E(G) = \bigcup_{1 \leq i \leq a(G)} E(F_i)$. Associate each edge of F_i with a vertex of G as follows: choose an arbitrary vertex u of each tree T in forest F_i as the root of T ; regard T as a rooted tree with root u in which all the edges are directed from the root to the descendants; and associate each edge e of tree T with the head vertex $h(e)$ of e . Thus, every vertex of F_i , except the roots, is associated with exactly one edge of F_i . Then we have

$$\begin{aligned} \sum_{(u,v) \in E} \min \{d(u), d(v)\} &\leq \sum_{1 \leq i \leq a(G)} \sum_{e \in E(F_i)} d(h(e)) \\ &\leq \sum_{1 \leq i \leq a(G)} \sum_{v \in V} d(v) \\ &= 2a(G)m. \end{aligned} \qquad \text{Q.E.D.}$$

3. Algorithm for listing triangles. The triangle detection problem often arises in many combinatorial problems such as (1) the minimum cycle detection problem [6], (2) the approximate Hamiltonian walk problem in maximal planar graphs [8], and (3) the approximate minimum vertex cover (or maximum independent set) problem in planar graphs [3], [4]. Itai and Rodeh [6] presented an algorithm for finding all the triangles, which uses an adjacency matrix, so requires $O(n^2)$ space but runs in $O(m^{3/2})$ time for general graphs and in $O(n)$ time for planar graphs. Bar-Yehuda and Even [3] improved the space complexity of the algorithm from $O(n^2)$ into $O(n)$ by avoiding the use of the adjacency matrix. On the other hand Papadimitriou and Yannakakis [9] gave a linear, but a little complicated, algorithm for finding all the complete subgraphs, i.e. K_i ($1 \leq i \leq 4$), in a planar graph with assuming a plane embedding of the graph.

Our algorithm for listing triangles in a graph G is very simple as shown below. Observe that each triangle containing a vertex v corresponds to an edge joining two neighbors of v .

procedure K3(G);

{Let G be a graph with n vertices and m edges.}

begin

 sort the vertices v_1, v_2, \dots, v_n of G in such a way that $d(v_1) \geq d(v_2) \geq \dots \geq d(v_n)$;

for $i := 1$ **to** $n - 2$

do begin

 {find all the triangles containing vertex v_i , each of which corresponds to an edge joining two neighbors of v_i }

1: mark all the vertices adjacent to v_i ;

for each marked vertex u

do begin

2: **for** each vertex w adjacent to u

do if w is marked

then print out triangle (v_i, u, w) ;

3: erase the mark from u

end;

 {delete v_i from G so that no duplication occurs.}

- 4: delete vertex v_i from G and let G be the resulting graph
 end
 end;

We have the following result on the algorithm.

THEOREM 1. *Let G be a connected graph with n vertices and m edges. Algorithm K3 lists all the triangles in G in $O(a(G)m)$ time, and especially in $O(n)$ time if G is planar.*

Proof. Since one can easily verify the correctness, we shall show that the algorithm runs in $O(a(G)m)$ time.

Clearly the degrees of vertices can be computed in $O(m)$ time. Since the degree of any vertex is at most $n-1$, one can sort the vertices in $O(n)$ time by the bucket sort [1]. We use doubly linked adjacency lists as a data structure to represent a graph G . The two copies of each edge (u, v) , one in the list of v and the other in the list of u , are also doubly linked. Using such a data structure, we can delete a vertex v from G in $O(d(v))$ time, and scan all the vertices adjacent to a vertex v in $O(d(v))$ time. Now consider the time required by the i th iteration of the outmost **for** statement. Statements 1, 3 and 4 require $O(d(v_i))$ time. Statement 2 requires at most $O(\sum_{u \in N(v_i)} d(u))$ time, where $d(u)$ denotes the degree of vertex u in the original graph and $N(v_i)$ denotes the set of neighbors of v_i in the current graph. Therefore the total running time T of the algorithm is bounded as follows:

$$T \leq O(m) + O(n) + \sum_{v_i \in V} O(d(v_i)) + \sum_{u \in N(v_i)} d(u).$$

Since v_i has the largest $d(v_i)$ among all the vertices in the current graph, we have $d(u) \leq d(v_i)$ for each $u \in N(v_i)$. Since v_i is deleted at Statement 4, each edge of G is involved exactly once in the double summations above. Thus we have

$$T \leq O(m) + O(n) + O\left(\sum_{(u,v) \in E} \min\{d(u), d(v)\}\right).$$

Using Lemma 2, we have $T \leq O(a(G)m)$.

If G is planar, the algorithm runs in $O(a(G)m) \leq O(n)$ time since $a(G) \leq 3$ by Lemma 1(c). Q.E.D.

Algorithm K3 is conceptually very simple and easy to implement. Furthermore it is at least as fast as the known ones [3], [6], [9] since $O(a(G)m) \leq O(m^{3/2})$ by Lemma 1(a).

The benefit of our strategy may be intuitively explained as follows: since we delete the vertices one by one in the largest degree order, the graph tends to become sparse soon; this also prevents the edges incident to a vertex of large degree from being scanned many often.

Applying the strategy, we will give three more algorithms for other subgraphs listing problems in the succeeding sections.

4. Algorithm for finding quadrangles. In this section, using our searching strategy, we design an efficient algorithm for finding all the quadrangles.

If vertices u_1, u_2, \dots, u_l ($l \geq 2$) are all adjacent to two common vertices v and w , that is, these $l+2$ vertices induce a complete bipartite graph $K_{2,l}$, then any quadruple (v, u_i, w, u_j) , $1 \leq i < j \leq l$, forms a quadrangle. Thus even in a planar graph, there may exist $O(n^2)$ quadrangles. Instead of listing these quadrangles individually, we list a triple $(v, w, \{u_1, u_2, \dots, u_l\})$ representing them altogether.

Our algorithm C4 depicted below proceeds, for each vertex v of a graph, to find all the quadrangles containing v : for each vertex w within distance two from v , the

algorithm finds all such u_1, u_2, \dots, u_l which are adjacent to both v and w , and stores them in a set $U[w]$. When the quadrangles containing v have been found, v is deleted in order to avoid the duplication.

```

procedure C4( $G$ );
  {Let  $G = (V, E)$  be a graph with  $n$  vertices.}
  begin
    sort the vertices in  $V$  in a way that  $d(v_1) \geq d(v_2) \geq \dots \geq d(v_n)$ ;
    for each vertex  $v \in V$  do  $U[v] := \emptyset$ ;
    for  $i := 1$  to  $n$ 
      do begin
        for each vertex  $u$  adjacent to  $v_i$ 
          do for each vertex  $w \neq v_i$  adjacent to  $u$ 
            do begin
               $U[w] := U[w] \cup \{u\}$ 
            end;
        for each vertex  $w$  with  $|U[w]| \geq 2$ 
          do print out the triple  $(v_i, w, U[w])$ ;
        for each vertex  $w$  with  $U[w] \neq \emptyset$  do  $U[w] := \emptyset$ ;
        delete the vertex  $v_i$  from  $G$  and let  $G$  be the new graph
      end
    end;

```

The graph depicted in Fig. 1 contains seven quadrangles. Algorithm C4 lists the following five triples: $(1, 5, \{2, 7, 10\})$, $(1, 4, \{2, 3\})$, $(3, 8, \{4, 6\})$, $(3, 9, \{4, 6\})$, and $(4, 6, \{8, 9\})$. The first triple represents three quadrangles.

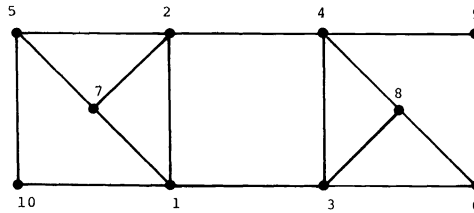


FIG. 1. A graph containing seven quadrangles.

We easily obtain the following theorem.

THEOREM 2. *Algorithm C4 obtains a representation of all the quadrangles in a connected graph G in $O(a(G)m)$ time, using $O(m)$ space.*

Note that Algorithm C4 does not store the triples. Since Algorithm C4 runs in $O(a(G)m)$ time, clearly all the quadrangles, if desired, could be represented by the triples in $O(a(G)m)$ space.

5. Algorithm for listing complete subgraphs. Observe the following fact: Algorithm K3 finds a triangle (K_3) containing a vertex v by detecting an edge (K_2) in a subgraph induced by the neighbors of v . In a similar manner, one can find a complete subgraph K_l containing a vertex v by detecting a complete subgraph K_{l-1} in a subgraph induced by the neighbors of v . We first present, for the sake of understanding, a simple recursive algorithm for listing the complete subgraphs K_l of fixed order $l (\geq 2)$ in a graph $G = (V, E)$.

```

procedure COMPLETE( $l, G$ )
  procedure K( $k, G_k$ );
    {find all  $K_k$  in a subgraph  $G_k$ .  $C$  is a global stack.}
  begin
    if  $k = 2$ 
      then for each edge  $(x, y)$  of  $G_k$ 
        do print out  $\{x, y\} \cup C$ 
      else for each vertex of  $G_k$ 
        do begin
          let  $G_{k-1}$  be the subgraph of  $G_k$ 
            induced by the neighbors of  $v$ ;
          add  $v$  to the top of  $C$ ;
          K( $k-1, G_{k-1}$ );    {find  $K_{k-1}$  in  $G_{k-1}$ , which, together
            with  $v$ , form  $K_k$  in  $G_k$ }
          delete  $v$  from the top of  $C$ ;
           $G_k := G_k - v$     {delete  $v$  to avoid the duplication}
        end
    end;
  begin
     $C := \emptyset$ ;
    K( $l, G$ )
  end;

```

In the algorithm above Stack C contains a sequence of vertices which have been known to be pairwise adjacent. When procedure $K(k, G_k)$ is executed (at a recursive call of depth $l-k$), C contains $l-k$ pairwise adjacent vertices, and the subgraph G_k contains all the vertices that are adjacent to every vertex in C . Procedure $K(k, G_k)$ finds all the K_k in G_k , each of which, together with the vertices in C , forms a K_l in G . Noting these facts, one can easily verify the correctness of the algorithm by induction on l . However the direct implementation of COMPLETE does not yield an efficient algorithm because it had to produce and store a sequence of induced subgraphs of G .

In order to avoid the trouble above, we introduce a certain kind of vertex-labeling, by which all the vertices are labeled either " l ", " $l-1$ ", \dots , or " k ". The vertices labelled " k " induce the subgraph G_k currently processed. Let U be the vertex set of G_k . We order the entries of the adjacency lists as follows: in the adjacency list of each vertex $v \in U$, the neighbors of v having labels not exceeding the label of v occupy the first part of the list and the other neighbors appear in the latter part in nondecreasing order of the labels. Thus all the neighbors of each vertex $u \in U$ appear in the adjacency list of u in nondecreasing order of the labels, so that the first parts of the adjacency lists represent G_k . We also employ the same strategy as the triangle listing algorithm, that is, process the vertices of G_k in the nonincreasing order of degrees in G_k . Thus the procedure is refined as follows.

```

procedure COMPLETE( $l, G$ );
  procedure K( $k, U$ );
    { $U$  is the vertex set of  $G_k$ .  $d_k(v)$  is the degree of vertex  $v$  in  $G_k$ .}
  begin
    if  $k = 2$ 
      then

```

```

1:  for each edge  $(x, y)$  of the subgraph induced by  $U$ 
    do print out  $\{x, y\} \cup C$ 
    else
    begin
2:  sort the vertices in  $U$  in way that  $d_k(v_1) \geq d_k(v_2) \geq \dots \geq d_k(v_{|U|})$ , and store
    them in list;
    for  $i := 1$  to  $|U|$ 
    do begin
        let  $U'$  be the set of all the vertices which are adjacent to  $v_i$  and labeled
        " $k$ ";  $\{U'$  is the vertex set of  $G_{k-1}\}$ 
3:  relabel all the vertices in  $U'$  " $k-1$ ";
4:  in the adjacency list of each vertex  $u \in U'$ , move the neighbors of  $u$  in
         $U'$  at the first part;  $\{\text{the vertices of } G_{k-1} \text{ occupy the first parts of the}$ 
        adjacency lists of vertices in  $U'$ , which realize the adjacency lists of
         $G_{k-1}\}$ 
5:  determine the degree  $d_{k-1}(u)$  of each  $u \in U'$  in  $G_{k-1}$ ;
6:  add the vertex  $v_i$  to  $C$ ;
7:   $K(k-1, U')$ ;
8:  delete the top entry  $v_i$  from  $C$ ;
9:  relabel all the vertices in  $U'$  " $k$ ";  $\{\text{recovery to } G_k\}$ 
10: relabel  $v_i$  " $k+1$ ";  $\{\text{logical (not physical) deletion of } v_i \text{ from } G_k\}$ 
11: in the adjacency list of each vertex  $v \in U'$ , move the entry  $v_i$  to the
        position next to the last entry containing a vertex labeled " $k$ ";
    end
    end
end
end;
begin {of COMPLETE}
    label all the vertices of  $G$  " $l$ ";
    determine  $d_l(v) (= d(v))$  for each  $v \in V$ ;
     $C := \emptyset$ ;
     $K(l, V)$   $\{V$  is the vertex set of  $G = G_l\}$ 
end {of COMPLETE};

```

We have the following result on the algorithm.

THEOREM 3. *If a connected graph G has n vertices and m edges, then Algorithm COMPLETE lists all the complete subgraphs of order l (≥ 2) in G in $O(la(G)^{l-2}m)$ time using linear space.*

Proof. (a) *Correctness.* Note that throughout the execution of COMPLETE the entries of the adjacency lists are ordered as mentioned just before the refined algorithm. then one can easily verify the correctness of the refined one as well as the original one.

(b) *Space.* We use the same data structure as the algorithm K3 to represent a graph. One recursive call with respect to a vertex v produces a list which stores the vertices in U in nonincreasing order of degree in G_k . The length of the list is at most $d(v)$. Therefore the total length over all the lists with respect to the vertices in C is at most $\sum_{v \in C} d(v) \leq 2m$ during the execution of the algorithm. Thus the algorithm requires linear space.

(c) *Time.* We now establish the claim on the running time. If the subgraph G_k induced by U has m edges and n vertices, let $T(k, m, n)$ be the time required by procedure $K(k, U)$ to find all the K_k in G_k . Here $T(k, m, n)$ does not count the time

required by printing out K_l in Statement 1. First consider the case $k=2$, in which Statement 1 is executed. One can find all the edges of G_k in $O(m+n)$ time, because the edges of G_k occupy the first parts of the adjacency lists. Thus Statement 1 requires at most $O(m+n)$ time, and so $T(2, m, n) = O(m+n)$. Next consider the case $k \geq 3$. Clearly Statement 2 can be executed in $O(n)$ time. Consider the time required by the i th iteration of the **for** statement. Statements 3 and 9 require $O(d_k(v_i)+1)$ time, and Statements 6, 8, and 10 require $O(1)$ time. Just before Statement 3 is executed, in the adjacency list of each $u \in U$, the neighbors of u appear in nondecreasing order of the labels, which are “ k ”, “ $k+1$ ”, \dots , “ l ”. Therefore Statement 4 is performed as follows: in the adjacency list of each $u \in U'$, choose the vertices in U' (labeled “ $k-1$ ”) among the first $d_k(u)$ entries; and move them to the first part of the list. Thus Statement 4 requires $O(\sum_{u \in U'} (d_k(u)+1))$ time. Similarly one can show that Statements 5 and 11 require $O(\sum_{u \in U'} (d_k(u)+1))$ time. Statement 7 requires $T(k-1, (\sum_{u \in U'} d_{k-1}(u))/2, d_k(v_i))$ time by the definition of T . Note that the graph G_{k-1} induced by U' has at most $(\sum_{u \in U'} d_{k-1}(u))/2$ edges and $d_k(v_i)$ vertices. Thus, the i th iteration of the **for** statement requires

$$O(d_k(v_i)) + O\left(\sum_{u \in U'} d_k(u)\right) + O(1) + T\left(k-1, \left(\sum_{u \in U'} d_{k-1}(u)\right) / 2, d_k(v_i)\right)$$

time. Each $v_i \in U$ satisfies $d_k(v_i) \geq d_k(u)$ for every $u \in U'$. Therefore Lemma 2 implies that

$$\sum_{v_i \in U} \left\{ O(d_k(v_i)) + O\left(\sum_{u \in U'} d_k(u)\right) + O(1) \right\} = O(a(G_k)m + n).$$

Thus we have the recurrence

$$T(2, m, n) = O(m + n),$$

$$T(k, m, n) \leq O(a(G_k)m + n) + \sum_{v_i \in U} T\left(k-1, \left(\sum_{u \in U'} d_{k-1}(u)\right) / 2, d_k(v_i)\right).$$

Solving the recurrence with noting $a(G_{k-1}) \leq a(G_k)$, we have $T(k, m, n) = O(a(G_k)^{k-2}m + n)$.

Since procedure COMPLETE(l, G) calls $K(k, U)$ with $k=l$ and $U=V$ for a connected graph $G=(V, E)$, it requires $O(a(G)^{l-2}m)$ time in total to find all the K_l in G . This fact implies that the number of K_l in G is at most $O(a(G)^{l-2}m)$. Since one can print out a K_l in $O(l)$ time, the total running time of COMPLETE including the time for printing is at most $O(la(G)^{l-2}m)$. Q.E.D.

Theorem 3 together with Lemma 1(c) imply that Algorithm COMPLETE lists all the K_4 in a planar graph in *linear* time. The time complexity is the same as the algorithm of Papadimitriou and Yannakakis [9], but our algorithm does not need the plane embedding of a graph.

6. Clique listing algorithm. Tsukiyama et al. [11] presented an algorithm MIS which lists all the maximal independent sets in a graph G and requires $O(mn)$ time per maximal independent set. In this section, we first show that our strategy can reduce the running time to $O(a(G)m)$. Then, employing their idea and our strategy, we present an algorithm which lists all the cliques in a graph G in $O(a(G)m)$ time per clique.

The algorithm of Tsukiyama et al. is outlined as follows. Let $G=(V, E)$ be a given graph with vertex set $V=\{1, 2, \dots, n\}$. Each vertex is referred by the number. Let $G_i, 1 \leq i \leq n$, be the subgraph of G induced by vertices $1, 2, \dots, i$. $N(i)$ denotes

the set of vertices adjacent to i in the given graph G . Assume that I_{i-1} is a maximal independent set of G_{i-1} , then one can decide by the following rules whether I_{i-1} or $(I_{i-1} - N(i)) \cup \{i\}$ is a maximal independent set in G_i :

- (1) If $I_{i-1} \cap N(i) \neq \emptyset$, then I_{i-1} is a maximal independent set of G_i .
- (2) If there is no independent set I of G_{i-1} such that $I - N(i) \supseteq I_{i-1} - N(i)$, then $(I_{i-1} - N(i)) \cup \{i\}$ is a maximal independent set of G_i .

Thus they recursively generate all the maximal independent sets of G_i from the maximal independent sets of G_{i-1} . However duplications may occur in maximal independent sets produced by rule (2), so they avoided the duplications by choosing the lexicographically largest one among all the independent sets I_{i-1} having the same $I_{i-1} - N(i)$.

Tsukiyama et al. [11] implemented the backtracking algorithm MIS in a way that one recursive step on vertex i is performed in $O(\sum_{x \in N(i) - \{i+1, \dots, n\}} d(x)) = O(m)$ time, so that MIS requires $O(mn)$ time to find one maximal independent set. An easy observation leads us to an algorithm which requires $O(a(G)m)$ time per maximal independent set. We simply number the vertices of a given graph G in such a way that $d(1) \leq d(2) \leq \dots \leq d(n)$, and apply the same recursive method. Then, applying Lemma 2, we can easily show that the new algorithm requires

$$O\left(\sum_{1 \leq i \leq n} \sum_{x \in N(i) - \{i+1, \dots, n\}} d(x)\right) \leq O\left(\min_{(u,v) \in E} \{d(u), d(v)\}\right) = O(a(G)m)$$

time per maximal independent set. Unlike the preceding three algorithms, we number the vertices in nondecreasing order of degree so that the newly added vertex i has the largest degree in G_i . If G is sparse, the time complexity $O(a(G)m)$ is considerably better than $O(mn)$.

The problem of listing all the cliques of a graph G is equivalent to that of listing all the maximal independent sets of the complement G^c of G . Therefore the algorithm suggested above can list all the cliques of a graph G in $O(a(G^c)m^c)$ time per clique, where $m^c = n(n-1)/2 - m$ is the number of edges of G^c . However, this algorithm is not necessarily efficient for sparse graphs. Using a recursive method similar to MIS, we next give an algorithm CLIQUE which lists all the cliques in $O(a(G)m)$ time per clique. Unlike the case of maximal independent sets, guaranteeing the time complexity of $O(a(G)m)$ is not straightforward in this case, but requires some nontrivial arguments especially on the "lexico. test".

The set of vertices in a clique C is also denoted by C . The following is the outline of our algorithm CLIQUE.

procedure CLIQUE

procedure UPDATE (i, C)

{generate a new clique of G_i from a clique C of G_{i-1} }

begin

if $i = n + 1$

then print out a new clique C { C is a clique of $G = G_n$ }

else

begin

if $C - N(i) \neq \emptyset$ **then** UPDATE ($i + 1, C$); { C is a clique of G_i }

if both "maximality test" and "lexico. test" succeed

then

begin

SAVE := $C - N(i)$; {save the vertices removed from current C }

$C := (C \cap N(i)) \cup \{i\}$; {new C is a clique of G_i }

```

UPDATE (i+1, C);
C := (C - {i}) ∪ SAVE {recovery to old C}
end
end
end;
begin
number the vertices of a given graph G in such a way that  $d(1) \leq d(2) \leq \dots \leq d(n)$ ;
C := {1}; {C is the unique clique of  $G_1$ .}
UPDATE (2, C)
end;

```

In the algorithm above, “maximality test” checks whether the candidate of a new clique $C' = (C \cap N(i)) \cup \{i\}$ is indeed a clique (i.e. maximal complete subgraph) of G_i . The “lexico. test” checks whether C is the lexicographically largest clique of G_{i-1} containing $C \cap N(i)$ ($= C_0$). This test avoids the duplications of cliques. Note that the same clique C' of G_i may be produced more than once from distinct cliques of G_{i-1} containing C_0 . One can easily verify the correctness of the algorithm CLIQUE by induction on n . In what follows, we refine the algorithm so that it runs in $O(a(G)m)$ time per clique.

We begin with the following lemma, which implies that if a clique of G_i is generated from a clique C of G_{i-1} in $O(\sum_{x \in C} d(x))$ time, then one clique of G can be found in $O(a(G)m)$ time.

LEMMA 3. *Let the vertices $1, 2, \dots, n$ of a graph G satisfy $d(1) \leq d(2) \leq \dots \leq d(n)$, and let $C_i, 1 \leq i \leq n$, be an arbitrary clique of G_i where $G = G_n$. Then*

$$\sum_{1 \leq i \leq n} \sum_{x \in C_i} d(x) \leq 4a(G)m.$$

Proof. Let $c = \max_{1 \leq i \leq n} |C_i|$, then Equation (2) implies that

$$(3) \quad c \leq 2a(K_c) \leq 2a(G).$$

Since $d(i) \geq d(x)$ for any $x \in C_i$,

$$\sum_{1 \leq i \leq n} \sum_{x \in C_i} d(x) \leq \sum_{1 \leq i \leq n} \sum_{x \in C_i} d(i) \leq \sum_{1 \leq i \leq n} d(i)c \leq 2mc.$$

Combining this with (3), we have

$$\sum_{1 \leq i \leq n} \sum_{x \in C_i} d(x) \leq 4a(G)m. \quad \text{Q.E.D.}$$

The following three lemmas are concerned with the tests.

LEMMA 4 [maximality test]. *Let C be a clique of G_{i-1} . Then, $(C \cap N(i)) \cup \{i\}$ is a clique of G_i if and only if G_i has no vertex $y \in N(i) - C$ such that $y < i$ and $N(y) \supset C \cap N(i)$.*

Proof. Immediate. Q.E.D.

Using Lemma 4, one can perform the “maximality test” once in $O(d(i) + \sum_{x \in C \cap N(i)} d(x))$ time as follows: first compute $T(y) = |N(y) \cap C \cap N(i)|$ for $y \in V$ (in that time); then check whether there exists $y \in N(i) - C$ such that $y < i$ and $T(y) = |C \cap N(i)|$. (We will describe the detail later in the refined algorithm CLIQUE.)

LEMMA 5. *Let C_0 be a complete subgraph of a graph G . A clique C ($\supset C_0$) of G is the lexicographically largest one among all the cliques containing C_0 if and only if there is no vertex $y \notin C$ such that $N(y) \supset C_0 \cup C^y$, where $C^y = \{k \in C \mid k > y\}$.*

Proof. Necessity. Assume that there exists a vertex $y \notin C$ such that $N(y) \supset C_0 \cup C^y$. Then clearly there exists a clique containing $\{y\} \cup C_0 \cup C^y$ which is lexicographically larger than C .

Sufficiency. Assume that there exists a clique $C' \supset C_0$ which is lexicographically larger than C . Let y be the largest vertex in $C' - C$. Then $C \cap C' \supset C^y$ since every vertex in $(C - C')$ is less than y . Thus we have $N(y) \supset C \cap C' \supset C_0 \cup C^y$. Q.E.D.

The direct application of Lemma 5 would require $O(m)$ time to perform the "lexico. test" once, so the algorithm would require $O(mn)$ time per clique. The following lemma yields a more efficient "lexico. test".

LEMMA 6 [lexico. test]. *Let C be a clique of G which includes a complete subgraph C_0 , where C_0 may be empty. Let $p = |C - C_0|$, let $j_1 < j_2 < \dots < j_p$ be the vertices in $C - C_0$, and let $j_0 = 0$. For each vertex $y \notin C$, let $S(y) = |N(y) \cap (C^y - C_0)|$, and let $j_k > y$ be the smallest vertex in $N(y) \cap (C^y - C_0)$ if $S(y) \geq 1$. Then C is the lexicographically largest clique containing C_0 if and only if every $y \notin C$ such that $N(y) \supset C_0$ satisfies*

- (a) if $S(y) \geq 1$ then either $S(y) + k - 1 < p$ or $j_{k-1} > y$;
- (b) if $S(y) = 0$ then $j_p > y$.

Proof. Necessity. Assume that there exists a vertex $y \notin C$ such that $N(y) \supset C_0$, violating either (a) or (b). If $S(y) = 0$ and $j_p < y$, then $C^y = \emptyset$ and there exists a clique which includes $\{y\} \cup C_0$ and is lexicographically larger than C . Thus we may assume that $S(y) \geq 1$, $S(y) + k - 1 = p$ and $j_{k-1} < y$. (Note that $S(y) + k - 1 \leq p$.) Then the inequality $j_{k-1} < y$ implies $C^y - C_0 = \{j_k, j_{k+1}, \dots, j_p\}$, so $|C^y - C_0| = p - k + 1$. Combining this with $S(y) + k - 1 = p$, we have $S(y) = |C^y - C_0|$. Therefore there exists a clique which includes $\{y\} \cup C^y \cup C_0$ and is lexicographically larger than C .

Sufficiency. Assume that there exists a clique $C' (\supset C_0)$ which is lexicographically larger than C . Let y be the largest vertex in $C' - C$. Then we have $N(y) \supset C^y \cup C_0$ as shown in the proof of Lemma 5. If $S(y) = 0$, then clearly $j_p < y$, violating (b). Thus we may assume that $S(y) \geq 1$. Then clearly $j_{k-1} < y$ and $S(y) = |C^y - C_0|$, so $S(y) + k - 1 = |C^y - C_0| + k - 1 = p$, violating (a). Q.E.D.

Using Lemma 6, one can perform the "lexico. test" once in $O(\sum_{x \in C} d(x))$ time. We first compute $|N(y) \cap (C - C_0)|$ for $y \in V - C$ and then alter them to $S(y) = |N(y) \cap (C^y - C_0)|$, as shown in the refined CLIQUE. Thus the computation of $S(y)$ requires $O(\sum_{x \in C - C_0} d(x))$ time. Let $G = G_{i-1}$ as in the algorithm, then the direct access of the vertices $y \notin C$ such that $N(y) \supset C_0 (= C \cap N(i))$ would require $O(i)$ time, which may be greater than $O(\sum_{x \in C} d(x))$. However, we can perform the access in $O(\sum_{x \in C} d(x))$ time as follows. If either $C_0 \neq \emptyset$ or $S(y) \geq 1$, then y is accessible from the adjacency lists of vertices in C . On the other hand, if $C_0 = \emptyset$ and $S(y) = 0$, then y is not accessible from these lists. However, if (i) $C_0 = \emptyset$, (ii) C is not the lexicographically largest clique containing C_0 in G_{i-1} , and (iii) every $y \notin C$ satisfies condition (a) of Lemma 6, then C does not contain the largest vertex $i - 1$ of G_{i-1} . (Consider the largest clique C' and the largest vertex y in $C' - C$.) Thus in this case we can perform the "lexico. test" simply by checking whether C contains vertex $i - 1$, as will be known in the algorithm.

We are now ready to present the refined algorithm CLIQUE.

```

procedure CLIQUE;
  procedure UPDATE ( $i, C$ );
    begin
      if  $i = n + 1$ 
        then print out a new clique  $C$ 
      else
        begin

```



```

1:   if  $C - N(i) \neq \emptyset$  then UPDATE ( $i+1$ ,  $C$ );
    {prepare for tests}
    {compute  $T[y] = |N(y) \cap C \cap N(i)|$  for  $y \in V - C - \{i\}$ }
2:   for each vertex  $x \in C \cap N(i)$ 
    do for each vertex  $y \in N(x) - C - \{i\}$ 
        do  $T[y] := T[y] + 1$ ;
    {compute  $S[y] = |N(y) \cap (C - N(i))|$  for  $y \in V - C$ }
3:   for each vertex  $x \in C - N(i)$ 
    do for each vertex  $y \in N(x) - C$ 
        do  $S[y] := S[y] + 1$ ;
     $FLAG := true$ ;
    {maximality test}
4:   if there exists a vertex  $y \in N(i) - C$  such that  $y < i$  and  $T[y] = |C \cap N(i)|$ 
    then  $FLAG := false$ ;  $\{(C \cap N(i)) \cup \{i\}$  is not a clique of  $G_i\}$ 
    {lexico. test}
     $\{C \cap N(i)$  corresponds to  $C_o$  in Lemma 6}
5:   sort all the vertices in  $C - N(i)$  in ascending order  $j_1 < j_2 < \dots < j_p$ , where
     $p = |C - N(i)|$ ;
    {case  $S(y) \geq 1$ . See Lemma 6.}
6:   for  $k := 1$  to  $p$ 
    do for each vertex  $y \in N(j_k) - C$  such that  $y < i$  and  $T[y] = |C \cap N(i)|$ 
        do if  $y \geq j_k$ 
            then  $S[y] := S[y] - 1$  {alter  $S[y]$  to  $S(y)$ }
            else
                if ( $j_k$  is the first vertex which satisfies  $y < j_k$ )
                    then  $\{S[y] = S(y)\}$ 
                    if ( $S[y] + k - 1 = p$ ) and ( $y \geq j_{k-1}$ )  $\{j_0 = 0\}$ 
                        then  $FLAG := false$ ;  $\{C$  is not lexico. largest}
            {case  $S(y) = 0$ }
7:   if  $C \cap N(i) \neq \emptyset$ 
    then for each vertex  $y \notin C \cup \{i\}$  such that  $y < i$ ,  $T[y] = |C \cap N(i)|$  and
     $S[y] = 0$ 
        {access  $y$  from the adjacency list of a vertex in  $C \cap N(i)$ }
        do if  $j_p < y$  then  $FLAG := false$        $\{C$  is not lexico. largest.}
        else if  $j_p < i - 1$  then  $FLAG := false$ ;     $\{C$  is not lexico. largest.}
    {reinitialize  $S$  and  $T$ }
8:   for each vertex  $x \in C \cap N(i)$ 
    do for each vertex  $y \in N(x) - C - \{i\}$ 
        do  $T[y] := 0$ ;
9:   for each vertex  $x \in C - N(i)$ 
    do for each vertex  $y \in N(x) - C$ 
        do  $S[y] := 0$ ;
    { $FLAG$  is true if and only if  $(C \cap N(i)) \cup \{i\}$  is a clique of  $G_i$  and  $C$  is the
    lexicographically largest clique of  $G_{i-1}$  containing  $C \cap N(i)$ .}
10:  if  $FLAG$ 
    then
        begin
             $SAVE := C - N(i)$ ;
             $C := (C \cap N(i)) \cup \{i\}$ ;
            UPDATE ( $i+1$ ,  $C$ );

```

```

      C := (C - {i}) ∪ SAVE
    end
  end
end;
begin {of CLIQUE}
  number the vertices of a given graph G in such a way that  $d(1) \leq d(2) \leq \dots \leq d(n)$ ;
  for  $i := 1$  to  $n$  {initialize S and T}
    do begin  $S[i] := 0$ ;  $T[i] := 0$  end;
    C := {1};
    UPDATE (2, C)
  end {of CLIQUE};

```

We have the following theorem.

THEOREM 4. *Algorithm CLIQUE lists all the cliques of a connected graph G in $O(a(G)m)$ time per clique, using $O(m)$ space.*

Proof. Using Lemmas 4 and 6, one can prove the correctness. Therefore we shall concentrate on the claim on time and space.

Let C_n be an arbitrary clique of $G = G_n$, and inductively define C_i , $n-1 \geq i \geq 1$, to be the clique of G_i from which C_{i+1} is generated by procedure CLIQUE.

Consider the time $T(i)$ required by UPDATE (i, C_{i-1}), excluding the time required by the recursive calls in Statements 1 and 10. Noting the remark mentioned just before the refined CLIQUE, one can easily show that all the Statements 1-10 except 5 can be executed in $O(d(i) + |C_{i-1}| + \sum_{x \in C_{i-1}} d(x))$ time. We now show that the sorting in Statement 5 also requires at most $O(\sum_{x \in C_{i-1}} d(x))$ time. One can sort p items in $O(p \log p)$ time where $p = |C_{i-1} - N(i)|$ [1]. Since the subgraph induced by $C_{i-1} - N(i)$ is a complete subgraph, $O(p \log p) \leq O(p(p-1)) \leq O(\sum_{x \in C_{i-1} - N(i)} d(x))$. Here the bucket sort should not be used, because it requires $O(j_p)$ time, which may be greater than $O(p \log p)$. Thus $T(i) \leq O(d(i) + |C_{i-1}| + \sum_{x \in C_{i-1}} d(x))$.

Hence the total time required to generate C_n is at most $\sum_{2 \leq i \leq n} T(i) \leq O(\sum_{2 \leq i \leq n} (d(i) + |C_{i-1}| + \sum_{x \in C_{i-1}} d(x)))$. Lemma 3 implies that the time is $O(a(G)m)$.

Every UPDATE (i, C), $i \leq n$, calls at least once UPDATE ($i+1, C$) in Statement 1 or 10. In fact, if the recursive call does not occur in Statement 1, then it necessarily occurs in Statement 10. Thus every call of UPDATE eventually generates at least one clique, and hence the time spent by any statement is counted in the time above at least once for some clique C_n of G_n . Thus we have shown that CLIQUE requires $O(a(G)m)$ time per clique.

Since set C is a global variable, C requires $O(n)$ space. Since the sets of vertices contained in the local variable $SAVE$ are pairwise disjoint, $SAVE$ requires $O(n)$ space in total. The arrays S, T and the adjacency lists require $O(m)$ space. Thus CLIQUE requires $O(m)$ space in total. Q.E.D.

7. Conclusion. In this paper we introduced a simple edge-searching strategy and presented the four efficient algorithms for the various subgraph listing problems. We used the arboricity $a(G)$, a rather unfamiliar graph invariant, as a parameter in bounding the running time of algorithms. Our algorithms are as fast as the previous ones if any, and a factor n is often reduced to $a(G)$ in the running time. The key idea is in Lemma 2, which implies that if a certain operation on a graph consumes $O(\min\{d(u), d(v)\})$ time for each edge (u, v) then the operation can be executed for all the edges in a graph G in $O(a(G)m)$ time. It is expected that this result will find a number of other applications in graph problems.

Finally we remark that in this paper only the concept of arboricity is used in the analysis of the running time of algorithms and that any of our algorithms requires neither to find $a(G)$ nor to decompose a graph into the minimum number of edge-disjoint forests.

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] J. G. AUGUSTON AND J. MINKER, *An analysis of some graph theoretical cluster techniques*, J. Assoc. Comput. Mach., (1970), pp. 571-588.
- [3] R. BAR-YEHUDA AND S. EVEN, *On approximating a vertex cover for planar graphs*, Proc. 14th Annual ACM Symposium on Theory of Computing, San Francisco, May 5-7, 1982, pp. 303-309.
- [4] N. CHIBA, T. NISHIZEKI AND N. SAITO, *An algorithm for finding a large independent set in planar graphs*, Networks, 13 (1983), pp. 247-252.
- [5] F. HARARY, *Graph Theory*, revised, Addison-Wesley, Reading, MA, 1972.
- [6] A. ITAI AND M. RODEH, *Finding a minimum circuit in a graph*, this Journal, 7, (1978), pp. 413-423.
- [7] C. ST. J. A. NASH-WILLIAMS, *Edge-disjoint spanning trees of finite graphs*, J. London Math. Soc., 36 (1961), pp. 445-450.
- [8] T. NISHIZEKI, T. ASANO AND T. WATANABE, *An approximation algorithm for the Hamiltonian walk problem on a maximal planar graph*, Discr. Appl. Math., 5 (1983), pp. 211-222.
- [9] C. H. PAPADIMITRIOU AND M. YANNAKAKIS, *The clique problem for planar graphs*, Inform. Proc. Lett. 13, 4, 5 (1981), pp. 131-133.
- [10] R. C. READ AND R. E. TARJAN, *Bounds on backtrack algorithms for listing cycles, paths, and spanning trees*, Networks, 5 (1975), pp. 237-252.
- [11] S. TSUKIYAMA, M. IDE, H. ARIYOSHI AND I. SHIRAKAWA, *A new algorithm for generating all the maximal independent sets*, this Journal, 6, (1977), pp. 505-517.

MAXIMUM WEIGHT CLIQUE ALGORITHMS FOR CIRCULAR-ARC GRAPHS AND CIRCLE GRAPHS*

WEN-LIAN HSU†

Abstract. Circle graphs and circular-arc graphs are the intersection graphs of chords and arcs in a circle. In this paper we present algorithms for finding maximum weight cliques in these graphs. The running times of the algorithms are $O(n^2 + m \log \log n)$ for circle graphs and $O(mn)$ for circular-arc graphs. Our algorithms are based on the scanning of appropriate endpoint sequences and efficient bookkeeping of results for subproblems.

Key words. algorithms, graphs

1. Introduction. Let $G = (V, E)$ be a *simple graph*, i.e., a finite, undirected, loopless graph without multiple edges. Let V and E denote the vertex and edge set of G , respectively. Let $n = |V|$ and $m = |E|$. A *clique* is a complete subgraph. An *independent set* is a subset $P \subseteq V$ such that $v_i, v_j \in P$ implies $(v_i, v_j) \notin E$. The complement \bar{G} of G is the graph $\bar{G} = (V, \bar{E})$, where $\bar{E} = \{(x, y) \mid x, y \in V, x \neq y \text{ and } (x, y) \notin E\}$. It is easy to see that an independent set in G is a clique in \bar{G} and vice versa. A *weighted graph* is a simple graph with weights on its vertices.

A graph $G = (V, E)$ is called a *circular-arc graph* if there is a one-to-one correspondence between V and a set S of arcs in a circle such that two vertices are adjacent if and only if the corresponding arcs have a nonempty intersection. S is called an *intersection model* for G . If S is a family of intervals on a real line, G is called an *interval graph*. If S is a family of chords in a circle, G is called a *circle graph*. Clearly, every interval graph is a circular-arc graph since we can represent the intervals by arcs on a circle. Fig. 1 gives a circular-arc graph and its corresponding intersection model. This graph is not an interval graph since we cannot represent the cycle $X_1X_2X_3X_4$ by intervals on the real line. The interval model of a circle graph (which is also called an *overlap graph*) is that (see [1]) each vertex of the graph corresponds to an interval on a real line and each edge corresponds to two intervals overlapping without one being completely contained in the other (i.e., strictly overlapping).

These classes of graphs have drawn considerable attention in recent years [1], [4], [5], [6], [7], [9], [10]. Circular-arc graphs (as a generalization of interval graphs) have a potential role in genetic research [10]. This class of graphs has also been applied to problems in traffic control [11] and computer compiler design [12]. Tucker [13] has shown that recognizing circular-arc graphs can be done in $O(n^3)$ time. Gavril has given a polynomial algorithm for finding a maximum clique in a circular-arc graph. His algorithm involves a procedure for finding a maximum independent set of a bipartite graph, which takes $O(n^{2.5})$ time in the cardinality case [8], and $O(n^3)$ time in the weighted case (using the maximum flow algorithm). Since this procedure is called n times in his maximum clique algorithm, the total complexity is $O(n^{3.5})$ in the cardinality case and $O(n^4)$ in the weighted case. In this paper we present an $O(mn)$ algorithm for the weighted problem assuming that the arc representation of a circular-arc graph is given. The maximum clique problem on circle graphs can also be done in polynomial time once an interval representation is given. Gavril has given an $O(n^3)$ algorithm for

* Received by the editors April 6, 1982, and in revised form October 28, 1983. This research was supported in part by the National Science Foundation under grant ECS-8105989.

† Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, Illinois 60201.



FIG. 1. A circular-arc graph and its arc representation.

the weighted case; Buckingham [1], Rotem and Urrutia [9] have, independently, given $O(n^2)$ algorithms for the cardinality case. We present in § 2 an $O(n^2 + m \log \log n)$ algorithm for the weighted case.

Our clique finding algorithm for circular-arc graphs is much more complicated than that for interval graphs. This is essentially due to the fact that in using the interval model for circular-arc graphs, there are arcs “crossing” both ends of the interval and it becomes harder to count things by scanning the endpoint sequence “only” once. Gavril’s approach to constructing a maximum clique in the neighborhood of each arc exploits very little of the structure of circular-arc graphs. We develop an algorithm based on the scanning of appropriate sets of arcs in a certain order. Using our approach, constructing a maximum clique for the entire graph is about the same order of complexity as constructing a maximum clique in the neighborhood of each arc.

2. An $O(n^2 + m \log \log n)$ maximum weight clique algorithm for circle graphs. In this section, we assume an interval representation of a circle graph is given; namely, each vertex of the graph corresponds to an interval on a real line and two vertices are connected by an edge if and only if the corresponding intervals strictly overlap. An example of a clique is shown in Fig. 2. Without loss of generality, assume all endpoints of the intervals are distinct. For each clique, let us call the interval with the smallest left endpoint the *beginning interval* of the clique and the one with the largest left endpoint the *ending interval*. Our maximum weight clique algorithm goes as follows. For each interval, we compute a maximum weight clique beginning with this interval. A maximum weight clique of the graph can then be chosen as the one with the maximum weight from among these cliques.

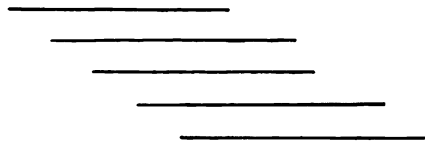


FIG. 2. A clique in a circle graph (overlap graph).

Label the intervals as $1, \dots, n$ according to their ascending left endpoint order. Denote by x_i, y_i , the coordinate of the left, right endpoint of i , respectively. Let the weight of an interval i be $wt(i)$ and the number of intervals strictly overlapping with i be $deg(i)$. In the following we describe an $O(deg(1) \log n)$ algorithm which finds a maximum weight clique beginning with interval 1. The same method is used to find maximum weight cliques beginning with any interval.

First of all we determine the interval s such that $x_s < y_1, x_{s+1} > y_1$ and s is the rightmost. We only have to consider those intervals $2, \dots, s$ which overlap with interval

1. For this purpose we first extract the adjacency representation from the interval representation of the circle graph; this operation takes at most $O(n^2)$ time. Our algorithm scans the intervals $2, \dots, s$ from left to right. For each interval i it scans, the weight (denoted by $\lambda(i)$) of a maximum clique beginning at interval 1 and ending at interval i is computed. After the s th interval is scanned, a maximum weight clique beginning at interval 1 can then be selected.

The computation of $\lambda(i)$ can be carried out by a dynamic programming algorithm. Let $\lambda(1) = \text{wt}(1)$ and $\lambda(i) = 0$ for those interval i with $y_i < y_1$.

LEMMA 1. $\lambda(i) = \text{wt}(i) + \max \{ \lambda(j) \mid 1 \leq j < i \ \& \ y_1 \leq y_j < y_i \}$ for all i s.t. $y_i > y_1$.

Proof. Clearly, if a maximum weight clique beginning at interval 1 and ending at interval i_m consists of the intervals $1, i_0, i_1, \dots, i_m$ (where $1 < i_0 < i_1 < \dots < i_m$), then $\{1, i_0, i_1, \dots, i_{m-1}\}$ is a maximum weight clique beginning at interval 1 and ending at interval i_{m-1} . Thus $\lambda(i_m) = \text{wt}(i_m) + \lambda(i_{m-1})$. It is also clear that $\lambda(i_m) \geq \text{wt}(i_m) + \lambda(j)$ for each j such that $1 \leq j \leq i_m$ and $y_1 \leq y_j \leq y_{i_m}$. \square

Hence, to find $\lambda(i)$ we just have to compare $\lambda(j)$ for all intervals $j < i$ such that $y_j < y_i$. This process can be speeded up by the following observation.

LEMMA 2. If $j_1 < j_2$, $y_{j_1} > y_{j_2}$ and $\lambda(j_1) \leq \lambda(j_2)$, then deleting the information $\lambda(j_1)$ will not change our computation of the weight of a maximum clique beginning with interval 1.

Proof. By Lemma 1, $\lambda(j_1)$ will possibly be considered only when we compute $\lambda(i)$ for some interval i such that $i > j_1$ and $y_i > y_{j_1}$. However, after $\lambda(j_2)$ has been computed we need not consider $\lambda(j_1)$ for any later calculation of $\lambda(i)$ with $i > j_2$ and $y_i > y_{j_1}$. \square

We say j_2 dominates j_1 w.r.t. $\lambda(j)$'s when they satisfy the assumptions in Lemma 2. Hence in the set $\{ \lambda(j) \mid 1 \leq j < i, y_1 \leq y_j \}$ we can delete dominated $\lambda(j)$'s and order the remaining undominated subset $\{ \lambda(j) \mid 1 \leq j < i, y_1 \leq y_j, \lambda(j) > \max_{1 \leq j' < j, y_{j'} \leq y_j} \lambda(j') \}$ into $\lambda(1) = \lambda(i_1) < \dots < \lambda(i_m)$ where $i_0 < i_1 < \dots < i_m$ and $y_0 < y_{i_1} < \dots < y_{i_m}$. In computing $\lambda(i)$ we determine the integer k such that $y_{i_k} < y_i < y_{i_{k+1}}$ and let $\lambda(i) = \text{wt}(i) + \lambda(i_k)$. This formula conforms to Lemma 1 since $\lambda(i_k) = \max \{ \lambda(j) \mid 1 \leq j < i \ \& \ y_1 \leq y_j < y_{i_k} \}$. The presence of the new interval i might create some other dominated intervals $\{ j \mid i_{k+1} \leq j \leq i_m, \lambda(j) < \lambda(i) \}$. This set can easily be identified and deleted by comparing $\lambda(i)$ and $\lambda(i_{k+1}), \dots$ one by one. Hence the insertion of $\lambda(i)$ and following deletions give us a new set of undominated $\lambda(j)$'s arranged in ascending $\lambda(i_j)$, y_{i_j} and the same procedure repeats for the next interval $i+1$. The only work involved at the i th iteration is: (i) determining the integer k , which takes at most $O(\log n)$ time. A more sophisticated implementation of these priority queue operations based on the idea of van Emde Boas [2], [3] would reduce it to $O(\log \log n)$; and (ii) deleting $\lambda(j)$'s dominated by interval i . The former operation has to be repeated at most $\text{deg}(1)$ times; the latter deletions totaled at most $\text{deg}(1)$ times. Hence the entire procedure of finding a maximum weight clique beginning at interval 1 takes at most $O(\text{deg}(1) \log \log n)$ time. Repeating this procedure for every interval would take at most $\sum_i O(\text{deg}(i) \log \log n) = O(m \log \log n)$ time. The running time of the maximum weight clique algorithm on circle graphs is therefore bounded by $O(n^2 + m \log \log n)$, where $O(n^2)$ comes from the adjacency representation.

3. An analysis of cliques in a circular-arc graph. Without loss of generality, assume that all endpoints of the n arcs are distinct. Let the length of the circle be 1; we can assume all arcs have lengths less than 1. Arbitrarily choose an arc from the collection. Starting from the counterclockwise end of this arc, we can label all ends along the clockwise direction as $1, 2, \dots, 2n$. Specify arbitrarily an order for the arcs and denote

arc i by (a_i, b_i) , where a_i is the label of its counterclockwise end and b_i , the label of its clockwise end. Note that a_i can be larger than b_i , in which case the arc (a_i, b_i) extends across $a_i, a_{i+1}, \dots, 2n, 1, \dots, b_i$. We also use (y, z) to denote the arc segment of the circle with counterclockwise end y and clockwise end z . A label x is said to be in (y, z) if its corresponding end falls in the open arc segment (y, z) (i.e., either (i) $y < x < z$ or (ii) $y > z$ & $(x > y$ or $x < z)$).

Two arcs $(a_i, b_i), (a_j, b_j)$ are said to *intersect strictly* if a_i is in (a_j, b_j) and b_j is in (a_i, b_i) or the above holds with i, j reversed (intuitively, i and j overlap but none is properly contained in the other). Two strictly intersecting arcs i and j are said to form a *Type I pair* if together they do not cover the whole circle; otherwise, they are said to form a *Type II pair*. These two types are depicted in Fig. 3. The next lemma gives a classification of the cliques in a circular-arc graph.

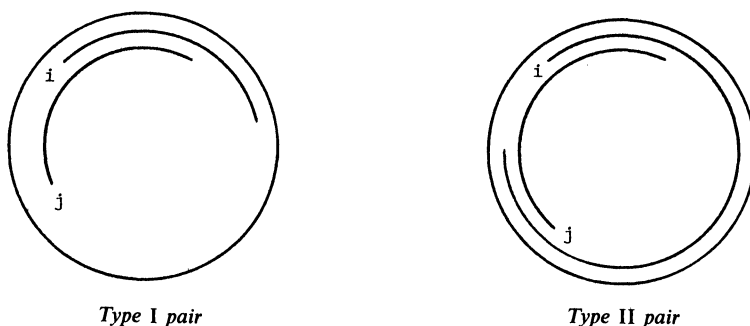


FIG. 3. Two types of pairs in P .

LEMMA 3. Let $(a_i, b_i), \dots, (a_k, b_k)$ be a collection A of arcs which form a clique in a circular-arc graph G . Then one of the following two conditions is true:

- (i) There exists an arc (a_i, b_i) which is properly contained in every other arc in A .
- (ii) There exist two strictly intersecting arcs $(a_i, b_i), (a_j, b_j)$ in A such that no arc of A is properly contained in either one of them and for every other arc (a_k, b_k) in A exactly one of the following conditions is true:

- (a) (a_k, b_k) properly contains (a_i, b_i) ;
- (b) a_k is in (a_j, a_i) ; b_k is in (b_j, b_i) ;
- (c) a_k is in (a_i, b_j) ; b_k is in (b_i, a_k) ;
- (d) a_k is in (b_j, b_i) and also in (b_j, a_j) ; b_k is in (a_j, a_k) ;
- (e) b_i is in (b_j, a_j) ; a_k is in (b_i, a_i) ; b_k is in (b_j, b_i) (note: in this case arcs i and j must form a Type I pair).

Proof. Let B be the subcollection of A that contains all arcs which do not properly contain any other arc of A . If B contains a single arc (a_i, b_i) , then condition (i) holds. Hence assume B contains more than one arc. If there exists a Type II pair i, j in B then choose this pair. Otherwise pick any arc i in B . Starting from the counterclockwise end a_i of i , we can find an arc j in B such that a_j is the most counterclockwise end before b_i . If a_j is in (a_i, b_i) , then we interchange the indices i and j . Note that there can be no arc in B with its counterclockwise end in (b_i, a_j) but its clockwise end in (a_i, b_j) .

The proof for condition (ii) does not depend on whether i and j form a Type I pair or a Type II pair except that condition (ii) (e) is possible only for Type I pairs. An illustration for each subcase is depicted in Fig. 4 for Type I pairs i and j .

Now consider an arc (a_k, b_k) in A such that $k \neq i, k \neq j$. Suppose (a_k, b_k) does not properly contain (a_i, b_i) . Consider the following cases.

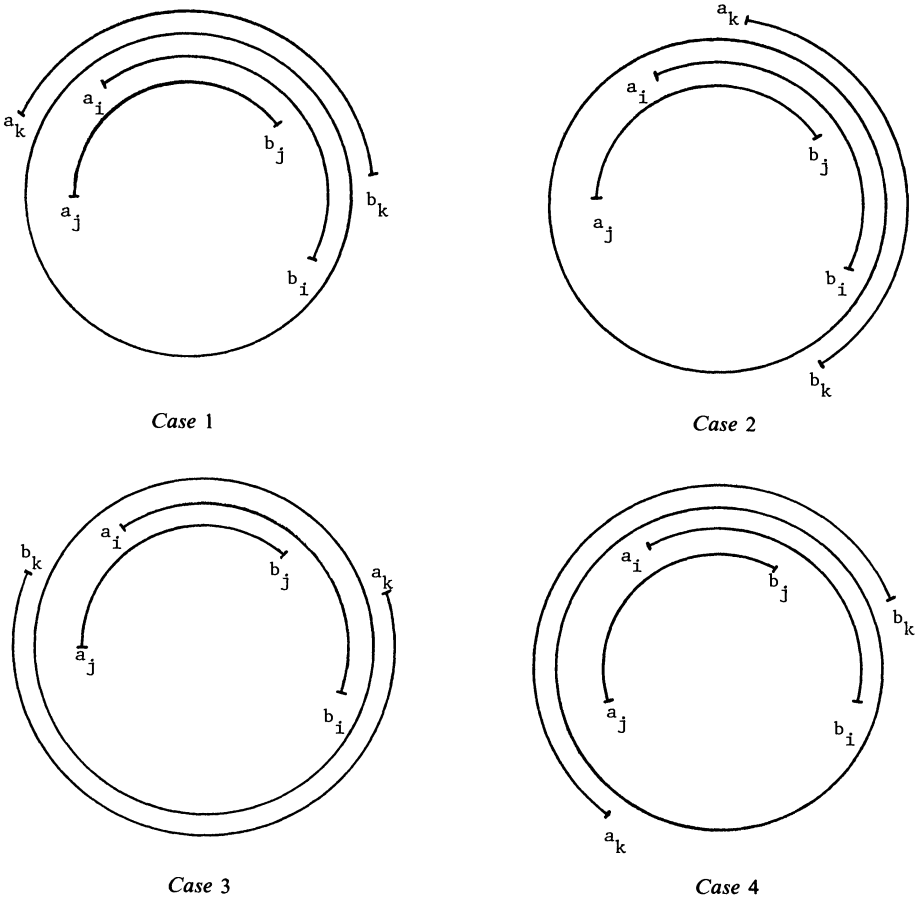


FIG. 4. Examples of condition (ii) with Type I pairs i and j .

Case 1. a_k is in (a_j, a_i) . Since (a_k, b_k) cannot be properly contained in (a_j, b_j) we have b_k in (b_j, a_k) . Since (a_k, b_k) does not properly contain (a_i, b_i) we conclude that b_k is in (b_j, b_i) . This implies condition (b).

Case 2. a_k is in (a_i, b_j) . Since (a_k, b_k) cannot be properly contained in (a_i, b_i) , we have b_k in (b_i, a_k) . This implies condition (c).

Case 3. a_k is in (b_j, b_i) and also in (b_j, a_j) . Since (a_k, b_k) cannot be properly contained in (a_i, b_i) , we have b_k in (b_i, a_k) . Since (a_k, b_k) must intersect (a_j, b_j) , we have b_k in (a_j, a_k) . This implies condition (d).

Case 4. None of the above cases apply. This can only happen when i, j form a Type I pair and a_k is in (b_i, a_j) . Since (a_k, b_k) must intersect (a_i, b_i) , we have b_k in (a_i, a_k) . However, b_k cannot be in (a_i, b_j) because then (a_k, b_k) would contain some $(a_s, b_s) \in B$ (could be itself) with a_s in (b_i, b_j) , b_s in (a_i, b_j) , which is impossible by the selection of (a_i, b_i) and (a_j, b_j) . Hence b_k must be in (b_j, a_k) . Since (a_k, b_k) does not properly contain (a_i, b_i) , we have b_k in (b_j, b_i) , which is condition (e). \square

By Lemma 3, a clique in a circular-arc graph satisfies either condition (i) or condition (ii). To find a maximum weight clique in G it suffices to find a maximum weight clique among all cliques satisfying condition (i) and another one among all cliques satisfying condition (ii). This is discussed in the next section.

4. An $O(mn)$ maximum weight clique algorithm for circular-arc graphs. Consider a circular-arc graph G represented by its arc intersection model. First of all, we construct a “containment graph” H from G with directed edges. Each vertex of H corresponds to an arc in G . An edge (i, j) directed from i to j appears in H iff arc i is properly contained in arc j . H is a transitive graph. The number of arcs which properly contain an arc i equal to the out-degree of i . Thus, it is not difficult to find a maximum weight clique among all cliques satisfying condition (i) of Lemma 3.

Next, we calculate the weight of a maximum clique satisfying condition (ii) of Lemma 3. A clique for which condition (ii) applies is said to be *generated* by its two arcs $(a_i, b_i), (a_j, b_j)$. It should be noted that such a clique does not contain any arc which is properly contained in (a_i, b_i) or (a_j, b_j) and its weight is certainly bounded by a maximum weight clique generated by $(a_i, b_i), (a_j, b_j)$. Since there is no advance information about which pair of arcs will generate a maximum weight clique satisfying condition (ii), we simply consider the set P of all pairs of arcs that strictly intersect and compute a maximum weight clique generated by each such pair.

For each pair (i, j) in P , we first compute the number of arcs satisfying one of conditions (a), (b), (c), (d) and (e). Arcs satisfying (a) or (c) can be determined by the containment graph H . Arcs satisfying (b), (c) and (d) can be found by searching through every arc in G , which takes $O(n)$ time. Denote the total weight of arcs satisfying (d) by $\lambda_d(i, j)$. Since there can be at most $O(m)$ pairs in P , this part of the algorithm takes at most $O(mn)$ time. (It can be carried out in $O(n^2 \log n)$ time by using a more sophisticated data structure. However the remaining part still takes $O(mn)$ time.) It remains to show that the remaining part can also be done in $O(mn)$ time.

Further analyzing condition (ii), one can see that every arc satisfying condition (a), (c) or (e) will intersect with any other arc satisfying condition (ii), but arcs satisfying (b) might not intersect arcs satisfying (d). Hence we still have to make a careful selection with regard to which arc satisfying (b) or (d) should be included in order to form a clique with the maximum weight. The optimal selection from arcs satisfying (b) or (d) will be determined separately for different types of pairs in P .

Let us first consider Type I pairs. For each arc i we determine an optimal selection for each Type I pair (i, j) in P . Denote the total weight of arcs selected (including the weight of j but not i) by $\lambda(i, j)$. It should be noted that once the selection of arcs satisfying (b) is determined, the corresponding set of arcs satisfying (d) which can be included in a clique is also determined. Hence it is sufficient to consider the optimal selection of arcs satisfying (b). We will use a dynamic programming procedure which recursively finds $\lambda(i, j)$ for all arcs j such that (i, j) is a Type I pair in P . Each Type I pair (i, j) satisfies that a_j is in (b_i, a_i) . Therefore we can arrange all these arcs into a list $L_i = \{i_1, \dots, i_m\}$ according to the counterclockwise order of a_i . Figure 5 shows an example of a possible arc j forming Type I pair (i, j) with arc i .

We now describe the recursive formula used in our algorithm. Suppose we have computed $\lambda(i, i_t)$ $t = 1, \dots, k-1$ and want to determine $\lambda(i, i_k)$. For each $t \in \{1, \dots, k-1\}$ such that y_t is in (b_{i_k}, b_i) , let $\alpha(i_t)$ be the total weight of arcs in $\{i_{t+1}, \dots, i_{k-1}\}$ whose clockwise ends are in (b_i, b_i) (or equivalently, which properly contain the arc i_t).

LEMMA 4.

$$\lambda(i, i_k) = \max \left\{ \max_{\substack{t \in \{1, \dots, k-1\} \\ b_i \text{ is in } (b_{i_t}, b_i)}} [\lambda(i, i_t) + \alpha(i_t) + \lambda_d(i, i_k)], \lambda_d(i, i_k) \right\}.$$

Proof. Consider an optimal selection $S = \{i, i'_1, i'_2, \dots, i'_l, i_k\}$ (ordered by their counterclockwise ends). If $l = 0$, then $\lambda(i, i_k) = \lambda_d(i, i_k)$. Otherwise proceed as follows.

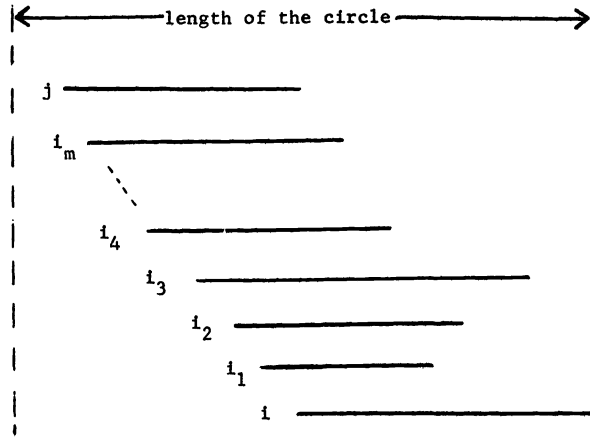


FIG. 5. An example of arcs between Type I pair (i, j) .

Starting from i'_t , we can find the first arc, say i'_q , such that it does not properly contain any other arc in the selection S . Since S is an optimal selection for the pair (i, i_k) , the following must be true:

1. The set $\{i, i'_1, \dots, i'_{q-1}, i'_q\}$ must be an optimal selection for the pair (i, i'_q) . This optimal selection results in a total weight $\lambda(i, i'_q)$.
2. The set $\{i'_{q+1}, \dots, i'_t\}$ consists of all those arcs in the ordered subset $\{i'_q (= i_r), i_{r+1}, \dots, i_{k-1}\}$ which properly contain the arc i'_q . The total weight of this set is $\alpha(i'_q)$.
3. The set of arcs satisfying (d) for (i, k) whose counterclockwise ends are in $(y_{i_k}, y_{i'_q})$ is included in the clique. The total weight of these arcs is $\lambda_d(i'_q, i_k)$.

The total weight of the optimal selection S is thus $\lambda(i, i'_q) + \alpha(i'_q) + \lambda_d(i'_q, i_k)$, which must be $\max_{t \in \{1, \dots, k-1\}} [\lambda(i, i_t) + d_k(i_t) + \lambda_d(i_t, i_k)] \quad \square$

Our algorithm for computing $\lambda(i, j)$ on Type I pairs goes as follows. Start with an arc i . Scan the list L_i from left to right. Whenever an arc i_k is scanned, execute the following:

1. For each $t \in \{1, \dots, k-1\}$ such that b_{i_t} is in (b_{i_k}, b_i) , calculate $\lambda(i, i_t) + \alpha(i_t) + \lambda_d(i_t, i_k)$. Take the maximum out of these numbers and $\lambda_d(i, i_k)$ to be $\lambda(i, i_k)$.
 2. For each $t \in \{1, \dots, k-1\}$ such that b_{i_t} is in (a_i, b_{i_k}) , augment $\alpha(t)$ by $wt(i_k)$.
- The calculation of each $\lambda(i, i_t)$ takes at most $O(n)$ time. Since there are $O(m)$ pairs at most, the computation of all $\lambda(i, j)$ can be done in $O(mn)$ time.

Next, we consider Type II pairs. For each arc j we determine an optimal selection for each Type II pair (i, j) in P . Again, denote the total weight of arcs selected (including the weight of i but not j) by $\lambda(i, j)$. Since it is possible that the same clique can be generated both by a Type I pair and by a Type II pair, we will now be concerned with those cliques which cannot be generated by Type I pairs.

LEMMA 5. *If a clique contains two arcs i, j such that (i, j) is a Type I pair in P and both i and j do not properly contain any other arc in the clique, then this clique can be generated by a Type I pair in P by condition (ii) of Lemma 3.*

Proof. Consider the subset B of arcs which do not properly contain any other arc of the clique. Then $i, j \in B$. Starting from the left endpoint of i , we search for an arc k in B such that a_k is the most counterclockwise end before b_i . The existence of such an arc k is ensured by the existence of the arc j . (i, k) is a Type I pair and the clique can be generated by (i, k) . \square

Now, if an optimal selection for a Type II pair (i, j) contains an arc k satisfying condition (d) with b_{i_k} in (a_j, a_i) , then we claim that the corresponding clique can be generated by a Type I pair. First, the two arcs j and k do not cover the whole circle. Pick an arc, say k' , contained in arc k which does not properly contain any other arc in this clique. Clearly, (j, k') is a Type I pair in P . By Lemma 5, this clique can be generated by a Type I pair according to condition (ii) of Lemma 3. Hence we do not have to consider the inclusion of arcs satisfying condition (d) with their clockwise ends in (a_j, a_i) . This leads us to consider arcs satisfying (b) and arcs satisfying (d) with their clockwise ends in arcs extending between a_i and their counterclockwise ends. However, these two types of arcs always intersect and there is no need to make any selection. Therefore, for each Type II pair (i, j) , the clique we need to consider includes arcs satisfying (a), (b), (c), (e) together with those arcs satisfying (d) with their right endpoints between x_i and their left endpoints.

Having determined an optimal selection for each Type I pair, formed the clique for each Type II pair as above and calculated the maximum total weight of arcs containing the same arc i for each arc i , we simply choose a clique with the maximum weight as our maximum weight clique. The time bound of our algorithm is $O(mn)$. The space bound is $O(m)$, the number of pairs in set P .

5. Conclusion. In this paper we present two algorithms for finding maximum weight cliques on circle graphs and circular-arc graphs. Both algorithms make use of dynamic programming techniques. It remains to be seen whether more sophisticated data representation could produce faster algorithms. For the case of circle graphs, we have not been able to reduce the complexity of the weighted case down to $O(n^2)$. We suspect that the weighted case is, indeed, harder than the cardinality case, which can be done in $O(n^2)$ time. For circular-arc graphs, there are some steps which can be carried out in $O(n^2 \log n)$ time, but the dominating step is in selecting arcs satisfying conditions (b) or (d) for Type I pairs, which takes $O(mn)$ time.

6. Acknowledgment. We are grateful to an anonymous referee who suggested two reductions in the time bounds of our algorithms.

REFERENCES

- [1] M. A. BUCKINGHAM, *Efficient stable set and clique finding algorithms for overlap graphs*, Dept. Computer Science, New York Univ., New York, 1981.
- [2] P. VAN EMDE BOAS, *Preserving order in a forest in less than logarithmic time and linear space*, Inform. Proc. Letters, 6 (1977), pp. 80-82.
- [3] P. VAN EMDE BOAS, R. KAAS AND E. ZIJLSTRA, *Design and implementation of an efficient priority queue*, Math. Systems Theory, 10 (1977), pp. 99-127.
- [4] M. R. GAREY, D. S. JOHNSON, G. L. MILLER AND C. H. PAPADIMITRIOU, *The complexity of coloring circular arcs and chords*, SIAM J. Alg. Discr. Meth., 1 (1980), pp. 216-228.
- [5] F. GAVRIL, *Algorithms on circular-arc graphs*, Networks, 4 (1974), pp. 357-369.
- [6] ———, *Algorithms for a maximum clique and a maximum independent set of a circle graph*, Networks, 3 (1973), pp. 261-273.
- [7] U. I. GUPTA, D. T. LEE AND J. Y-T. LEUNG, *Efficient algorithms for interval graphs and circular-arc graphs*, Networks, 12 (1982), pp. 459-467.
- [8] J. HOPCROFT AND R. M. KARP, *An $n^{5/2}$ algorithm for maximum matching in bipartite graphs*, this Journal, 2 (1973), pp. 225-231.
- [9] D. ROTEM AND J. URRUTIA, *Finding maximum cliques in circle graphs*, Networks, 11 (1981), pp. 269-278.
- [10] F. W. STAHL, *Circular genetic maps*, J. Cell Physiol., 70 (Suppl. 1) (1967), pp. 1-12.
- [11] K. E. STOFFERS, *Scheduling of traffic lights—a new approach*, Transportation Res., 2 (1968), pp. 199-234.
- [12] A. TUCKER, *Coloring a family of circular-arc graphs*, SIAM J. Appl. Math., 29 (1975), pp. 493-502.
- [13] ———, *An efficient test for circular-arc graphs*, this Journal, 9 (1980), pp. 1-24.

NEW DATA STRUCTURES FOR ORTHOGONAL RANGE QUERIES*

DAN E. WILLARD†

Abstract. Consider a set of N records corresponding to points in k -dimensional space ($k \geq 2$). This article introduces one new data structure which uses memory $O(N \log^{k-1} N)$ for supporting orthogonal range queries with worst-case complexity $O(\log^{k-1} N)$ and several modifications of this proposal for a dynamic environment. These results are especially useful when $k = 2$.

Key words. augmented tree, data base, geometric retrieval, k -fold tree, multidimensional retrieval, orthogonal range query, range query, relational data base, super- B -tree, partial match retrieval, decomposable data structure

1. Introduction. Throughout this paper, S will denote a set of N records, each corresponding to a point in k -dimensional space, and q will denote a request for the subset of S satisfying a condition similar to:

$$(1) \quad a_1 < \text{KEY} \cdot 1 < b_1 \ \& \ a_2 < \text{KEY} \cdot 2 < b_2 \ \& \ \cdots \ \& \ a_k < \text{KEY} \cdot k < b_k.$$

Such requests q are called *orthogonal range queries* and we use the following notation:

- (i) SET(q) will denote the subset of the initial file that satisfies q ;
- (ii) COUNT(q) will denote the number of records belonging to SET(q);
- (iii) SUM(q) will denote the sum under any group-operator of some designated value stored in the records belonging to SET(q).

The *locate-and-copy time* of a specified retrieval algorithm will be defined as the amount of runtime needed by the procedure to find and transfer the members of SET(q) into the user's workspace. This concept is not very useful because the degenerate case where COUNT(q) = N forces all procedures to have an $O(N)$ *worst-case* locate-and-copy time for querying a file of N elements. Consequently, the literature of multidimensional retrieval has employed other measurements of complexity in its worst-case analyses; this paper will rely on the following three measurements:

(1) *Worst-Case Locate Time.* A retrieval algorithm will be said to respect the worst-case locate-bound $O(f(N))$ if it will require no more than time $O(f(N) + \text{COUNT}(q))$ for copying all the elements of SET(q) into the user's workspace. (Worst-case locate time is more meaningful than worst-case locate-and-copy time because the new definition corrects for the copy time, which must be proportional to COUNT(q).

(2) *Aggregate-Scan Time.* A retrieval algorithm will be said to respect the aggregate-scan bound $O(f(N))$ on a specified data structure if this algorithm needs no more than time $O(f(N))$ to scan this data structure and calculate SUM(q) and COUNT(q), for any query q .

(3) *Expected Locate Time.* Essentially, this bound represents the expected locate portion of the locate-and-copy task, where this component is defined in the most conservative possible sense. That is, suppose for any query q there exists integers J_q , whose *worst-case* value lies in $O(f(N))$, and T_q , whose *expected* value lies in $O(f(N))$, such that a retrieval algorithm can find in time T_q a collection of J_q lists L_1, L_2, \dots, L_{J_q}

* Received by the editors March 12, 1982, and in revised form August 27, 1983. This research was supported in part by the Office of Naval Research under contract N00014-76-C-0914 while the author was at Harvard University. The Harvard Aiken Laboratory Report TR-22-78 indicates the status of the research at the time when I left Harvard (1978).

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

and two indices I_i^{INF} and I_i^{SUP} for each list such that the members of $\text{SET}(q)$ consist precisely of the disjoint union of the portions of lists L_1, \dots, L_{J_q} that lie between the list's indices I_i^{INF} and I_i^{SUP} . Then, this retrieval algorithm will be said to respect the expected locate bound of $O(f(N))$.

The previous literature did not use this terminology, but its results can be readily translated into it. Various data structures have been proposed, some of which are more efficient in time and others in space. Some of the more space-efficient data structures include k - d trees, quad trees, and box array hashing. The first two data structures, originally proposed in [Be75], [FiBe74], were shown in [LeWo77] to have a worst-case orthogonal locate complexity $kN^{1-1/k}$ when the trees are perfectly balanced. ([Wi78c], [Wi80] discuss some surprising aspects of imperfectly balanced variants of these trees.) Techniques similar to box-array hashing [Bo81] are efficient when the range query spans a very small geometric region. Nonoverlapping k -ranges improve retrieval time by modestly increasing memory space, and overlapping k -ranges illustrate the potential of quite large memory spaces [BeMa80]. For any $p > 1$, it is possible to construct an overlapping k -range that uses space $O(N^p)$ and has a worst-case locate complexity $O(\log N)$; since the time and space coefficients inside the O -notation become quite large as p decreases, this very nice asymptotic result is unlikely to have many practical applications. Although [LeWo77], [BeMa80] confined their proofs technically to worst-case locate time, they easily generalize to worst-case aggregates.

[Be80], [BeSh77], [LeWo80], [Lu79], [LuWi82], [Wi78a], [Wi79] and [WiLu84] have studied how to optimize orthogonal range query time in a memory space that lies intermediate between the extremes of the space $O(N^p)$ for overlapping k -ranges and the space $O(N)$ of the other data structures mentioned above. Each of these articles established some query time of the form $O(\log^k N)$ in space $O(N \log^{k-1} N)$. Since they were written during overlapping periods of time, several different names have been assigned to the basically similar data structures proposed in these articles. We will use the term first-generation k -fold trees to refer to these data structures. This paper will introduce a more refined concept, called the second-generation k -fold tree, which will occupy the same space as first generation k -fold trees but improve by a factor of $\log N$ most of the time complexities of k -fold trees.

The precise results known about k -fold trees prior to the first draft of this paper [Wi78b] are listed below:

(A) $\text{SUM}(q)$ and $\text{COUNT}(q)$ can be calculated in aggregate-scan time $O(\log^k N)$. (This result first appeared in [BeSh77] and is explained more fully in [Be80].)

(B) $\text{SET}(q)$ can be retrieved in worst-case locate time $O(\log^k N)$. (This result was explicitly discussed in [Be80], [LeWo80], [Wi78a] and implicitly in most of the other articles.)

(C) Any sequence of n insertion and deletion commands can be executed in worst-case time $O(n \log^k N)$ on a first-generation k -fold-tree data structure whose cardinality never exceeds N and which is initially empty [Lu78], [Lu79], [LuWi82], [Wi78a], [Wi79], [WiLu84].

(D) The result in (C) can be strengthened to indicate the existence of a procedure that executes individual insertion and deletion commands in strict worst-case time $O(\log^k N)$ [Lu79], [Wi78a], [Wi79], [WiLu84].

(E) Several of the results above can have their runtimes reduced by a factor of $\log N$ in a *batch environment* where N operations on a set of size N are simultaneously performed. Such batch procedures include:

- (1) an algorithm that constructs an entire k -fold data structure in time $O(N \log^{k-1} N)$ [BeSh77], [Be80]; and

- (2) a procedure that calculates estimated distribution functions in time $O(N \log^{k-1} N)$ [Be80], [BeSh77]; also, given a batch of N queries q_1, q_2, \dots, q_N , this procedure can calculate all their SUM (q) and COUNT (q) values in the same time.

Note that except for (E), all the items above discuss the complexities of retrievals, insertions and deletions in an on-line (as opposed to batch) environment. Although the articles cited in these items discussed technically only worst-case complexities, their algorithms, in fact, had a complexity $\log^k N$ in both the best and worst cases. In this article, we show that it is possible to produce an almost across-the-board factor $\log N$ reduction in complexity, thus deriving the new magnitude $O(\log^{k-1} N)$ for on-line processing in any dimension $K \geq 2$. We say "almost" because our algorithms do not reduce aggregate-scan complexities in a dynamic environment. Fredman has proven that the latter reduction is impossible at least for the case of aggregates calculated over semigroups [Fr81a]. However, without increasing memory space, we do improve the worst-case complexity of aggregate-scan and locate-retrievals in a static environment, and alternatively the complexity of insertions, deletions, and locate-retrievals in a dynamic environment. The latter factor $\log N$ improvement in expected complexity can be accompanied by controls ensuring a worst-case retrieval complexity $O(\log^{k-1/2} N)$ and a worst-case insertion-deletion time $O(\log^k N)$. It can also guarantee $O(\log^{k-1/2} N)$ worst-case bounds on insertion, deletion, and locate-retrieval operations. Essentially we obtain these improvements and more by proposing seven new data structures which provide different advantages in a dynamic environment.

Three of our seven new data structures were proposed in the first draft of this paper [Wi78b], and the other four were developed subsequently. No improvements over our data structures are known in the aggregate model of retrieval, but recently [Ch83] extended results from [Ed81, Mc81] and showed that a factor $\log \log N$ savings in memory is possible for locate-retrievals if the copy time in the locate-and-copy complexity is increased by a factor of 2 and the memory space coefficient expands by an amount often exceeding $\log \log N$'s typical value. Section 8 describes the different advantages of our seven data structures and Chazell's alternative.

The discussion in this paper will be reasonably self-contained. Some of the sections at the end will use Dietz's theorem [Di84], the bounded balance method [WiLu84] and q -fast tries [Wi81]. The reader will be able to follow the gist even if he is unfamiliar with this material. Our techniques are likely to have further applications beyond those discussed here. For instance, our concept could improve some of the complexities from [AvSh81] by a factor of $\log N$ and it is partially relevant to detecting rectangle intersections [BeWo80], [Ed80], [LeWo81], [VaWo80]. Section 9 of Overmars' dissertation [Ov83] illustrates how our data structure $T_e(2)$ can be applied to queries about the past, and [EdOv83] shows how to reduce its memory to $O(N)$ in a batch environment. [Wi78a], [Wi83a, 83b, 84a, 84b] describe the principal practical application of this article by illustrating how orthogonal range queries are very relevant to commercial data bases.

Lower bounds for orthogonal queries are discussed in [Fr81a], [Fr81b], [Ya82]. References [EdKiMa82], [EdWe83], [CoYa83], [Wi82], [Ya83] discuss the variations of this problem for nonorthogonal regions, such as polygons and polytopes. Although our theorems apply to any dimension $k \geq 2$, they are especially practical when $k = 2$ because the memory cost $O(N \log^{k-1} N)$ can become prohibitive otherwise.

2. Review of earlier literature and some useful intuitions. How do second-generation k -fold trees differ from their first-generation predecessor? This section will

introduce the notations that enable us to discuss both concepts and their distinction. Henceforth S will denote a set of N distinct k -tuples and $T_1(k, S)$ a first-generation k -fold tree for representing this set, similar to any of the structures mentioned in [BeSh77], [Be80], [LeWo80], [Lu78], [Lu79], [LuWi82], [Wi78a], [Wi79], [WiLu84]. The seven new data structures proposed in the article will be distinguished by their subscripts, thus denoted as $T_e(k, S)$, $T_a(k, S)$, $T_d(k, S)$, etc.

Sometimes for simplicity, we will omit the symbols k and S when discussing our new data structure; that is, T_i and $T_i(k)$ will sometimes serve as abbreviations for $T_i(k, S)$. The differences between our seven proposed data structures will become apparent in the course of our discussion. Before defining these new data structures, it is useful to review the definition of the first-generation data structure T_1 . This review, as well as most of the other discussion in this paper, will take place in the context of the dimension $k = 2$; this dimension provides both the simplest example of nontrivial k -fold trees and their most practical example. Generalization to the case of higher dimensions will take place at the end of this article.

The first-generation data structure $T_1(2, S)$ will consist of two basic parts. Its first section will be a binary tree of height $O(\log N)$ where the records are stored at the leaf-level in order of increasing $\text{KEY} \cdot 2$ value. This section will be called the base-tree and be denoted by B . The second part of the 2-fold tree $T_1(2, S)$ will consist of a series of auxiliary fields. Each node v in the base-tree B will be associated with an auxiliary field $\text{AUX}(v)$, consisting of a binary tree of height $O(\log N)$ that represents the subset of S which descends from v in the base-tree B in order of increasing $\text{KEY} \cdot 1$ value. Each node w in those auxiliary-field trees may (optionally) contain aggregate information about its descendants (such as a field indicating the number of records descending from w or sum of some special value stored in these descendants).

A data structure similar to a first-generation k -fold tree was first introduced by Bentley and Shamos, in [BeSh77], and it was subsequently explored by Bentley, Lee, Lueker, Willard, and Wong, as mentioned in the Introduction. We will now review how orthogonal range queries may be performed in this tree $T_1(2, S)$.

Define a node v in the base-tree B to be *critical* with respect to the condition $a_2 < \text{KEY} \cdot 2 < b_2$ iff all descendants of v satisfy this range condition but the same is not true of v 's father. [BeSh77], [Be79] and [LeWo80] have noted that since the base trees have heights $O(\log N)$, they contain no more than $O(\log N)$ critical nodes, all of which can be found in a single search consuming time $O(\log N)$. The first step of the two-dimensional orthogonal-range-query algorithm will find these critical nodes. The second step will search on the range condition $a_1 < \text{KEY} \cdot 1 < b_1$ the auxiliary fields of these critical nodes. The precise nature of this search will depend on whether the query consists of a request for $\text{SUM}(q)$, $\text{COUNT}(q)$ or $\text{SET}(q)$ (see [BeSh77], [Be79] and [LeWo80] for more details). In all cases, an overhead search time $O(\log N)$ will be needed to search each auxiliary field—thereby producing a total complexity $O(\log^2 N)$ for searching the $\log N$ distinct auxiliary fields.

Note that this algorithm is repetitive in the sense that it applies the same search-query $a_1 < \text{KEY} \cdot 1 < b_1$ to each of the auxiliary fields. In this paper, we show how modified forms of k -fold trees that connect the auxiliary fields with special types of pointers can usually avoid such repetitive searching, thereby reducing the query time complexities by a factor of $\log N$. These reductions will usually be possible, but not always since they would then violate Fredman's lower bound [Fr81a].

3. Log N improvements of the time with downpointers. Throughout this paper the term *augmented tree* will refer to a data structure, similar to first-generation 2-fold

trees, consisting of a binary tree of logarithmic height, called the base, where the records are stored at the leaf-level and sorted by increasing order of one specified key, and a set of auxiliary fields where the particular field $AUX(v)$, associated with a base-tree node v , provides an alternate description of v 's descendants. The representative of a particular record R in one of the augmented tree's auxiliary fields will be called R 's *entry*. Throughout our discussion, we assume each auxiliary field $AUX(v)$ has precisely one entry for each leaf descending from v . All auxiliary fields in the next four sections will be sorted by $KEY \cdot 1$, and the term "pseudo-entry" will refer to an especially stored termination mark placed in the rightmost position of these auxiliary fields with the value $+\infty$ assigned to its $KEY \cdot 1$ value. The symbols X , Y and Z will denote typical entries or pseudo-entries in an auxiliary field. The i th keys of these entries will be denoted $X \cdot i$, $Y \cdot i$ and $Z \cdot i$. The symbols v_l and v_r will denote the left and right sons of an internal node v from the base-tree. Also for each entry X belonging to $AUX(v)$, its *left-down-son* is defined as that item Y in $AUX(v_l)$ with the smallest $Y \cdot 1$ value satisfying $Y \cdot 1 \geq X \cdot 1$. The *right-down-son* of X will be defined as the analogous entry in the field $AUX(v_r)$. Pseudo-entries have been introduced into our data structure because they guarantee every entry will have a nonnull left- and right-down-son, a fact needed later to guarantee our algorithm will never get lost during a search.

Now we define the two new data structures, $T_e(2, S)$ and $T_a(2, S)$, which essentially are augmented trees where every entry X contains two special pointers to its left- and right-down-sons. These special pointers will enable us to avoid the repetitive searching of § 2 and thereby provide a factor $\log N$ saving in retrieval time. For simplicity, until the end of the paper, we assume no two elements of S have the same value stored in either their $KEY \cdot 1$ or $KEY \cdot 2$ fields. The generalization to the case of repeating values is easy and will appear in § 7.

A second-generation 2-fold tree $T_e(2, S)$ will be defined as an augmented tree data structure with three parts. Its first two components are the usual sections of a logarithm height base-tree B sorted by $KEY \cdot 2$ and auxiliary fields sorted by $KEY \cdot 1$. The latter will be stored in the format of a doubly-linked list. In addition to the forward and back pointers required by a doubly-linked ordered list, each entry in $AUX(v)$ will contain a pointer to its left- and right-down-son. These fields will be called the *left- and right-downpointers*. The third part of the second-generation tree $T_e(2)$ will be its dictionary D ; this will consist of a binary tree of height $O(\log N)$ that indexes the entries in the root's auxiliary field by $KEY \cdot 1$. An example of a second-generation tree $T_e(2)$ is illustrated in Fig. 1. The main difference between this data structure and first-generation trees is that the tree $T_e(2)$ contains the new left- and right-downpointer fields. The pseudo-entry ∞ serves as a termination mark at the right end of each auxiliary field. This mark guarantees every node has a nonnull left- and right-down-son, as is illustrated in Fig. 1.

Now, we introduce a definition which will play a major role in our algorithm descriptions. Recall that a node in the base-tree B is said to be *critical* with respect to the condition $a_2 < KEY \cdot 2 < b_2$ iff all its descendants satisfy this condition but the same is not true for its father. Let $CRITICAL(a_2, b_2)$ denote the set of nodes critical to this range condition and $LEAST(v, c)$ the smallest entry in $AUX(v)$ satisfying $KEY \cdot 1 > c$. Then the *critical entry set* relative to the query $a_2 < KEY \cdot 2 < b_2$ and the constant c will be defined as the set $\{LEAST(v, c) | v \in CRITICAL(a_2, b_2)\}$. This set will be denoted as $CES(c, a_2, b_2)$; we will prove that it is possible to retrieve the full set of records from any critical entry set in time $O(\log N)$. Most of our other time-complexities will be a consequence of this retrieval theorem.

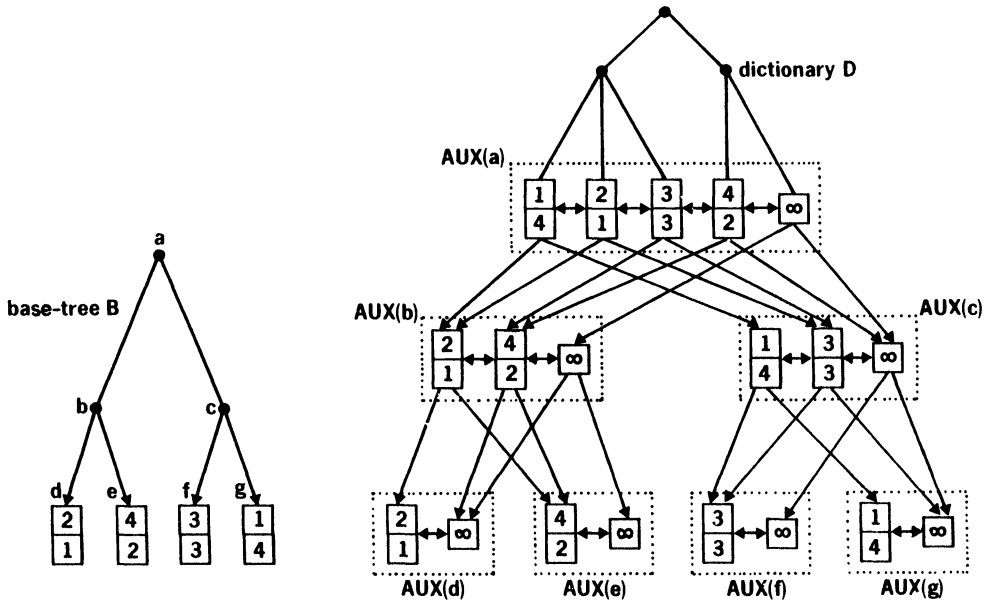


FIG 1. An illustration of a tree $T_e(2)$ whose 4 records are the ordered pairs: (1, 4), (2, 1), (3, 3) and (4, 2).

One preliminary definition should be introduced. For any entry X belonging to the root's auxiliary field, the *entry-tree* of X , denoted $t(X)$, will be defined to be a binary tree whose root is X , whose depth-1 nodes are the left- and right-down-sons of X , whose depth-2 nodes are the sons of these sons, and so on. By the definition of the data structure $T_e(2)$, all its entry-trees are isomorphic to the base-tree B . Theorem 1 shows how to exploit this isomorphism to retrieve critical entry sets efficiently.

THEOREM 1. For any three constants a_2, b_2 and c with $a_2 \leq b_2$, the second-generation k -fold tree $T_e(2)$ makes it possible to retrieve the set $CES(c, a_2, b_2)$ in worst-case time $O(\log N)$.

Proof. The algorithm for retrieving $CES(c, a_2, b_2)$ consists of four steps. Its first step will search the dictionary D to find that entry X belonging to $AUX(\text{root})$ which has the smallest $X \cdot 1$ value satisfying $X \cdot 1 \geq c$. The second step will walk the subtree inside the base-tree B whose nodes are either critical with respect to the condition $a_2 < \text{KEY} \cdot 2 < b_2$ or are the ancestors of critical nodes. The third step will find the isomorphic image of this subtree lying inside the entry-tree $t(X)$, using the down-pointers to locate each node that it traverses. Define an entry Y in $t(X)$ to be "bottom-level" relative to (c, a_2, b_2) if the third step of our search visited Y but none of its descendants. The final step of our search will make a list of all visited bottom nodes. It is easy to see that this list corresponds to $CES(c, a_2, b_2)$ and that each of the four steps run in time $O(\log N)$, thereby showing $CES(c, a_2, b_2)$ can be retrieved in this time. Q.E.D.

The remainder of this section will explain the usefulness of the critical entry set in performing two-dimensional orthogonal range queries. For any entry X belonging to the critical entry set $CES(a_1, a_2, b_2)$, let $\text{COUNT}(X, a_1, b_1)$ denote the number of entries which satisfy the range condition $a_1 < \text{KEY} \cdot 1 < b_1$ and lie in the same auxiliary field as X . Note that these elements lie in consecutive positions to the right of X in their common auxiliary field. Therefore, the time needed to copy these elements into the user's workspace is $O(\text{COUNT}(X, a_1, b_1))$ when one begins with a pointer to the

entry X . This observation suggests the following algorithm for performing an orthogonal range query of the type $\text{SET}(q)$ in a k -fold tree of the form $T_e(2)$:

(1) Apply the algorithms from Theorem 1 to construct the critical entry set $\text{CES}(a_1, a_2, b_2)$.

(2) For each X belonging to the set $\text{CES}(a_1, a_2, b_2)$, perform a walk that begins at X and proceeds to the right through the auxiliary field containing X until reaching the first element Y with $Y \cdot 1 \geq b_1$. Print a list of all the elements visited during this walk except for Y .

THEOREM 2. *The procedure above will correctly find all the elements of $\text{SET}(q)$ in worst-case locate time $O(\log N)$ when it searches a 2-fold tree $T_e(2)$. This k -fold tree will occupy $O(N \log N)$ space.*

Proof. The memory space $O(N \log N)$ of $T_e(2)$ follows because a 2-fold tree of height $O(\log N)$ will have no more than $O(\log N)$ entries per record. The correctness of the search above follows because each record belonging to $\text{SET}(q)$ will lie in precisely one of the auxiliary fields visited. This algorithm has a locate-and-copy time $O(\log N + \text{COUNT}(q))$ because Theorem 1 implies that the first step consumes time $O(\log N)$ and because step 2 consumes time $\sum_{X \in \text{CES}(a_1, a_2, b_2)} \text{COUNT}(X, a_1, b_1) = \text{COUNT}(q)$. Therefore, the locate-component of its time complexity is $O(\log N)$. Q.E.D.

Now, we will define a new data structure $T_a(2)$ that will produce the analogues of Theorem 2 for aggregate queries of the type $\text{SUM}(q)$ and $\text{COUNT}(q)$. This data structure will be defined to be the same as $T_e(2)$ except that its entries will contain two additional fields about aggregates. These will store in each entry X a quantity $\text{COUNT}^*(X)$, which indicates how many records lie to the left of X in its auxiliary field, and a second quantity $\text{SUM}^*(X)$, indicating the sum of some specially designated value stored in these entries. Our upper bound $O(\log N)$ is consistent with Fredman's lower bound $\Omega(\log^2 N)$ because the former is over static groups while the latter concerns aggregates over dynamic semigroups.

THEOREM 3. *The data structure $T_a(2)$ makes it possible to calculate the aggregates $\text{SUM}(q)$ and $\text{COUNT}(q)$ in worst-case time $O(\log N)$, for any two-dimensional orthogonal range query.*

Proof. Since the algorithms for evaluating $\text{COUNT}(q)$ and $\text{SUM}(q)$ are quite similar, we will present only the former. For simplicity, we restrict our attention to an orthogonal range query of the precise form: $q = \{a_1 < \text{KEY} \cdot 1 \leq b_1 \text{ and } a_2 < \text{KEY} \cdot 2 < b_2\}$. Note that the subset of S satisfying q has cardinality equal to:

$$(2) \quad \sum_{X \in \text{CES}(b_1, a_2, b_2)} \text{COUNT}^*(X) - \sum_{Y \in \text{CES}(a_1, a_2, b_2)} \text{COUNT}^*(Y).$$

Since Lemma 1 indicates that any critical entry set can be walked in time $O(\log N)$, it follows that this time is sufficient to visit the sets $\text{CES}(b_1, a_2, b_2)$ and $\text{CES}(a_1, a_2, b_2)$, and to gather the information necessary to evaluate the expression above. Q.E.D.

Remark 1. Theorems 2 and 3 are significant because they offer a better combination of time and space than their predecessors. Thus, the first-generation trees $T_1(2)$ will have a $\log N$ more expensive time with no compensating advantages in memory space, and the two-dimensional version of overlapping k -ranges [BeMa80] will do no better than matching the time of $T_e(2)$ while requiring substantially more space. We should also point out that while the trees T_a and T_e are unambiguous improvements over overlapping k -ranges for the case of dimension $k=2$, comparisons for higher dimensions are more difficult because overlapping k -ranges have the better time

complexities, and T_a and T_e have the better space complexity for these higher dimensions (as we will discuss in § 8).

4. Review of the Willard–Lueker dynamic transformation. Sections 5 and 6 of this article discuss several dynamic data structures which each support the same expected complexity $O(\log N)$ simultaneously for insertion, deletion, and two-dimensional locate-retrievals; these data structures will differ only with respect to worst-case complexity. Some parts of our worst-case analysis will rely on dynamic transformation techniques developed independently by Lueker and Willard [Lu78], [Lu79], [Wi78a], [Wi79], [LuWi82] and [WiLu84]. In this section, we will offer a brief review.

The notation used here will be that of Willard and Lueker [WiLu84]. Let n denote the length of a sequence of insertion and deletion commands, which manipulate a set that is initially empty and whose size never exceeds N . Also, let $T_{n,N}$ denote the largest amount of time needed by a particular algorithm to execute such a sequence of commands. Then the *worst-case sequence-average* complexity for this insertion-deletion algorithm will be defined as $\text{MAX}(T_{n,N}/n | n, N > 0)$. The cited papers on dynamic augmented trees studied essentially how to manipulate these trees to assure that their base sections retain a height $O(\log N)$. Our general approach to retaining height $O(\log N)$ was similar to the conventional AVL and bounded balance methods [Kn73], [NiRe73] insofar as it also relied on the single and double rotations to maintain a good balance for the trees. The differences between our approach and those earlier articles arose because they considered an environment where a rotation had a cost $O(1)$ while we faced a much more stringent setting where the auxiliary fields of the involved nodes require costly readjustment after each rotation. [Wi78a], [Wi79] and [WiLu84] proved that the AVL method [Kn73], the 2-3 tree method [AhHoUI74], the B -tree [Kn73] method, and the usual bounded balance method [NiRe73] are inefficient when applied to augmented trees, with respect to both the worst-case cost criterion and the worst-case sequence-average cost criterion. All six of the papers on dynamic transformations showed that these costs can be reduced significantly with any one of several closely related types of modified bounded balance algorithms.

More precisely, let $\pi(N)$ designate a positive-valued monotonically increasing function of N , and N_v the number of leaves descending from the internal node v in the base-tree B . We will call N_v the *rank* of the node v . Assume that no more than time $O(\pi(N_v) \cdot N_v)$ is required to adjust the involved auxiliary fields when one of the single or double rotations from Fig. 3 is applied to the node v . Then the weak theorems from [Lu78], [Lu79], [LuWi82], [Wi78a], [Wi79] and [WiLu84] state that augmented trees have a worst-case sequence-average insertion-deletion complexity $O(\pi(N) \log(N))$; and the strong theorems [Lu79], [Wi78a], [Wi79] and [WiLu84] indicate that several essentially equivalent data structures support the same time complexity in the strictly worst case.

We will only outline the ideas behind the proof of the weak theorem in this paper; this discussion will provide the reader with sufficient intuition to understand how these methods may be usefully applied to the new data structures introduced in this paper.

Let v_l denote the left son of internal node v and $p(v)$ the ratio N_{v_l}/N_v (following the notation of Nievergelt and Reingold [NiRe73]). For any fixed positive constant $\alpha \leq 1/2$ a binary tree is said to satisfy the “bounded balance” condition $BB(\alpha)$ iff its every internal node satisfies the condition $\alpha \leq p(v) \leq 1 - \alpha$. Nievergelt and Reingold showed that these trees have height $O(\log N)$ and support worst-case insertion-deletion complexities $O(\log N)$ when one chooses a constant $\alpha < 1 - \sqrt{2}/2$. The algorithms in [Lu79], [Wi78a], [Wi79] and [WiLu84] are, for the most part, natural generalizations

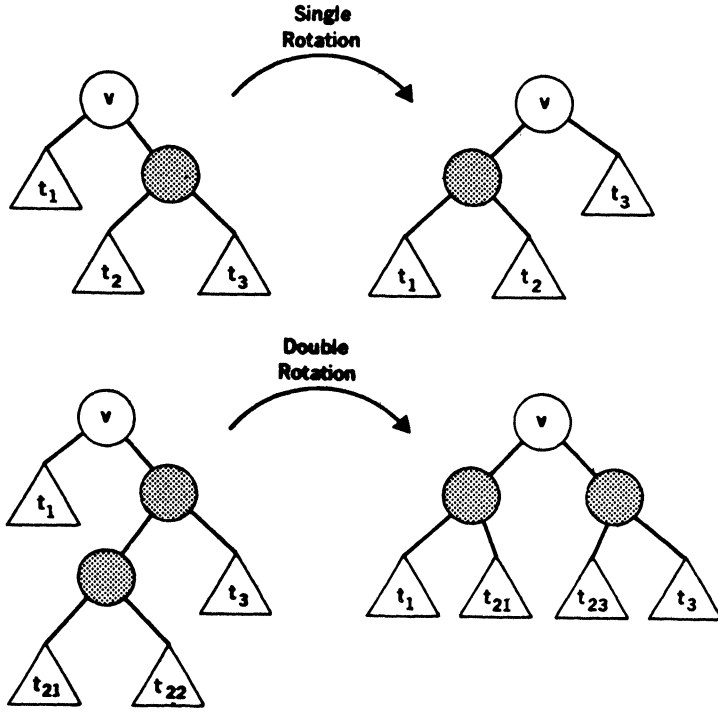


FIG. 2. Rebalance operations for trees of bounded balance.

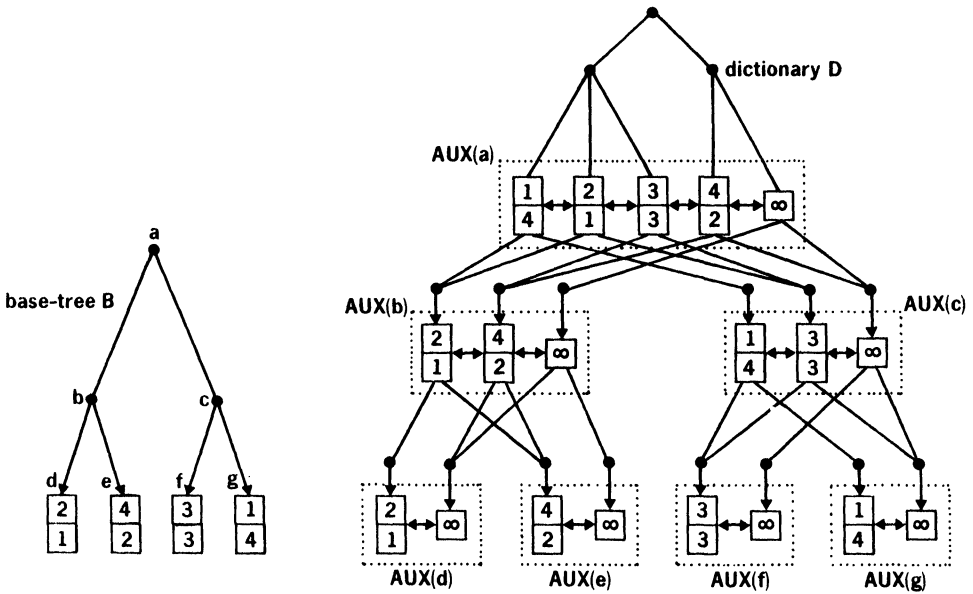


FIG. 3. A tree $T_d(2)$ which represents the same set $\{(1, 4), (2, 1), (3, 3), (4, 2)\}$ as its analogue $T_e(2)$ in Fig. 1.

of the Nievergelt–Reingold method for augmented trees; one significant difference between the two methods is that the timing of rotations under the Willard–Lueker approach is different for reasons of efficiency. Given any constant $\alpha \leq 1/6$, one example of our method would consist of an algorithm that first inserts or deletes the relevant leaf-record (in both the base and auxiliary fields), next checks the ancestors of this leaf to determine which one of them now temporarily violate the condition $BB(\alpha)$ as a result of the first step, and finally rebalances these nodes in a bottom-up order using the following rules:

I) In the essentially uninteresting special case where $N_v \leq 1/\alpha^2$, use a brute-force procedure to build an essentially perfectly balanced sub-tree descending from v where the numbers of leaves descending from any two brothers differ by no more than one.

II) If $N_v > 1/\alpha^2$ and $p(v) < \alpha$ then apply the single rotation of Fig. 2 when at least $2\alpha N_v$ leaves belong to the subtree t_3 of that figure; otherwise apply its double rotation. Apply the mirror image of these rotation rules in the alternate case where $N_v > 1/\alpha^2$ and $p(v) > 1 - \alpha$. In both cases, adjust the auxiliary fields to reflect the changes.

This algorithm is a good example of the general methods from [Lu78], [Lu79], [LuWi81], [Wi78a], [Wi79] and [WiLu81] because its correctness and efficiency are very easy to verify. First note that the bounded balance condition $BB(\alpha)$ is clearly maintained after each insertion and deletion operation, by inspection¹ of rebalance rules I and II and the fact that $\alpha \leq 1/6$. Next note that since rebalance step I is applied only to a node whose rank N_v is less than a constant, the cost of this step has an inconsequential magnitude. Also inconsequential are all the other costs of this algorithm except for reconfiguring the auxiliary field after a rotation by step II. Because this last step is potentially expensive, we will examine it very carefully.

Define the *total rank* of a bounded balance tree to be equal to the sum of the ranks of all its internal nodes. We will use an accounting argument, similar to the paradigm of Willard and Lueker (references just cited) based on the following two lemmas.

LEMMA 1. *No individual insertion can change the total rank by more than $O(\log N)$ before the rebalancing. The sum of all the increases during n operations is therefore bounded by $O(n \log N)$.*

Proof. Easy: the first statement follows because the trees have height $O(\log N)$; and the second is an immediate consequence of the first. Q.E.D.

LEMMA 2. *Let w denote the amount of time some invocation of step ii) needs to adjust the auxiliary fields. This step will cause the total rank to decrease by at least $\Omega[w/\pi(N)]$.*

Proof. Using reasoning similar to [WiLu84] in the context of $\alpha < 1/6$, one may verify that: i) an invocation of step II must produce a decrease in total rank of at least αN_v ; and ii) the time of w of the same invocation is bounded above by $O(\pi(N_v) \cdot N_v)$, by the definitions of w and π . These observations imply that the ratio of work to decrease in rank must satisfy Lemma 2, with the coefficient inside the Ω -notation depending on the constant α . Q.E.D.

Now we will sketch how results similar to Lemmas 1 and 2 enable Willard and Lueker to prove their first theorem. Note that since the total rank is initially zero and cannot go negative, Lemma 2 implies that the aggregate time for all the adjustments in auxiliary fields is bounded by the product of $\pi(N)$ with the sum of all the increases in total rank. By Lemma 1, the aggregate cost of these adjustments is

¹ This inspection follows by reasoning similar to that in [NiRe73].

$\leq O(n \cdot \pi(N) \log N)$. [WiLu84] observes that all other steps of their algorithm satisfy this cost-bound by a trivial argument, thereby verifying the next theorem.

THEOREM 4 [Lu79], [Wi78a], [Wi79], [WiLu84]. *It is possible to devise insertion-deletion algorithms for any augmented tree with worst-case sequence-average complexities $O(\pi(N) \cdot \log N)$.*

See [WiLu84] for a stronger version of Theorem 4 that also guarantees strict worst-case costs. Most of the rest of this paper will explain how to apply Theorem 4 to obtain new complexities with the special data structures introduced in this article. Although it is not chiefly germane, it should be pointed out that the proof of sequence-average complexity in [Lu78], [Lu79], [LuWi82], [Wi78a], [Wi79] and [WiLu84] is stronger than the results above because the former also applied to values of α greater than $1/6$. These refinements will improve substantially the coefficient associated with the height $O(\log N)$ of $BB(\alpha)$ trees. [Lu78] and [LuWi82] illustrated a noteworthy special case of Theorem 4, whose advantage is that it is applicable for especially large α , and whose disadvantage is that it does not generalize to all augmented trees.

5. Two-dimensional dynamic data structures with either downpointers or down-trees. During our discussion of probability models and expected complexities, $\{\gamma_i\}$ will denote a sequence of n distinct integer-values, chosen from $\{-1, 0, 1\}$ subject to the constraint $\sum_{i=1}^m \gamma_i \geq 1$ for every positive $m \leq n$. We will say $\{\gamma_i\}$ is a *random control sequence* over a set S of ordered pairs in the xy -plane iff the set S is initially empty and the i th command manipulating this set consists of:

- a) inserting a new record when $\gamma_i = 1$ (the ordered pair inserted in S will be drawn from the uniform distribution over the unit square);
- b) deleting one of the elements from S when $\gamma_i = -1$ (where the record chosen for deletion is selected at random); and
- c) processing the orthogonal range query $a_1 \leq \text{KEY} \cdot 1 \leq b_1 \wedge a_2 \leq \text{KEY} \cdot 2 \leq b_2$ when $\gamma_i = 0$ (where the values a_i and b_i are chosen by the uniform probability distribution subject to the constraints $0 \leq a_i \leq b_i \leq 1$).

The particular goal of this section is to develop two algorithms which satisfy some type of constraint on worst-case complexity, and which assure that every insertion, deletion, and two-dimensional locate-retrieval command during any random control sequence has expected complexity $O(\log N)$. These algorithms will differ primarily according to the type of worst-case constraint they satisfy.

In our discussion, X will again denote an entry in the field $AUX(v)$ of a 2-fold tree and f the father of the node v from the tree's base. Also, $\text{DOWNSET}(X)$ will denote the set of entries Y in $AUX(f)$ such that X is one of Y 's down-sons. Several of our algorithm descriptions will employ steps whose time-complexity is either proportional to the cardinality of some downset, or at least to the logarithm of this cardinality. The efficiency of these algorithms will follow from the observation that the involved downsets, under a random control sequence, will have expected sizes lying in $O(1)$.

THEOREM 5. *The 2-fold tree data structure $T_e(2)$ (defined in § 3) makes it possible to achieve simultaneously worst-case locate-time $O(\log N)$ and expected complexity $O(\log N)$ for each insertion and deletion command in any random control sequence.*

Proof. Note that Theorem 2 states that the data structure $T_e(2)$ supports worst-case locate-time $O(\log N)$. Therefore, only the second half of Theorem 5, involving the complexity of insertions and deletions, remains to be proven.

Let α designate any positive constant $\leq 1/6$. Recall that the definition of second-generation trees $T_e(2)$ requires that their base sections have heights lying in $O(\log N)$. We will guarantee this height by assuring that the balance condition $BB(\alpha)$ is

maintained after each insertion and deletion operation. Our algorithm for insertions consists of the following seven-step procedure:

- 1) Find the location where the new record should be inserted in the base-tree and insert it in the obvious manner.² This record will be denoted by R .
- 2) Search the dictionary D to find the entry Z in $AUX(\text{root})$ with the smallest $Z \cdot 1$ value satisfying $Z \cdot 1 \cong R \cdot 1$.
- 3) Let s denote the set of ancestors of the leaf-record R in the base-tree B , and s' the isomorphic image of s in the entry-tree $t(Z)$. Find all the entries in s' by using the subtree s and the downpointers in the obvious manner.
- 4) For each entry X in s' , insert an entry for the record R to X 's immediate left in its auxiliary field. Make the downpointers of this new entry initially point to the same two addresses as the entry X .
- 5) Now take each entry X in s' and check all the entries Y in $\text{DOWNSET}(X)$ for whether they satisfy $Y \cdot 1 \cong R \cdot 1$. Those which do should have their downpointers changed so that they point to R 's new entry rather than to X .
- 6) If step 1 caused the base-tree B to violate the bounded balance condition $BB(\alpha)$, then correct all the unbalanced nodes by using the Willard-Lueker algorithm (outlined in the last section) to rebalance this tree and adjust the auxiliary fields accordingly.
- 7) Insert a representative of the record R in the dictionary D by using any balanced tree method running in time $O(\log N)$.

Now we will examine the runtime of this procedure. Since the base-tree B and the dictionary D have heights in $O(\log N)$, steps 1 through 4 and 7 clearly must run in a worst-case time $O(\log N)$. Also, it is easy to see that whenever step 6 applies a rotation to a node v , the time needed to adjust its auxiliary fields is $O(N_v)$; this observation, in conjunction with Theorem 4, implies that step 6 has a worst-case sequence-average complexity $O(\log N)$. Thus, except for step 5, all the components of our insertion algorithm have a complexity $O(\log N)$ either by inspection of the procedure or from previous reasoning.

The time spent by step 5 on each entry X is easily seen to be proportional to the size of $\text{DOWNSET}(X)$. Appendix A outlines a tedious but otherwise straightforward proof that the downsets processed by step 5 always have expected cardinalities $O(1)$ during random control sequences. Since step 5 manipulates $O(\log N)$ distinct downsets, this result implies that it runs in expected time $O(\log N)$. Hence our full seven-step insertion algorithm has this asymptotic time; and by similar reasoning, deletions have an expected complexity $O(\log N)$ under the natural inverse of the procedure above. Q.E.D.

Note that the procedure outlined in step 5 had a very inefficient worst-case complexity lying in $\Omega(N)$. This problem extends to worst-case sequence-average complexity, which also lies in $\Omega(N)$. The remainder of this section will explain how to reduce the worst-case cost of step 5 to $O(\log^2 N)$, thereby producing insertion-deletion algorithms with expected complexities $O(\log N)$ and worst-case sequence-average complexities $O(\log^2 N)$. This method will have one disadvantage: It will increase worst-case locate-retrieval complexity to $O(\log^2 N)$. However, since expected locate-complexity will remain $O(\log N)$, this rise in worst-case costs will often be acceptable.

² Since records are stored at the leaf level in base trees, our algorithm for inserting a new record R will also create a new internal node whose sons are R and one of its brothers. Also, to remain consistent with the other steps, step 1 will initialize the auxiliary field of this new node so that it describes only R 's brother at the end of this step.

This alternate method will rely on a further modified 2-fold tree data structure, called $T_d(2)$. This data structure will be defined to be the same as $T_e(2)$ except that entries in auxiliary fields will no longer contain pointers to the addresses of their two down-sons. Instead, all entries X belonging to the same downset will form the leaves of a special 2-3 tree whose root points to their common down-son and each of whose other nodes contains a pointer to its father; this 2-3 tree will be called a *downtree*. An example of a data structure $T_d(2)$ is illustrated in Fig. 3; this data structure differs from the tree $T_e(2)$, illustrated in Fig. 1, only by having downpointers replaced by downtrees.

The algorithms for performing insertions, deletions, and retrievals under the data structure $T_d(2)$ are essentially the natural modifications, required by the concept of downtree, of the algorithm for the tree $T_e(2)$. That is, the insertion, deletion, and retrieval algorithms for the tree $T_d(2)$ will differ from their counterparts under $T_e(2)$ only in the following two respects:

i) On each occasion when an algorithm for $T_e(2)$ would use a downpointer to find a down-son, the counterpart under $T_d(2)$ will begin at the same entry X , find the root of its analogous downtree by traversing the bottom-up path of its ancestors, and then advance to this down-son by using the pointer stored in the root.

ii) Step 5 of the procedure for insertions into $T_d(2)$ trees will differ from its counterpart for $T_e(2)$ by consisting of an operation that simply splits X 's downtree so that all the leaves Y satisfying $Y \cdot 1 \leq R \cdot 1$ form the basis of a new downtree pointing to R 's recently added entry; similarly, the counterpart of step 5 for deletions will consist of an operation merging two downtrees. The general algorithms from [AhHoU174] will be used to perform these split and merge operations on downtrees. Note that [AhHoU174] has shown that all 2-3 trees have logarithmic height and permit the operations of split and merge to run in logarithmic time. Now, the worst-case number of leaves in any downtree is clearly $O(N)$; furthermore, the downsets (and therefore also downtrees) encountered by our insertion, deletion, and retrieval algorithms will have expected sizes $O(1)$ under any random control sequences, by Appendix A. It therefore follows that operations i) and ii) above will both have expected time-complexities $O(1)$ and worst-case times $O(\log N)$.

Using the observations from the paragraph above to modify the proofs of Theorems 1, 2, and 5, we see that the data structure $T_d(2)$ will have the same expected complexities as $T_e(2)$, a worst-case locate complexity which is less efficient than $T_e(2)$ by a factor of $\log N$, and a better worst-case sequence-average insertion-deletion complexity, which is bounded by $O(\log^2 N)$. All these changes are due to the use of downtrees. In a whole, the data structure $T_d(2)$ is probably slightly more useful than $T_e(2)$. In a dynamic environment, the following theorem formally summarizes its properties.

THEOREM 6. *The 2-fold-tree data structure $T_d(2)$ makes possible achieving simultaneously:*

- 1) *an expected complexity $O(\log N)$ for insertions, deletions, and locate-retrievals under the probability model of a random control sequence, and*
- 2) *a worst-case locate-retrieval complexity $O(\log^2 N)$ and a worst-case sequence-average insertion-deletion complexity $O(\log^2 N)$.*

Remark 2. One can apply dynamic methods similar to [Lu79], [Wi78a], [Wi79] and [WiLu84] to convert Theorem 6's $O(\log^2 N)$ sequence-average update complexity into a strict worst-case result. Although all the expected time complexities in this section and the accompanying Appendix A were predicated on a uniform distribution generating the records to be inserted, they can be generalized to any distribution where all permutations of input occur with equal probability, as well as many measurable

nonuniform distributions where the probability density respects well defined upper and lower bounds. Appendix B illustrates another new data structure, called $T_f(2)$, which is a modification of $T_e(2)$ for applications which have insertions but no deletions. This data structure satisfies all the constraints of Theorem 5 and additionally guarantees a worst-case sequence-average complexity $O(\log^2 N)$ for insert-operations. $T_f(2)$ is an extremely useful data structure when deletions are either absent or infrequent.

6. Backward indexing and its implications for multidimensional retrieval. Let S denote a time-varying set of records whose cardinality never exceeds N , and $\text{KEY}(R)$ the key of a typical member of this set. A *backward index* governed by the constant λ will be defined as a data structure that enables one to calculate a nonnegative integer $\Phi(R)$, in worst-case time $O(\log N)$ for any record R in S , satisfying the constraints:

- i) $\Phi(R) \leq N^\lambda$,
- ii) if $\text{KEY}(R_1) < \text{KEY}(R_2)$ then $\Phi(R_1) < \Phi(R_2)$.

Suppose the cost to change the backward index value Φ of j records from the set S is bounded by $\pi(N) \cdot j$, for some well-defined function π which monotonically increases with the size N of the set S . Then Dietz [Di84] has shown how to build a data structure which guarantees worst-case sequence-average complexity $O[\pi(N) + \log N]$ for any sequence of insert-delete operations. (See also [Di82] for a slightly weaker version of this result.)

This section will explain how the combination of Dietz's theorem, the bounded balance method of [WiLu84] and Willard's q -fast tries [Wi81], [Wi84c] lead to an alternate structure whose worst-case insertion-deletion complexity is better than $T_d(2)$, $T_e(2)$ and $T_f(2)$ but whose retrieval time is worse. The q -fast tries of [Wi81], [Wi84c] occupy memory $O(N)$ and have worst-case time $O(\sqrt{\log M})$ for insertion, deletion, and (one-dimensional) locate-retrieval, when all the keys are nonnegative integers less than some upper bound M . Consider now a two-dimensional set S which also contains nonnegative integer keys less than some upper bound M in their first components. Then if every auxiliary field of a 2-fold tree contains a q -fast trie, the resulting data structure will provide a combination of space $O(N \log N)$ and of time-measurement $O(\sqrt{\log M \log N})$ for both worst-case locate-retrieval and for worst-case sequence-average insertion-deletion complexity, by an easy application of Theorem 4 and the general methods from § 2. This data structure will be denoted as $T_c^M(2)$.

Now, we will explain how to combine the concepts of backward indexing and the trees described above to produce a new data structure which generalizes the result above even when the keys are real numbers. Consider a two-part data structure:

- i) whose first section is backward index Φ on the $\text{KEY} \cdot 1$ values of set S which is governed by some constant λ ;
- ii) and whose second section is a data structure of type $T_c^M(2)$, where the first component of each record is $\Phi(\text{KEY} \cdot 1)$ rather than the usual $\text{KEY} \cdot 1$ and where the constant M is chosen to equal N^λ .

This data structure will be denoted $T_b(2)$. For any real number x , define $\Phi(x)$ as $[\Phi(R^+) + \Phi(R^-)]/2$, where R^+ and R^- are the members of the set S with the least and greatest $\text{KEY} \cdot 1$ values satisfying respectively $R^+ \cdot 1 \geq x$ and $R^- \cdot 1 \leq x$. Then the records satisfying

$$(3) \quad a_1 < \text{KEY} \cdot 1 < b_1 \wedge a_2 < \text{KEY} \cdot 2 < b_2$$

also obviously satisfy

$$(4) \quad \Phi(a_1) < \Phi(R) < \Phi(b_1) \wedge a_2 < \text{KEY} \cdot 2 < b_2.$$

Therefore, one algorithm for performing the query (3) in $T_b(2)$ consists of a two-part procedure which first uses the backward index to calculate the specific function-values $\Phi(a_i)$ and $\Phi(b_i)$ and then searches the second part of $T_b(2)$ with the modified query (4). This retrieval algorithm will clearly run in time $O(\sqrt{\log M} \log N)$; since $M = N^\lambda$, this locate-time reduces to $O(\log^{3/2} N)$. Also, it is easy to see that time $O(\log^{3/2} N)$ is sufficient to adjust all the parts of the data structure $T_b(2)$ when the backward function value $\Phi(R)$ of any record R changes; this observation implies that $T_b(2)$ has a worst-case sequence-average insertion-deletion complexity $O(\log^{3/2} N)$, by Dietz's theorem and the complexity of $T_c^M(2)$. If the data structure $T_b(2)$ is slightly modified so that the entry for every record R in the backward index contains a pointer to all R 's other entries in $T_b(2)$, then the *expected* cost of insertions and deletions can be reduced to $O(\log N)$ under the probability model of random control sequences.

Remark 4. Although the backward indexing k -fold tree $T_b(2)$ is very different from the downpointer variants of this concept, $T_a(2)$, $T_d(2)$, and $T_e(2)$, there is nevertheless one common intuition which motivated all these approaches. This is that the large number of auxiliary fields needing inspection in 2-fold trees makes it cost-effective to develop certain special data structures which reduce the costs of these searches. No analogues of this phenomenon arise in dimension one because the overhead of these new components is typically larger than the time $\log N$ needed to perform the straightforward search. Research on queries of higher dimensions is interesting largely because a great number of new algorithmic techniques are then cost-effective.

Remark 5. It is feasible to combine the data structures $T_b(2)$ and $T_d(2)$ into a hybrid data structure, $T_h(2)$, which essentially consists of the union of their individual components. The advantage of this hybrid is that it combines the best asymptotic aspects of the retrieval times of $T_b(2)$ and $T_d(2)$. It would therefore have expected complexity $O(\log N)$ and worst-case time $O(\log^{3/2} N)$ for locate-retrieval. However, the worst-case insertion-deletion complexity of this hybrid is $O(\log^2 N)$, which is less efficient asymptotically than $T_b(2)$. The coefficients of its memory space and insertion-deletion complexities would also be larger than those of both $T_b(2)$ and $T_d(2)$. This hybrid therefore has both advantages and disadvantages. Its expected insertion-deletion complexity is $O(\log N)$, similar to both $T_b(2)$ and $T_d(2)$ with a slightly larger coefficient.

7. Further results. Recall that § 3 indicated we would assume until further notice that no two elements of the set S have the same values stored in either their fields $\text{KEY} \cdot 1$ or $\text{KEY} \cdot 2$. We now show our retrieval times will also hold without this constraint. Our generalization is proven in a context sufficiently broad to apply to literally any data structure supporting orthogonal range queries.

THEOREM 7. *Suppose S^* is a set of k -tuples where no two elements have the same value stored in any component $\text{KEY} \cdot i$ and that the data structure D guarantees a worst-case retrieval time $O(f)$ for orthogonal queries on this set. Then there must exist data structures D' with the same worst-case complexity for orthogonal range queries on any arbitrary set S .*

Proof. Let (u_1, u_2) and (v_1, v_2) denote two ordered pairs. Define $(u_1, u_2) < (v_1, v_2)$ iff either

- i) $u_1 < v_1$ or
- ii) $u_1 = v_1$ and $u_2 < v_2$.

As usual we assume the set S is initially empty, and each record R is inserted at some unique time $T(R)$. Let $\text{KEY}^* \cdot i$ denote the ordered pair $(\text{KEY} \cdot i, T(R))$. Also let a^- and a^+ be abbreviations for the respective ordered pairs $(a, -\infty)$ and $(a, +\infty)$. Then

by definition the same set of records will satisfy (5) and (6):

$$(5) \quad a_1 < \text{KEY}_1 < b_1 \wedge a_2 < \text{KEY} \cdot 2 < b_2 \wedge \dots \wedge a_k < \text{KEY} \cdot k < b_k,$$

$$(6) \quad a_1^- < \text{KEY}^* \cdot < b_1^+ \wedge a_2^- < \text{KEY}^* \cdot 2 < b_2^+ \wedge \dots \wedge a_k^- < \text{KEY}^* \cdot k < b_k^+.$$

As each record R contains a unique value $T(R)$, no two elements of S contain the same values in $\text{KEY}^* \cdot i$; the hypothesis of Theorem 7 thus implies that some data structure, call it D' , will answer any query of the form (6) in time $O(f)$. As equations (5) and (6) are equivalent, we can answer (5) in a time exceeding $O(f)$ by an inconsequential additive constant simply by translating (5) into an equivalent representation in (6) and using the latter to search D' . Q.E.D.

Remark 6. Although the proof of Theorem 7 is trivial, the proposition is worth remembering because its analogue does not hold for partial match queries. The worst-case retrieval time for the latter operations changes dramatically for some data structures if the elements of S are permitted to contain repeating values in some of their fields. (k - d trees [Be75] are an example of a data structure whose worst-case partial match time changes with this assumption.) (I thank the referee for noting the distinction between partial match and range queries, and for suggesting it would therefore be useful to prove Theorem 7.) This theorem also holds in a strengthened version without the constructs of $T(R)$ and $\text{KEY}^* \cdot i$, but the latter are greatly useful in shortening the proof.

In one type of application, it may be desirable to apply the binary static-to-dynamic transformation [BeSa80], [OvLe81] to k -fold trees. Given any initial (usually static) data structure which can be built in time $P(N)$ and which has a retrieval time $R(N)$, this transformation will produce a dynamic data structure with retrieval time $O(R(N) \cdot \log N)$ and insertion-deletion complexity $O[P(N)(\log N)/N]$.

The static-to-dynamic transformation has many uses in multidimensional retrieval; for instance, Willard [Wi78c], [Wi80] independently developed a special version of this concept for k - d trees and proved it has the best possible retrieval time on k - d trees. In the context of k -fold trees, the static-to-dynamic transformation will usually be undesirable because its retrieval time is typically more expensive than the transformation of [Lu78], [Lu79], [LuWi82], [Wi78a], [Wi79] and [WiLu84] by an extra factor of $\log N$; however, an important exception to this rule is the 2-fold-tree data structure $T_a(2)$. This data structure is too rigid³ for any other dynamic method to manipulate it. When the binary-static-to-dynamic transformation is applied to $T_a(2)$, a new data structure results with complexities $O(\log^2 N)$ for insertion, deletion, and two-dimensional aggregate retrieval over groups.

We will denote this data structure by $T'_a(2)$. Note that its complexity for the dynamic aggregate-retrieval problem is similar to the results for this particular problem which [Lu78], [Lu79], [LuWi82], [Wi78a], [Wi79] and [WiLu84] obtained with an entirely different method. One difference is that the data structure $T'_a(2)$ does not support aggregates under *semi-group* operators. However, a dynamic aggregate-retrieval complexity $O(\log^2 N)$ on *group* operators follows by applying either the static-to-dynamic transformation to $T_a(2)$ or the bounded balance method to $T_1(2)$. These upper bounds on complexity match Fredman's lower bound [Fr81a] for the semi-group version of the dynamic orthogonal range query problem. Aggregate orthogonal range queries in a dynamic environment are thus more expensive than locate-retrievals by at least the semi-group measurement of complexity.

³ The cost of inserting or deleting even a single record in $T_a(2)$ is prohibitive because a large number of aggregate fields, $\text{SUM}^*(X)$ and $\text{COUNT}^*(X)$, will then need updating.

8. Generalizations. As we mentioned in § 4, Willard and Lueker [WiLu84] developed a more elaborate version of Theorem 4 which guarantees the worst-case time of individual commands as well as sequences. Using this result in the place of Theorem 4, the update complexities of $T_b(2)$, $T_c^M(2)$ and $T_h(2)$ become results which are worst-case estimates over both individual and sequences of commands.

One can also convert the sequence-average update upper bound $O(\log^2 N)$ of $T_d(2)$ into a worst-case quantity. In this case, the precise methods of [WiLu84] are inapplicable, but one can readily develop a suitable modification.

The generalizations of the preceding results to dimensions $k \geq 3$ follow from our two-dimensional results and the methods of [Be80], [Lu79], [Wi78a], [Wi79] and [WiLu84]. In particular, [Be80] showed that given any k -dimensional data structure, it is possible to develop a $(k+1)$ -dimensional data structure with factor $\log N$ greater retrieval time and memory space; and [Lu79], [Wi78a], [Wi79] and [WiLu84] illustrated another version of this same transformation which also guarantees that insertion-deletion complexity will increase by no more than $\log N$. The final results for the eight different versions of k -fold trees are illustrated in Table 1. Each data structure in this table differs by at least some measure of complexity.

TABLE 1
Various complexities of different types of k -fold trees for dimension $k \geq 2$.

| | Expected locate-retrieval | Worst-case locate-retrieval | Expected insertion-deletion | Worst-case insertion-deletion | Aggregate- retrieval in all cases |
|------------|-------------------------------|--------------------------------|--------------------------------|----------------------------------|--|
| $T_1(k)$ | $\log^k N$ | $\log^k N$ | $\log^k N$ | $\log^k N$ | $\log^k N^\dagger$ |
| $T_a(k)$ | $\log^{k-1} N$ | $\log^{k-1} N$ | * | * | $\log^{k-1} N$ |
| $T'_a(k)$ | $\log^k N$ | $\log^k N$ | $\log^k N$ | $\log^k N$ | $\log^k N$ |
| $T_b(k)$ | $\log^{k-1/2} N$ | $\log^{k-1/2} N$ | $\log^{k-1} N$ | $\log^{k-1/2} N$ | * |
| $T_c^M(k)$ | $(\log^{k-1} N)\sqrt{\log M}$ | $(\log^{k-1} N)\sqrt{\log M}$ | $(\log^{k-1} N)\sqrt{\log M}$ | $(\log^{k-1} N)\sqrt{\log M}$ | * |
| $T_d(k)$ | $\log^{k-1} N$ | $\log^k N$ | $\log^{k-1} N$ | $\log^k N$ | * |
| $T_e(k)$ | $\log^{k-1} N$ | $\log^{k-1} N$ | $\log^{k-1} N$ | § or * | * |
| $T_h(k)$ | $\log^{k-1} N$ | $\log^{k-1/2} N$ | $\log^{k-1} N$ | $\log^k N$ | * |

* The cited complexity, not optimized under this k -fold tree, is typically $\Omega(N)$ or worse.

† This data structure differs from all the others by supporting the added capability to calculate aggregates over semi-group operators.

§ Normally this complexity is unoptimized, but it respects a sequence-average bound $O(\log^k N)$ when only insertions are present (see Appendix B).

The subscripts a through h in the names of the eight data structures were chosen as mnemonic devices. Thus “ a ” stands for supports aggregate-retrievals (with superscript “ t ” indicating the presence of the static-to-dynamic transformation), “ b ” for backward indexing, “ c ” for integer key-space bounded by the constant M (appearing in superscript), “ d ” for the most efficient dynamic data structure, “ e ” for optimizes only expected insertion-deletion complexity, and “ h ” for hybrid. First-generation k -fold trees were naturally given the subscript “1”.

All the data structures in Table 1 occupy memory space $O(N \log^{k-1} M)$. The data structure $T_d(k)$ is clearly less desirable than $T_h(k)$ asymptotically. No other data structure is asymptotically worse than another by all measures of comparison. Even the comparison between $T_d(k)$ and $T_h(k)$ is difficult because the former has a much better coefficient associated with its memory space and insertion-deletion complexities.

Since the time [Wi78b] proposed the seven data structures in Table 1, McCreight [Mc81] noticed that a factor $\log N$ savings in memory was possible for locate queries of the special fractional dimension 1.5, Edelsbrunner [Ed81] developed an initial application of [Mc81]'s result to integer dimensions using $\log N$ more space and [Ch83] refined this idea to prove that $O(N \log^{k-1} N / \log \log N)$ space made possible $O(\log^{k-1} N)$ locate time in a complexity model where the copy component increases by a factor of 2. The practical applications of Chazell's proposal are unclear since it is inapplicable to aggregates retrievals, its update time is not as good as $T_b(K)$ and most of its $\log \log N$ improvement in space is offset by either the increased value of the memory coefficient or the weaker complexity model. However, [Ch83] is a very eloquent paper, and we recommend both it and the earlier work to the reader.

Extensions of Fredman's formalism [Fr81a], [Fr81b] can probably produce a lower bound $\Omega(\log^{k-1} N)$ on the sum of any data structure's expected insertion, deletion, and k -dimensional locate-retrieval complexities. An open question is whether or not a simultaneous matching upper bound for worst-case insertions and deletions is possible.

Articles relevant to this open question include [Ch83], [Ed81], [Fr81a], [Fr81b], [Mc81], [WiLu84], [Ya82]. Edelsbrunner and Overmars have recently noted that the memory space of all augmented trees can be reduced to $O(N)$ for the special case of batch environments [EdOv83]. [Wi83a], [Wi83b], [Wi84a], [Wi84b] have recently developed some extremely practical ideas about how to apply range query theory to typical commercial data base problems, and we recommend these articles in the strongest terms to the reader.

Appendix A. Recall that Step 5 of our algorithm for insertions and deletions in $T_e(2)$ consumed time proportional to the cardinalities of the involved downsets, and that several steps of our algorithm for $T_d(2)$ consumed time proportional to the logarithms of these cardinalities. This appendix will prove that the downsets in these steps have expected cardinality $O(1)$ under any random control sequence, thus showing these steps are highly efficient. The following lemma is a useful preliminary proposition; it is also helpful in explaining the intuition behind much of our analysis.

LEMMA 3. *The average size of a downset in any tree $T_e(2)$ or $T_d(2)$ will always be slightly less than 2.*

Proof. Let v_l and v_r denote the left and right sons of an internal node v , belonging to the base of any 2-fold tree. Note that the sum of the cardinalities of $AUX(v_l)$ and $AUX(v_r)$ will exceed by precisely one the cardinality of their father auxiliary field $AUX(v)$. (This slight difference in size arises because the pseudo-entry infinity is present in each of the three auxiliary fields.) This observation implies that the average size of a DOWNSET, taken over the union of these two sibling auxiliary fields, must be slightly less than two. Since the same statement can be made about any pair of brothers, the average downset-size over the whole base-tree must also be slightly less than 2. Q.E.D.

Although Lemma 3 provides some useful intuition about the average size of the downsets, it does not provide the precise information needed to calculate the expected cardinality for many of the probability distributions needed in this section. The difficulty

is that Lemma 3 calculates an average based on an assignment of equal weight to every downset, while our analysis will also need a calculation where the weight of a downset is proportional to its cardinality. The expected size $O(1)$ under the latter average is proven below.

LEMMA 4. *Consider either a $T_e(2)$ or $T_d(2)$ tree that is built by the algorithms from § 5 in response to some random control sequence. Let v denote any internal node in the base of this tree other than the root (which is not interesting because it is associated with no downsets). Then the expected value of the cardinality of $\text{DOWNSET}(X)$ will be $O(1)$, under any random process which selects entries X from $\text{AUX}(v)$ with a probability proportional to the cardinality of $\text{DOWNSET}(X)$.*

Proof sketch. Let f denote the father of v and N_f and N_v the ranks of these two nodes. Then $N_v \cong \alpha N_f$ because all trees manipulated in § 5 are $BB(\alpha)$. The expected size of the downsets will be greatest when the preceding inequality degenerates to an equality; therefore, the rest of this proof can make the conservative assumption $N_v = \alpha N_f$.

Let $e(j, v)$ denote the expected fraction of entries in $\text{AUX}(v)$ whose downsets have cardinality precisely equal to j . Since random control sequences generate ordered pairs whose two components are stochastically independent, the value of $e(j, v)$ can be calculated using standard probability models where all permutations occur with equal probabilities. Under our declared assumption $N_v = \alpha N_f$, the fraction $e(j, v)$ will respect the following limit for any $j \geq 1$:

$$(A.1) \quad \lim_{N_v \rightarrow \infty} e(j, v) = \alpha(1 - \alpha)^{j-1}.$$

A more detailed analysis of this limit will show that $e(j, v)$ converges quickly enough to assure the existence of a constant $c > 0$ such that all j and v satisfy the inequality:

$$(A.2) \quad e(j, v) \leq c \cdot (1 - \alpha)^j.$$

It is also apparent that the sum $\sum_{j=1}^{\infty} j^2 e(j, v)$ bounds the expected cardinality of the downsets under any random variables satisfying the hypothesis of Lemma 4. In view of equation (A.2), this expected value is bounded by $\sum_{j=1}^{\infty} cj^2(1 - \alpha)^j$, which is finite since $0 < \alpha < 1$. Hence, we have proven the existence of a constant that always bounds the expected sizes of the downsets under the assumptions of Lemma 4. Q.E.D.

Lemmas 3 and 4 are relevant to the algorithms of § 5 because all the downsets which these algorithms process respect probability distributions described by one of these two lemmas, by a fairly trivial inspection of their different steps. Thus, the relevant downsets have the expected sizes $O(1)$, which was needed in the proofs in § 5.

Appendix B. One disadvantage of the tree $T_e(k)$ is that it does not guarantee the worst-case of insertion and deletion. This section will describe a modified k -fold tree, called $T_f(k)$, which satisfies the same retrieval-time and memory-occupancy complexities as $T_e(k)$, and also guarantees a worst-case sequence-average complexity $O(\log^k N)$ for insertions. The surprising characteristic of $T_f(k)$ is that it controls insertion time only when deletions are absent.

For simplicity, we again focus on the case $k = 2$. Then $T_f(2)$ will be defined to be a data structure identical to $T_e(2)$, except that each entry X in an auxiliary field will contain two additional pointers, $\text{LEAST}(X)$ and $\text{GREATEST}(X)$, to the elements in $\text{DOWNSET}(X)$ with the smallest and largest $\text{KEY} \cdot 1$ values.

Consider the insertion algorithm outlined in the proof of Theorem 5. Let I_X denote the size of $\text{DOWNSET}(X)$ before the execution of step 5 of that algorithm, and I_X^* its size after step 5. Step 5 has a worst-case cost $O(I_X)$ in the context of Theorem 5,

but the following lemma shows its cost can be reduced under the new data structure $T_f(2)$.

LEMMA 5. *The tree $T_f(2)$ makes it possible to revise steps 4 and 5 of Theorem 5's insertion algorithm so that the worst-case time of these steps is proportional to $\text{MIN}(I_X - I_X^*, I_X^*)$.*

Proof. Since the elements of $T_f(2)$'s auxiliary fields are doubly linked lists ordered by KEY·1 and since entry X has pointers to LEAST(X) and GREATEST(X), a revised algorithm can certainly determine in time $\text{MIN}(I_X - I_X^*, I_X^*)$ whether the inequality $I_X^* > I_X - I_X^*$ holds or not.

The first part of our revised algorithm will test this inequality. If it holds, the second part will apply steps 4 and 5 of Theorem 5's algorithm to the record R and the entry X , just as before. Otherwise, steps 4 and 5 of the old algorithm will be replaced by a procedure which

- a) places the record R in the position previously used by X ,
- b) places X 's information in a new entry immediately to the right of R , and
- c) updates the rightmost I_X^* elements in DOWNSET(X) so that their downpointers reflect X 's movement.

It is desirable to replace steps 4 and 5 with the procedure above when $I_X^* \leq I_X - I_X^*$ because the latter alternative is less expensive when this inequality holds. That is, the quantities $I_X - I_X^*$ and I_X^* represent the numbers of downpointers needing modifications under our two methods; the suggested algorithm executes whichever method is cheaper. Its cost is therefore $\text{MIN}(I_X^*, I_X - I_X^*)$. Q.E.D.

THEOREM 8. *If the tree $T_f(2)$ is initially empty, if absolutely no deletion commands occur, and if insertions are executed by Lemma 5's modification of Theorem 5's procedure, then these operations will have a worst-case sequence-average complexity $O(\log^2 N)$.*

Proof. We will apply the principles of [WiLu84] and Lemma 5 to prove Theorem 8. Let t denote the total rank of the base our k -fold tree. The proof-sketch of Theorem 4 indicated that

- i) t increases by no more than $O(\log N)$ after the insertion of a record; and
- ii) a rotation by the bounded-balance algorithm taking time M will decrease t by at least $\Omega(M)$.

Let E denote the set of entries stored in the k -fold tree's auxiliary fields, and A_c the accounting function

$$A_c = ct \cdot \log N + \sum_{X \in E} (I_X \cdot \log I_X),$$

for some constant c . Lemma 5 combined with observations (i) and (ii) implies that if we assign c a sufficiently large constant value then A_c will satisfy the following three conditions:

- A) Each insertion of a record R by step 1 of the Theorem 5 algorithm will increase A_c by no more than $O(\log^2 N)$.
- B) An adjustment of the k -fold tree taking time $O(M)$ by either a bounded balance rotation (i.e., step 6) or an application of the procedure from Lemma 1 will decrease A_c by at least $\Omega(M)$.
- C) No other aspect of Theorem 5's procedure (steps 2, 3, or 7) will increase A_c .

Clearly A_c initially equals zero, since $T_f(2)$ is initially empty. Since A_c cannot go negative, observations (A) through (C) imply that the aggregate time consumed by all executions of step 6 and the Lemma 5 procedure during a sequence of n commands is $O(n \log^2 N)$. These aspects of the insertion algorithm therefore satisfy the sequence-average constraint $O(\log^2 N)$, and the other aspects satisfy an even tighter constraint $O(\log N)$, by § 5. Q.E.D.

Remark 7. The result above, combined with § 5 and [WiLu84], implies $T_f(k)$ meets all the complexities of $T_e(k)$, and additionally has a worst-case sequence-average insertion complexity $O(\log^k N)$ when deletions are absent. $T_f(k)$'s memory space is greater than $T_e(k)$'s by a small constant factor because of its two new pointers.

Acknowledgments. I would like to thank Eric Wolman and both referees for their suggestions on presentation.

REFERENCES

- [AhHoU174] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
- [AvSh81] Z. AVID AND E. SHAMIR, *A direct solution to range search and related problems for product regions*, Proc. 22nd Annual Symposium on Foundations of Computer Science, 1981, pp. 123-216.
- [Be75] J. L. BENTLEY, *Multidimensional binary search trees used for associative searching*, Comm. of ACM, 18 (1975), pp. 509-517.
- [Be80] ———, *Multidimensional divide-and-conquer*, Comm. of ACM, 23 (1980), pp. 214-228.
- [BeMa80] J. L. BENTLEY AND H. A. MAURER, *Efficient worst-case data structures for range searching*, Acta Inf., 13 (1980), pp. 155-168.
- [BeSa80] J. L. BENTLEY AND J. B. SAXE, *Decomposable searching problems #1: static to dynamic transformations*, J. Algorithms, 1 (1980), pp. 301-358.
- [BeSh77] J. L. BENTLEY AND M. I. SHAMOS, *A problem in multivariate statistics: algorithm, data structure, and applications*, Proc. Fifteenth Allerton Conference on Communication, Control, and Computing, 1977, pp. 193-201.
- [BeWo80] J. L. BENTLEY AND D. WOOD, *An optimal worst-case algorithm for reporting intersection of rectangles*, IEEE Trans. on Computers, 29 (1980), pp. 571-577.
- [Bo81] A. BOLOUR, *Optimal retrieval algorithms for small region queries*, this Journal, 10 (1981), pp. 721-741.
- [Ch83] B. CHAZELLE, *New algorithmic methods based on filtering*, 24th IEEE Symposium on Foundations of Computer Science, 1983, pp. 122-132.
- [CoYa83] R. COLE AND C. YAP, *Geometric retrieval problems*, 25th IEEE Symposium on Foundations of Computer Science, 1984, pp. 112-121.
- [Di82] P. F. DIETZ, *Maintaining order in a linked list*, Proc. 14th Annual ACM Symposium on Theory of Computing, 1982, pp. 122-128.
- [Di84] ———, an improved version of [Di82] to appear in Information and Control.
- [DoMu80] D. DOBKIN AND J. I. MUNRO, *Efficient use of the past*, Proc. 21st Annual Symposium on Foundations of Computer Science, 1980, pp. 200-206.
- [Ed80] H. EDELSBRUNNER, *Dynamic data structures for orthogonal intersections queries*, Technische Universität Graz Report #59, 1980.
- [Ed81] ———, *A note on dynamic range searching*, Bulletin of EATCS, 15 (1981), pp. 34-40.
- [EdOv83] H. EDELSBRUNNER AND M. H. OVERMARS, *Batch dynamic solutions for decomposable search problems*, V. Graz F118, 1983.
- [FiBe74] R. A. FINKEL AND J. L. BENTLEY, *Quad trees: a data structure for retrieval on composite keys*, Acta Inf., 4 (1974), pp. 1-9.
- [Fr81a] M. F. FREDMAN, *A lower bound on the complexity of orthogonal range queries*, J. ACM, 28 (1981), pp. 696-706.
- [Fr81b] ———, *Lower bounds on the complexity of some optimal data structures*, this Journal, 10 (1981), pp. 1-10.
- [Kn73] D. E. KNUTH, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
- [LeWo77] D. T. LEE AND C. K. WONG, *Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees*, Acta Inf., 9 (1977), pp. 23-29.
- [LeWo80] ———, *Quintary tree: a file structure for multidimensional database systems*, ACM Transactions on Database Systems, 5 (1980), pp. 339-347.
- [LeWo81] ———, *Finding intersection of rectangles by range search*, J. Algorithms, 2 (1981), pp. 337-347.

- [Lu78] G. S. LUEKER, *A data structure for orthogonal range queries*, Proc. 19th Annual Symposium on Foundations of Computer Science, 1978, pp. 28–34.
- [Lu79] ———, *A transformation for adding range restriction capability to dynamic data structures for decomposable searching problems*, Technical Report #129, Department of Information and Computer Science, Univ. California, Irvine, 1979.
- [LuWi82] G. S. LUEKER AND D. E. WILLARD, *A data structure for dynamic range queries*, Inf. Proc. Letters, 15 (1982), pp. 209–213.
- [Mc81] E. M. MCCREIGHT, *Priority search trees*, Xerox Report CSL-81-5, Xerox PARC, Palo Alto, 1981, this Journal, 14 (1985), to appear.
- [NiRe73] J. NIEVERGELT AND E. M. REINGOLD, *Binary search of bounded balance*, this Journal, 2 (1973), pp. 33–43.
- [Ov83] M. H. OVERMARS, *The design of dynamic data structures*, Lecture Notes in Computer Science 156, Springer-Verlag, Berlin, 1983.
- [OvLe82] M. OVERMARS AND J. VAN LEEUWEN, *Dynamics multidimensional data structures based on quad and K-d trees*, Acta Inf. 17 (1982), pp. 267–286.
- [VaWo80] V. VAISHNAVI AND D. WOOD, *Data structures for rectangle containment and enclosure problems*, Computer Graphics and Image Processing, 13 (1980), pp. 372–384.
- [Wi78a] D. E. WILLARD, *Predicate-oriented database search algorithms*, Ph.D. thesis, Harvard University, 1978. Also in Outstanding Dissertations in Computer Science, Garland Publishing, New York, 1979.
- [Wi78b] ———, *New data structures for orthogonal queries*, Technical Report TR-22-78, Center for Research in Computing Technology, Harvard Univ., Cambridge, MA, 1978 (the earlier draft of this report).
- [Wi78c] ———, *Balanced forests of K-d* trees as a dynamic data structure*, Technical Report TR-23-78, Center for Research in Computing Technology, Harvard Univ., Cambridge, MA, 1978.
- [Wi79] ———, *The super-B-tree algorithm*, Technical Report TR-03-79, Center for Research in Computing Technology, Harvard Univ., Cambridge, MA, 1979.
- [Wi80] ———, *K-d trees in a dynamic environment*, Technical Report No. 80-1, Department of Computer Science, Univ. Iowa, Ames, 1980.
- [Wi81] ———, *Two very fast tree data structures*, Proc. 19th Annual Allerton Conference on Communication, Control and Computing, 1981, pp. 355–363.
- [Wi82] ———, *Polygon retrieval*, this Journal, 11 (1982), pp. 149–162.
- [Wi83a] ———, *Predicate retrieval theory*, 21st Allerton Conference on Communication Control and Computing, 1983, pp. 663–674.
- [Wi83b] ———, *Abstract predicate retrieval theory*, SUNY Albany Technical Report 83-3, 1983.
- [Wi84a] ———, *A very efficient algorithm for evaluating relational calculus expressions*, 1984 ACM SIGMOD Conference.
- [Wi84b] ———, *A sampling algorithm for differentiable batch retrieval problems*, 11th Int. Colloquium on Automata, Languages and Programming (ICALP-1984).
- [Wi84c] ———, *New Trie data structures which support very fast search operations of order $\sqrt{\log M}$* , J. Comput. System Sci., in June 1984 issue.
- [WiLu84] D. E. WILLARD AND G. S. LUEKER, *Adding range restriction capability to dynamic data structures*, J. ACM, to appear.
- [Ya82] A. C. YAO, *On the complexity of maintaining partial sums*, 14th ACM STOC Symposium (1982), pp. 128–135.
- [Ya83] F. YAO, *A 3-space partition and its applications*, 15th ACM Conference on Theory of Computing, 1983, pp. 258–264.

ADDENDUM:
**SIMPLE LINEAR-TIME ALGORITHMS TO TEST CHORDALITY OF
GRAPHS, TEST ACYCLICITY OF HYPERGRAPHS, AND SELECTIVELY
REDUCE ACYCLIC HYPERGRAPHS***

ROBERT E. TARJAN† AND MIHALIS YANNAKAKIS‡

Jack Edmonds (private communication) has raised the question of efficiently finding an unchorded cycle in a nonchordal graph. We can find such a cycle in $O(n + m)$ time with the help of the chordality-testing algorithm of § 2. We need a variant of Lemma 4. Let $G = (V, E)$ be a graph with vertices numbered from 1 to n so that property P holds (see Lemma 4). A *violating triple* is a triple of vertices u, v, w such that $u <_{\alpha} v <_{\alpha} w$, $\{u, v\} \in E$, $\{u, w\} \in E$, and $\{v, w\} \notin E$. A *maximum violating triple* is a violating triple u, v, w that is lexicographically maximum with respect to the numbers of u, v , and w . (Vertex u is chosen to be of maximum number, with a tie broken by choosing v of maximum number, and a secondary tie broken by choosing w of maximum number.)

LEMMA 6. *Let $G = (V, E)$ be a graph, let α be an ordering of G with property P, and let u, v, w be a maximum violating triple. Then there is a path from v to w containing neither u nor any vertex adjacent to u except v and w .*

Proof. An *ascending path* is a path x_0, x_1, \dots, x_k such that $x_i <_{\alpha} x_{i+1}$ for $0 \leq i < k$. Let X be the set of vertices containing u and all its adjacent vertices except v and w . Let Y be the set of vertices reachable from v by an ascending path containing no vertices of X . Let Z be the set of vertices reachable from w by an ascending path containing no vertices of X . If Y and Z contain a common vertex, the lemma is true. Thus suppose $Y \cap Z = \emptyset$.

There can be no pair of vertices $y \in Y, z \in Z$ such that $\{y, z\} \in E$, for $y <_{\alpha} z$ implies $z \in Y$ and $y >_{\alpha} z$ implies $y \in Z$. Let z be the maximum-numbered vertex in $Y \cup Z$. Suppose $z \in Z$. (The case $z \in Y$ is similar.) Let y be the maximum-numbered vertex in Y , let $z_0 = u$, and let $z_1 = w, z_2, \dots, z_k = z$ be an ascending path from w to z containing no vertices in U . For $1 \leq i \leq k, z_i \in Z$.

Let z_i, z_{i+1} be the pair such that $z_i < y < z_{i+1}$. Since $\{y, z_{i+1}\} \notin E$, Property P implies the existence of a vertex x such that $x >_{\alpha} y, \{x, y\} \in E$, and $\{x, z_i\} \notin E$. If $\{x, u\} \notin E$, then $x \in Y$, contradicting the choice of y as the maximum-numbered vertex in Y . But if $\{x, u\} \in E$, then since $\{x, z_i\} \notin E$ there must exist some j in the range $0 \leq j < i$ such that $\{x, z_j\} \in E$ and $\{x, z_{j+1}\} \notin E$. But then either z_j, x, z_{j+1} or z_j, z_{j+1}, x is a violating triple, contradicting the choice of u, v, w as the maximum violating triple. We conclude that $Y \cap Z \neq \emptyset$, i.e. the lemma is true. \square

We can use Lemma 6 in the following way to find an unchorded cycle in a nonchordal graph G . We number the vertices of G so as to satisfy P, e.g. by performing a maximum cardinality search and numbering the vertices from n to 1. For each vertex v we compute the follower $f(v)$ of v . Applying the zero-fill-in test of § 2, we find the vertex u of maximum number such that, for some edge $\{u, w\}$, $f(u) \neq w$ and $\{f(u), w\} \notin E$. Then $u, v = f(u), w$ is a violating triple; furthermore it is a violating triple with u of maximum number, since the graph formed by deleting from G all vertices numbered $\alpha(u)$ or lower must be chordal.

* This Journal, 13 (1984), pp. 566-579. Received by the editors May 29, 1984.

† AT & T Bell Laboratories, Murray Hill, New Jersey 07974.

Having found a violating triple with u of maximum number, we find the maximum violating triple by finding all the edges $\{x, y\}$ such that $\{u, x\} \in E$ and $\{u, y\} \in E$, lexicographically sorting them on the pair $(\min\{\alpha(x), \alpha(y)\}, \max\{\alpha(x), \alpha(y)\})$, and scanning the sorted list to find the largest missing pair, i.e. the pair v, w such that $\{u, v\} \in E$, $\{u, w\} \in E$, $\{v, w\} \notin E$, $v <_{\alpha} w$, and $(\alpha(v), \alpha(w))$ is lexicographically largest. Then u, v, w is the maximum violating triple.

The last step is to extend the triple u, v, w to an unchorded cycle. To do this we find a path of fewest edges from v to w that avoids u and vertices adjacent to u other than v and w . Such a path exists by Lemma 6 and together with u, v, w forms an unchorded cycle. The path can be found in $O(n + m)$ time using breadth-first search. Finding the maximum violating triple also takes $O(n + m)$ time, giving a linear time bound overall.

PRIORITY SEARCH TREES*

EDWARD M. McCREIGHT†

Abstract. Let D be a dynamic set of ordered pairs $[x, y]$ over the set $0, 1, \dots, k-1$ of integers. Consider the following operations applied to D :

- (1) Insert (delete) a pair $[x, y]$ into (from) D .
- (2) Given test integers x_0, x_1 , and y_1 , among all pairs $[x, y]$ in D such that $x_0 \leq x \leq x_1$ and $y \leq y_1$, find a pair whose x is minimal (or maximal).
- (3) Given test integers x_0 and x_1 , among all pairs $[x, y]$ in D such that $x_0 \leq x \leq x_1$, find a pair whose y is minimal.
- (4) Given test integers x_0, x_1 , and y_1 , enumerate those pairs $[x, y]$ in D such that $x_0 \leq x \leq x_1$ and $y \leq y_1$.

Using a new data structure that we call a priority search tree, of which two variants are introduced, operations (1), (2), and (3) can be implemented in $O(\log n)$ time, where n is the cardinality of D . Operation (4) is performed in at most $O(\log n + s)$ time, where s is the number of pairs enumerated. The priority search tree occupies $O(n)$ space.

Priority search tree algorithms can be used effectively as subroutines in diverse applications. With them one can answer questions of intersection or containment in a dynamic set of linear intervals. They can be used in combination with a well-known plane-sweep technique, to implement off-line algorithms for enumerating all intersecting pairs of rectangles. Priority search trees can also be used to implement best-/first-fit storage allocation.

Key words. computational geometry, search trees, priority queues, intersection, intervals, rectangles, storage allocation, concrete complexity

CR categories. 5.25, 3.74, 5.39

1. Introduction. Efficient multi-dimensional searching is one of the persistent puzzles of computer science. Many lovely one-dimensional search structures with linear space requirements and guaranteed logarithmic-time maintenance and search algorithms have been discovered. But multi-dimensional structures with similar attractive properties continue to elude discovery.

We present here a new data structure, called a priority search tree, for representing a dynamic set D of ordered pairs $[x, y]$ over the set $0, 1, \dots, k-1$ of integers, and a set of algorithms that operate on the priority search tree to implement the following operations:

InsertPair (x, y): Insert a pair $[x, y]$ into D .

DeletePair (x, y): Delete a pair $[x, y]$ from D .

MinXInRectangle (x_0, x_1, y_1): Given test integers x_0, x_1 , and y_1 , among all pairs $[x, y]$ in D such that $x_0 \leq x \leq x_1$ and $y \leq y_1$, find a pair whose x is minimal.

MaxXInRectangle (x_0, x_1, y_1): Given test integers x_0, x_1 , and y_1 , among all pairs $[x, y]$ in D such that $x_0 \leq x \leq x_1$ and $y \leq y_1$, find a pair whose x is maximal.

MinYInXRange (x_0, x_1): Given test integers x_0 and x_1 , among all pairs $[x, y]$ in D such that $x_0 \leq x \leq x_1$, find a pair whose y is minimal.

EnumerateRectangle (x_0, x_1, y_1): Given test integers x_0, x_1 , and y_1 , enumerate those pairs $[x, y]$ in D such that $x_0 \leq x \leq x_1$ and $y \leq y_1$.

This searching might fairly be described as 1.5-dimensional. The data has two independent dimensions, but the priority search tree does not allow equally powerful searching operations on both. There is a major dimension (x) permitting arbitrary range queries, and a minor one (y) permitting only enumeration in increasing order.

* Received by the editors July 14, 1980, and in final revised form April 26, 1983.

† Xerox Corporation, Palo Alto Research Center, Palo Alto, California 94304.

All the search rectangles have only three sides free; the fourth side is anchored at $y_0 = 0$.

In § 2 we present the simple radix priority search tree, and examine some of its properties. In § 3 we elaborate this to the balanced priority search tree. In § 4 we discuss a few of the applications to which these priority search trees can be put.

2. Radix priority search trees. First off, let us simplify the problem somewhat. In the following exposition we assume that the set D of pairs contains no duplicate x values. A restriction like this might or might not occur naturally in a real application. If not, we can work with a derived set D_π consisting of a pair $[F(x, y), y]$ for every pair $[x, y]$ in D . The function F is an invertible encoding function that maps pairs of integers into single integers with the property that differences in x are more significant than differences in y . For example, we might use the function $F(x, y) = j^*x + y$, which maps pairs of integers in the domain $0 \dots j-1$ to single integers in the range $0 \dots j^2 - 1 \cong k - 1$. Other ways of implementing such a function F are left to the reader's imagination. If $[x, y]$ pairs are unduplicated in the original problem, then x_π values are unduplicated in the derived problem. (Going even further, we could accommodate duplicated $[x, y]$ pairs in the original problem by representing them as unduplicated pairs with associated counts.)

The simple idea that underlies priority search trees is most easily seen from a diagram. Suppose that you wanted to represent the set of pairs in Fig. 1 so that **EnumerateRectangle** could be executed efficiently on this representation. One good way to do this is to select the pair $[x^*, y^*]$ with minimum y , write it at the root of a binary data structure, divide the region in two with a line of constant x , and recursively represent the remaining points in the two subregions in the two subtrees of the root in the same manner.

If one divides the region along the line $x = x^*$, then the resulting data structure is the Cartesian tree of Vuillemin [13]. This structure allows very good performance in the average case, but its performance in the worst case is no better than a linear

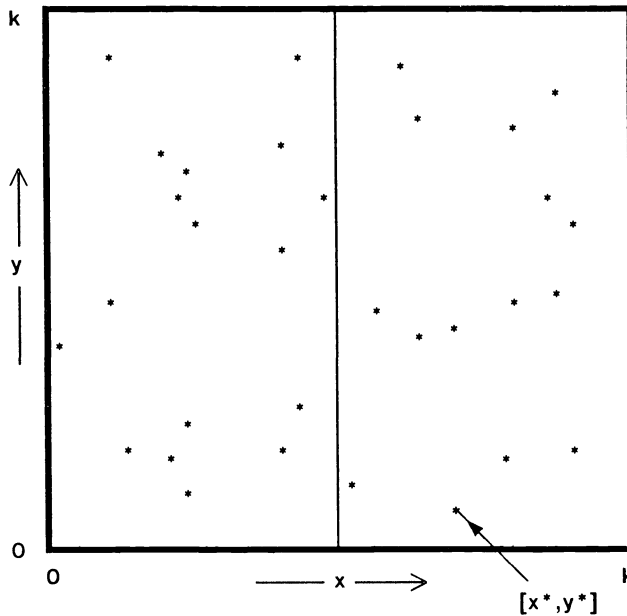


FIG. 1

list. For example, the set of pairs with $x = y$ forces the data structure to degenerate to a linear list.

Fortunately there is no compelling reason to divide the region along the line $x = x^*$. We define a radix priority search tree such that at each recursive level the previous level's x -interval is cut exactly in half (geographically). This is called a radix bisection, and it has two important consequences.

The first is that after at most $\lg k$ levels of bisection in x , we encounter a stripe in y that is one unit wide in x . By our nonduplication assumption, there can be at most one pair in D within this stripe. Therefore, even though a radix priority search tree might contain k pairs, it is at most $\lg k$ levels deep.

The other important consequence of radix bisection is that a node in a given position in the radix search tree represents a fixed rectangle in x - y space. The only way a pair enters or leaves one of these fixed rectangles is through an updating operation involving that pair. Therefore no lateral data motion is ever required during updating operations; an updating operation only moves down a single spine in the tree. This fact enables a $\lg k$ time bound on updating operations.

2.1. Data structure. We can represent a radix priority search tree in Pascal as follows:

```

CONST
  k = 30000;
  (*Comfortable for a 16-bit 2's complement machine*)
  FirstKey = 0;
  LastKey = k - 1;
  FirstNonKey = LastKey + 1;
TYPE
  KeyRange = FirstKey .. LastKey;
  KeyBound = FirstKey .. FirstNonKey;
  Pair = RECORD x, y: KeyRange END;
  RPSTPtr = ↑RPST;
  RPST = RECORD
    p: Pair;
    left, right: RPSTPtr
  END;

```

A radix priority search tree is characterized by a fidelity condition and two data structural invariants. The fidelity condition asserts that if the tree is representing a set D of pairs, then each pair of D will appear in the p field of exactly one node (or RPST record) of the tree. Thus a tree representing a set of n pairs occupies $O(n)$ words of storage, where each word is $O(\log k)$ bits long. In conventional algorithm-analytic terms, a radix priority search tree is a linear-space data structure.

The first invariant is a priority queue invariant on y -values. It asserts that for any node t in a radix priority search tree, if $t.left$ is not NIL then $t.p.y \leq t.left.p.y$, and if $t.right$ is not NIL then $t.p.y \leq t.right.p.y$. This first invariant constrains only direct ancestor-descendant relations; it does not constrain sibling relations at all.

The second invariant is a radix search tree invariant on x -values. It asserts that associated with each node t in a radix priority search tree is an x -interval [**lower** .. **upper**) (this notation denotes all integers between **lower** and **upper**, including **lower** and excluding **upper**) within which $t.p.x$ lies. The x -interval associated with the root of the radix search tree is the interval **KeyBound**. For any node t such that $t.left$ is not

NIL, the x -interval associated with the node $t.left\uparrow$ is $[lower .. floor((lower+upper)/2))$. For any node t such that $t.right$ is not **NIL**, the x -interval associated with the node $t.right\uparrow$ is $[floor((lower+upper)/2) .. upper)$.

2.2. Algorithms. The complete radix search tree algorithms, represented in Pascal, are presented in Appendix A, and the reader is encouraged to read the following in parallel with Appendix A.

In all of these algorithms, two preconditions are assumed true, and their truth is maintained in recursive calls. The first precondition is that the interval $[lowerX .. upperX)$ is nonempty; that is, that $lowerX < upperX$. To maintain the truth of this first precondition in recursive calls, the algorithms depend upon our assumption that no two pairs have the same x -value. The second precondition is that whenever a procedure takes an $[x0 .. x1]$ argument range as a parameter, the procedure is only called if the interval $[x0 .. x1]$ shares at least one integer in common with the interval $[lowerX .. upperX)$.

First consider the **InsertPair** procedure. To insert a new pair, we begin at the root of the priority search tree. First we discover whether the new pair “beats” the pair already sitting at the root, in the sense that its y -value is smaller. If not, then the new pair is inserted recursively into either the left or right subtree, determined by its x -value. Otherwise, the new pair belongs at the root, so the pair that originally lay at the root is saved, the new pair is put there instead, and the saved pair is inserted recursively into the subtree determined by its x -value.

DeletePair operates in two distinct phases. The first phase locates the pair to be deleted and it operates as a search in an ordered search tree. Once the pair to be deleted has been located its deletion leaves a hole, which is filled by a priority queue tree selection (or a “knock-out tournament”) phase [1], in which a pair of brothers compete for the vacated spot formerly occupied by their father, and then the sons of the victor compete for his former spot, and so on. The second phase completes when the vacant spot has fewer than two sons.

Consider **MinXInRectangle** applied to a subtree rooted at node t . If $t.p.y$ lies above the top of the constraint rectangle, then because a priority search tree is a priority queue in y , no pair in the subtree rooted at t lies within the constraint rectangle. Otherwise, the solution might be found in the left subtree. If no pair in the left subtree lies within the constraint rectangle, then (and only then) the solution might be found in the right subtree. This is because every constrained pair in the left subtree is better than any constrained pair in the right subtree. Finally, if $t.p$ lies within the constraint rectangle, then it might or might not be the correct solution, depending on whether it is better than the best constrained pair found in a subtree. **MaxXInRectangle** is entirely symmetric in x .

Next consider **MinYInXRange** applied to a subtree rooted at node t . If $t.p$ lies within the constraint x -interval, then because a priority search tree is a priority queue in y , $t.p$ is the correct solution. Otherwise, the solution, if it exists, is the better of the solutions of the two subtrees. The tests of **middleX** against $x0$ and $x1$ simply guarantee that subtrees that are certain to be fruitless are not explored. These tests also maintain the truth of the second precondition in recursive calls.

Finally, **EnumerateRectangle** is a depth-first enumeration that calls the function **Report** whenever a pair is found within the constraint rectangle. **Report** returns **TRUE** if the enumeration should continue, and **FALSE** if it should terminate.

Each of these procedures is called at the top level with $t\uparrow$ being the root of a radix priority search tree, and with **lower** being **FirstKey** and **upper** being **FirstNonKey**.

2.3. Execution time analysis. The logarithmic time bounds can be seen from the recursive structure of the algorithms. Each recursive level of **InsertPair** is called on a node with a [**lower** .. **upper**] interval, and makes at most one recursive call on **InsertPair**, handing it a son node with a [**lower** .. **upper**] interval at most half as large. The recursion must stop before the size of this [**lower** .. **upper**] interval shrinks to zero. This implies a bound of $\lg k$ on the depth of recursion and the number of nodes visited, and the same order found on running time. An identical analysis applies to **DeletePair**.

The analyses of **MinXInRectangle** (and, symmetrically, **MaxXInRectangle**), **MinYInXRange**, and **EnumerateRectangle** are more complicated because each sometimes calls itself recursively on both sons of a tree node. How many nodes can these procedures visit? We begin to answer this question by classifying tree nodes according to how their [**lower** .. **upper**] intervals, denoted by $\langle \rangle$, compared with the interval [**x0** .. **x1**], denoted by $\{ \}$. There are six such classes:

- 1: $\langle \rangle$, 2: $\{ \langle \rangle$, 3: $\langle \{ \}$, 4: $\{ \{ \}$, 5: $\{ \langle \{ \}$, and 6: $\{ \langle \rangle$.

Neither **MinXInRectangle** nor **MinYInXRange** ever visits nodes in classes 1 or 2; this is prevented by the second precondition. The second data structure invariant guarantees at most $\lg k$ nodes in each of classes 3, 4, and 5, so every one of these nodes could be visited without violating a logarithmic time bound. Finally, there can be a very large number of nodes in class 6, but these nodes can be grouped into maximal subtrees that are sons of nodes in classes 4 or 5, at most one such son each, so there are at most $2 \lg k$ of these maximal subtrees. Within each of these class-6 subtrees, any **t.p.x** lies within the interval [**x0** .. **x1**], so **MinYInXRange** will be prevented by its second **IF** statement from exploring beyond the roots of these maximal subtrees. This shows a time bound for **MinYInXRange** that is logarithmic in k .

An identical argument applies to **EnumerateRectangle** on all node classes except 6. In each of the class-6 maximal subtrees, beyond that subtree's root level **EnumerateRectangle** can visit a node only if the pair in the node's father was **Report**'ed. It follows that if **EnumerateRectangle** in fact enumerates s pairs, it runs in a time bound of $\lg k + s$. This is true whether or not the enumeration is terminated by the **Report** function.

The operation of **MinXInRectangle** is more subtle, because **MinXInRectangle** might find its answer deep in a subtree of class-6 nodes. The key observation is that once a single recursive instance of **MinXInRectangle** succeeds (in the sense that it returns a **valid CondPair**), there will be no further recursive calls of **MinXInRectangle**, and all currently active recursive invocations (of which there can be at most $\lg k$, the length of the longest path in the tree) will succeed. In other words, a top-level call to **MinXInRectangle** will generate some number of recursive invocations applied to various nodes of the tree that will fail, plus at most $\lg k$ invocations applied to other nodes that will succeed. Now how many recursive invocations might fail? We observe that whenever **MinXInRectangle** will fail when applied to a class-6 node, then it will fail on its second **IF** statement, in constant time and without making any further recursive calls. Thus a loose count concludes that **MinXInRectangle** can encounter at most $3 \lg k$ nodes of classes 3, 4, and 5, and at most $\lg k$ successful nodes, and therefore at most $4 \lg k$ other (failed) class-6 nodes. This confirms a time bound for **MinYInXRange** that is logarithmic in k . Closer reasoning tightens the constants considerably.

I have tried to code the procedures in Appendix A for clarity. There are several straightforward program transformations that would improve execution time by significant constant factors. The most important of these replace recursion with iteration and division by 2 with a binary shift.

Another important optimization reduces the number of unary (nonbinary) nodes within the tree in many applications. One way of thinking about the algorithms in this section is that a search for x is steered left or right through the tree by the sequence of bits in the binary representation of x , one bit per left/right decision. The definition of **RPST** can be augmented with a bit count field to allow a single left/right decision to consume several bits of the binary representation of x , thereby eliminating unary nodes for the intermediate bits. Depending on the application, this optimization can result in large reductions of average path length, with attendant improvements in speed.

3. Balanced priority search trees. Careful consideration of the literature on search structures suggests that when a radix structure permits certain operations to be performed in certain asymptotic time bounds, there almost always exists a parallel balanced comparative structure (that is, a structure within which order may be inferred only by comparing with keys that are present in the structure) that permits the same operations to be performed in the same asymptotic time bounds. It would be a surprise and a disappointment if this observation did not also hold true for priority search trees.

Fortunately, it does hold true. The structure of a balanced priority search tree node can be expressed in Pascal as follows:

```

TYPE
  BPSTPtr = ↑BPST;
  BPST = RECORD
    p, q: Pair;
    p, q: Pair;
    validP, duplQ: BOOLEAN;
    left, right: BPSTPtr;
    balance: BalanceInfo (*appropriate to the
      underlying tree form chosen*)
  END;

```

A balanced priority search tree is characterized by a fidelity condition and four data structural invariants. The fidelity condition asserts that if the tree is representing a set D of pairs, then each pair of D will appear in the **q** field of exactly one node of the tree. Thus a balanced priority search tree is also a linear-space data structure.

In a **BPST** node, unlike a **RPST** node, two pairs can be recorded: the pair **q** is chosen for its near-median x -value, while the pair **p** is chosen for its minimal y -value. Two pairs are necessary because, as we saw in § 2, for some sets of pairs it is impossible to satisfy both criteria with the same pair. Any pair $[x, y]$ in D appears as the **q** field of exactly one node **t**, and may also appear as the **p** field of at most one ancestor node of **t**.

The first structural invariant is a standard search tree invariant on **q.x**. It asserts that with each node **t** in a balanced priority search tree is associated a search key interval $[x_0 \dots x_1)$ containing **t.q.x**, and also containing **t.p.x** if **t.validP** is true. The x -interval associated with the root of the search tree is **KeyBound**. For any node **t**, if **t.left** is not **NIL**, then the x -interval associated with the node **t.left**↑ is $[x_0 \dots t.q.x)$. Similarly, if **t.right** is not **NIL**, then the x -interval associated with the node **t.right**↑ is $[t.q.x \dots x_1)$.

The second structural invariant is a priority queue invariant on **p.y**. Let **t** be any node in a balanced priority search tree, and let **a** be a proper ancestor of **t**, and **d** a proper descendant of **t**. If $\{a.p\} \ni \{d.q\}$ then **t.validP** is **FALSE**. Otherwise **t.validP** is **TRUE**, and **t.p** is a pair chosen from $\{d.q\} - \{a.p\}$ so that **t.p.y** is minimal. In other words,

$t.p$ is a pair with minimal y that appears as the q field of one of t 's descendants and does not appear as the p field of any of t 's ancestors. If no such pair exists, $t.validP$ is **FALSE**. It is easy to show that if $validP$ is **FALSE** at t , it is **FALSE** at all of t 's descendants as well. Conversely, if $validP$ is **TRUE** at t , it is **TRUE** at all of t 's ancestors as well.

The third structural invariant specifies the **duplQ** field. It asserts that the field $t.duplQ$ is **TRUE** if and only if there is some proper ancestor node a of t such that $a.validP = \text{TRUE}$ and $a.p = t.q$. This field allows our algorithms easily to avoid duplicate enumeration of pairs.

The fourth and final structural invariant is a balance invariant inherited from whatever underlying form of balanced tree is chosen. This invariant is usually a relation between weights or path lengths in the left and right subtrees of a node, or on the sequence of "colors" on arcs leading to the node.

The operation necessary for maintaining balance in all known forms of balanced comparative tree is some form of "rotation." [1, p. 454] This is a way of moving some "weight" from the "heavier" subtree of a node to the "lighter" one, thereby preserving the balance invariant that guarantees a maximal path length at most logarithmic in the number of nodes. The **BalanceInfo** field in the type of definition allows determination of when and where to do these rotations.

Figure 2 shows the standard picture of a single rotation. Lower-case letters indicate points along the x -value line, and also tree nodes whose $q.x$ fields contain those points. Upper-case letters indicate intervals on this line, and also subtrees containing nodes whose $p.x$ and $q.x$ fields lie within those intervals. All intervals are assumed to include their lower endpoint, and exclude their upper one.

During the priority search tree rotation, the q fields remain unchanged, just as they would in an ordinary balanced tree rotation. The interesting question is, what happens to the p fields? First of all, it is clear that node c can use node e 's original p field, because it represents the "best" pair in the $[a..g)$ interval that is not represented higher in the tree. Now, what happens to node c 's original p field, and where does the

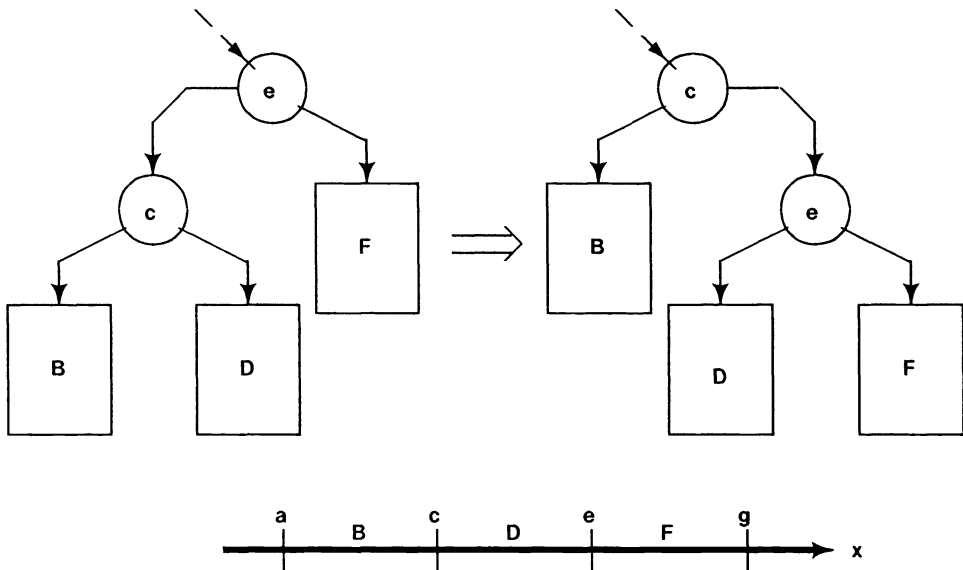


FIG. 2

new **p** field for node **e** come from? As luck might have it, **c**'s original **p** field might lie in the interval **D**, and if so it would be a candidate for **e**'s new **p** field. But in the general case we need to dispose of **c**'s old **p** field in one of **c**'s old subtrees, and to extract a new **p** field for node **e** from one of its new subtrees.

Appendix B contains Pascal procedures for doing just that, along with skeleton procedures for the balanced forms of **InsertPair** and **DeletePair** that manipulate the **p**, **q**, **validP** and **duplQ** fields correctly but ignore re-balancing. The balanced forms of the other algorithms are left as exercises for the interested reader.

The **DisposeP** and **ExtractP** procedures are each recursive down at most one path in the tree, so their execution time is at most linear in the length of the longest path in the tree. In a balanced tree, this longest path length is at most logarithmic in the number of nodes in the tree. This means that in a balanced priority search tree **BalancedInsertPair** and **BalancedDeletePair** each run in logarithmic time, except for rotations, and that at most logarithmic extra time is needed for a single rotation.

To attain an overall logarithmic time bound, the number of rotations per updating operation must be bounded by a constant. The usual families of balanced tree, such as AVL [1, pp. 451–458], weight-balanced [1, p. 468], and *B*-tree [1, pp. 471–479], do not guarantee such a bound. Fortunately there are at least two families that do guarantee at most a constant (in fact, three) rotations per updating operation: 2-3-4 trees [14], [15], [16], and the half-balanced binary trees of Olivie [2].

4. Applications. There are many real computer applications involving a large number of data items where the size of the computer to be used mandates a linear-space data structure. All of the applications below require space only linear in the number of data items. In several cases algorithms are known with smaller asymptotic time bounds (generally by a factor of $\log n$) but larger space requirements (generally by the same factor) [10].

In all of the two-dimensional applications below, line segments are taken to be parallel to the *x*- or *y*-axis. In my own applications this restriction is not a serious one. But for others it may be, and it is not known in general how essential this restriction is for the existence of fast algorithms.

4.1. On-line intersections in a dynamic set of linear intervals. We can use a priority search tree to represent a dynamic set of linear intervals, letting the *x*-value of the pair represent the upper endpoint of an interval, and letting its *y*-value represent the lower endpoint. To enumerate all intervals that share at least one point with a test interval $[u \dots v]$, we use the **EnumerateRectangle** algorithm to enumerate all pairs whose *x*-value lies in the interval $[u \dots \text{LastKey}]$ and whose *y*-value lies in the interval $[\text{FirstKey} \dots v]$. If the set contains n intervals, then the structure to represent it requires $O(n)$ space, a new interval can be added or an old one deleted in $O(\log n)$ time, and all s intervals in the set that intersect a test interval can be enumerated in $O(\log n + s)$ time. Results almost as good as this have been discovered previously by McCreight [3] and Guibas and Saxe [4], and independently by Edelsbrunner [5], [6]. The improvement over [3] and [5] is that here the parameter k can be ignored because balanced priority search trees are used. The improvement over [4] and [6] is that here the time bound applies to each operation individually, rather than to an average taken over a sequence of operations.

4.2. On-line containments in a dynamic set of linear intervals. With exactly the same data structure we can enumerate all intervals that completely contain a test interval $[u \dots v]$ by using the **EnumerateRectangle** algorithm to enumerate all pairs whose

x -value lies in the interval $[v.. \text{LastKey}]$ and whose y -value lies in the interval $[\text{FirstKey}.. u]$. If the set contains n intervals, then all s intervals in the set that contain a test interval can be enumerated in $O(\log n + s)$ time. This is thought to be a new insult.

4.3. On-line visibility in a dynamic set of semi-infinite line segments. Suppose one has a dynamic set of semi-infinite line segments beginning at points $[x, y]$ and extending upward in y . From a given point $[x', y']$, which of these line segments would be visible along a line of increasing x ? To solve this problem one can represent the endpoints of the semi-infinite lines as $[x, y]$ pairs in a priority search tree. One could either think of the line segments as being translucent or opaque. In the former case, the solution is all pairs in the rectangle bounded on the left by $x = x'$ and above by $y = y'$, which can be enumerated by **EnumerateRectangle** in $O(\log n + s)$ time. In the latter case, the solution is the single pair within that rectangle whose x is minimal, which can be produced by **MinXInRectangle** in $O(\log n)$ time.

4.4. On-line visibility in a dynamic set of line segments. Relaxing the restriction in the previous problem that the line segments be semi-infinite gives the problem an additional degree of freedom. To deal with this extra degree of freedom we adapt a previous technique [3], [5] that recursively bisects the y -space, dividing the line segments at each level into three classes: segments that lie entirely above the bisector, segments that lie entirely below it, and segments that are cut by it. Segments that are not cut by the bisector are represented in deeper recursive levels. Segments that are cut by the bisector are represented in two priority search trees: one representing the pieces of the cut segments below the bisector, and one representing those above. In each of these two priority search trees the line segments are now semi-infinite, because they extend to the limit of the (reduced) y space. Therefore the solution from § 4.3 carries over for each recursive level, and there are $\log k$ such levels, so the translucent (opaque) problem can be solved in $O(\log n \log k (+s))$ time. This is also thought to be a new result.

4.5. On-line point containment in a dynamic set of rectangles. We apply recursive bisection one more time, this time in x . For those rectangles cut by the bisector, we now consider their left and right bisected pieces. These are symmetric, so we describe only how to deal with the right-hand pieces. Each of these rectangular pieces is completely described by the line segment of constant x that is its right-hand side, because its left-hand side is the bisector. The set of these right-hand sides can be handled as in § 4.4 above. A point is in one of these rectangular pieces whenever the right-hand side of the piece is visible along a line of increasing x from the point. Therefore the solution is simply $\log k$ iterations of the solution of § 4.4 above, and one can enumerate the set of all rectangles in a dynamic set of rectangles that contain a test point in $O(\log n \log^2 k + s)$ time, a further new result. By now the reader can see how to extend this indefinitely, so we shall next consider a different class of applications.

4.6. Off-line intersections among a set of rectangles. We can enumerate all intersecting pairs among a set of axis-aligned rectangles (rectangles with sides parallel to the axes) by using the plane sweep technique first proposed by Shamos and Hoey [7]. This technique simulates the motion of a horizontal line across a plane from bottom to top, and considers the sequence of cross-sectional slices induced by this line.

For aligned rectangles a cross-sectional slice is a set of horizontal intervals. As the sweep line moves upward onto a new rectangle, that rectangle's horizontal interval is added to the set. As the sweep line moves upward off a rectangle, that rectangle's

horizontal interval is removed from the set. Every time a new horizontal interval is added to the set, an enumeration is made of all other intervals in the set that touch the new interval.

We now analyze the performance of the rectangle intersection algorithm somewhat more carefully. The sweep technique requires that the rectangles be sorted by their bottom edges, and that their top edges be maintained in a priority queue. For n rectangles this takes $O(n)$ space and $O(n \log n)$ time. The priority search tree operations can be done in $O(n)$ space and $O(\log n + s)$ time apiece. Each rectangle causes one **InsertPair**, one **DeletePair**, and one **EnumerateRectangle** operation. Moreover each rectangle intersection is discovered by exactly one **EnumerateRectangle** operation, and therefore contributes to the s of that operation. Thus the overall time performance is $O(n \log n + s)$. This performance, which has been achieved before with more complex data structures [5], [10], is within a constant factor of the best possible worst-case performance.

An application like circuit extraction from IC masks might involve rectangles of several colors, and be concerned only with intersections between rectangles of different colors. For this purpose one could have a different priority search tree for each color. As the sweep line passes the bottom edge of a red rectangle, for example, the corresponding red horizontal interval would be inserted into the red priority search tree, while the same interval would be used in a **EnumerateRectangle** operation on every nonred priority search tree. Now arbitrary intersection patterns of red with red rectangles do not increase the time beyond $O(n \log n)$. The s term counts only the number of *inter-color* intersections.

4.7. Memory allocation. Many computer operating systems satisfy dynamic requests for memory according to a first-fit (use the free block of adequate size at the smallest address) or a best-fit (use the smallest free block of adequate size) policy. One might imagine that a synthesis of these two policies could perform better than either one separately, but at first glance it is not apparent how to organize the free blocks into a single space-efficient structure that will allow the time-efficient implementation of either policy.

Now consider a priority search tree (of either kind, but one would probably want to use the radix kind), where the x dimension is an encoding of [free block length, free block address] for uniqueness (see § 2) and the y dimension is the free block address. Best-fit can be implemented using only the search tree part of the radix search tree in the obvious way. A first-fit implementation uses **MinYInXRange** on an x range of [**neededBlockSize** .. **largestPossibleBlockSize**]. Each of these operations, as well as insertion or removal of free blocks in the structure requires at most logarithmic time. The extra space requirements are minimal: a radix priority search tree for this purpose requires that each free block contain two pointers and a field to hold the length of the free block.

A result similar to this is attributed to McCreight in [2], and that earlier data structure bears a striking resemblance to a priority search tree. The difference between them is that in the earlier structure, the pair whose y -value is minimal in a subtree not only appears at the root of the subtree, but might also be repeated on a spine all the way down to a leaf of the subtree. In a priority search tree, as in a proper priority queue, pairs are not repeated. The effect of this is that the earlier structure and the priority search tree perform equally well (within constant factors) for all operations except **EnumerateRectangle**, but the $O(\log n + s)$ time bound for **EnumerateRectangle** cannot be attained with the earlier structure. This is because in the earlier structure

enumerations can encounter the same pairs over and over again, often enough to ruin linearity in s .

5. Open questions. The question that led me from tile trees [3] to priority search trees remains unanswered. I still do not know whether it is possible, for an arbitrary set of n aligned rectangles, to enumerate all s pairs that totally contain one another in linear space and time $O(n \log n + s)$. The methods in this paper allow one to determine containment on three edges, but alas, three edges do not a rectangle make.

Priority search trees are one small step closer to the ultimate goal of general two-dimensional range searching in linear space and logarithmic worst-case time. Is that ultimate goal attainable? If not, or if we cannot discover how, are there further small but useful steps?

Priority search trees are an interesting case of two data structures (a search tree and a priority queue) in symbiosis, defined as “the living together of two dissimilar organisms, especially when this association is mutually beneficial.” Are there other pairs of data structures that also benefit from symbiosis?

Acknowledgments. The ideas leading to this paper have developed over several years, and I have many people to thank. I especially thank Jon Bentley for pointing out the gaping hole in my original attempt to solve this problem, and Howard Sturgis for pointing out that within the rococo walls of one of my intermediate formulations lay the elegant machinery of § 2. Jan van Leeuwen first made me aware of Olivie’s balanced trees with guaranteed constant rotations per update, thereby saving the reader a very tortuous and otherwise useless new data structure in § 3, and Bob Tarjan recently observed that 2–3–4 trees can have the same property. I also thank John Warnock, Leo Guibas, Bob Sedgwick, Mark Brown, Jurg Nievergelt, Herbert Edelsbrunner, Jean Vuillemin, the referees, and others for discussions that led to the present algorithms and presentation.

Appendix A. Pascal procedures for radix priority search trees.

```

PROCEDURE InsertPair(VAR t: RPSTPtr; newPr: Pair;
  lowerX: KeyRange; upperX: KeyBound);
VAR
  p: Pair;
  middleX: KeyRange;
BEGIN
  IF t = NIL THEN
    BEGIN
      NEW(t); (* add a new leaf node *)
      t.p := newPr;
      t.left := NIL;
      t.right := NIL;
    END
  ELSE IF t.p.x <> newPr.x (* assumes unique x values *)
    THEN
      BEGIN
        IF newPr.y < t.p.y THEN (* new pair beats existing one *)
          BEGIN p := t.p; t.p := newPr END
          (* exchange new/existing *)
        ELSE p := newPr;
          middleX := (lowerX+upperX) DIV 2;
          IF p.x < middleX
            THEN InsertPair(t.left, p, lowerX, middleX)
            ELSE InsertPair(t.right, p, middleX, upperX);
          END;
        (* ELSE this pair already present, so don't insert it *)
      END;
  END;
END; (* of InsertPair *)

```

```

PROCEDURE DeletePair(VAR t: RPSTPtr; oldPr: Pair;
  lowerX: KeyRange; upperX: KeyBound);
VAR
  middleX: KeyRange;
BEGIN
  IF t <> NIL THEN
    BEGIN
      IF t↑.p.x = oldPr.x (* assumes unique x values *)
      THEN
        BEGIN (* have located pair to delete *)
          IF t↑.left <> NIL THEN
            BEGIN
              IF t↑.right <> NIL THEN
                BEGIN (* node has both left and right subtrees *)
                  IF t↑.left↑.p.y < t↑.right↑.p.y THEN
                    BEGIN (* left beats right *)
                      t↑.p := t↑.left↑.p;
                      DeletePair(t↑.left, t↑.p, lowerX, upperX);
                    END
                  ELSE
                    BEGIN (* right beats left *)
                      t↑.p := t↑.right↑.p;
                      DeletePair(t↑.right, t↑.p, lowerX, upperX);
                    END;
                  END
                ELSE
                  BEGIN (* node has only left subtree *)
                    t↑.p := t↑.left↑.p;
                    DeletePair(t↑.left, t↑.p, lowerX, upperX);
                  END;
                END
              ELSE
                BEGIN
                  BEGIN
                    IF t↑.right <> NIL THEN
                      BEGIN (* node has only right subtree *)
                        t↑.p := t↑.right↑.p;
                        DeletePair(t↑.right, t↑.p, lowerX, upperX);
                      END
                    ELSE
                      BEGIN (* node has no subtrees *)
                        DISPOSE(t);
                        t := NIL;
                      END;
                    END;
                  END;
                END
              ELSE
                BEGIN (* pair to delete is in a subtree *)
                  middleX := (lowerX+upperX) DIV 2;
                  IF oldPr.x < middleX
                    THEN DeletePair(t↑.left, oldPr, lowerX, middleX)
                    ELSE DeletePair(t↑.right, oldPr, middleX, upperX);
                END;
            END;
          END;
        END;
      (* ELSE this pair wasn't in the tree so it can't be deleted *)
    END; (* of DeletePair *)
  
```

```

TYPE CondPair = RECORD
  valid: BOOLEAN;
  p: Pair;
END;

FUNCTION MinXInRectangle(t: RPSTPtr; x0, x1, y1: KeyRange;
  lowerX: KeyRange; upperX: KeyBound): CondPair;
VAR
  c: CondPair;
  middleX: KeyRange;
BEGIN
  IF t <> NIL THEN
    BEGIN
      IF t↑.p.y > y1 THEN
        (* No nodes in this subtree lie in the search
          rectangle, because they all have y-values that are
          too large. *)
          c.valid := FALSE
        ELSE
          BEGIN
            middleX := (lowerX+upperX) DIV 2;

            IF x0 < middleX THEN
              (* The answer can only lie in the left subtree
                if some point in the search rectangle
                could lie in the left subtree. *)
                c := MinXInRectangle(t↑.left, x0, x1, y1,
                  lowerX, middleX)
              ELSE c.valid := FALSE;

              IF (NOT c.valid) AND (middleX <= x1) THEN
                (* The answer can only lie in the right subtree
                  if no point in the left subtree lies in the search
                  rectangle, but some point in the search rectangle
                  could lie in the right subtree. *)
                c := MinXInRectangle(t↑.right, x0, x1, y1,
                  middleX, upperX);

                IF (x0 <= t↑.p.x) AND (t↑.p.x <= x1) AND
                  ((NOT c.valid) OR (t↑.p.x < c.p.x)) THEN
                    (* t↑.p is best of all in the search rectangle *)
                    BEGIN
                      c.valid := TRUE;
                      c.p := t↑.p;
                    END;
                END
              END
            ELSE c.valid := FALSE; (* empty subtree *)
            MinXInRectangle := c;
          END;
        END;
      END;
    END;
  END; (* of MinXInRectangle *)

```



```

FUNCTION MaxXInRectangle(t: RPSTPtr; x0, x1, y1: KeyRange;
  lowerX: KeyRange; upperX: KeyBound): CondPair;
VAR
  c: CondPair;
  middleX: KeyRange;
BEGIN
  IF t <> NIL THEN
    BEGIN
      IF t↑.p.y > y1 THEN
        (* No nodes in this subtree lie in the search
          rectangle, because they all have y-values that are
          too large. *)
          c.valid := FALSE
        ELSE
          BEGIN
            middleX := (lowerX+upperX) DIV 2;

            IF middleX < x1 THEN
              (* The answer can only lie in the right subtree
                if some point in the search rectangle
                could lie in the right subtree. *)
                c := MaxXInRectangle(t↑.right, x0, x1, y1,
                  middleX, upperX)
              ELSE c.valid := FALSE;

              IF (NOT c.valid) AND (x0 <= middleX) THEN
                (* The answer can only lie in the left subtree
                  if no point in the right subtree lies in the search
                  rectangle, but some point in the search rectangle
                  could lie in the left subtree. *)
                c := MaxXInRectangle(t↑.left, x0, x1, y1,
                  lowerX, middleX);

                IF (x0 <= t↑.p.x) AND (t↑.p.x <= x1) AND
                  ((NOT c.valid) OR (c.p.x < t↑.p.x)) THEN
                  (* t↑.p is best of all in the search rectangle *)
                  BEGIN
                    c.valid := TRUE;
                    c.p := t↑.p;
                  END;
                END
              END
            ELSE c.valid := FALSE; (* empty subtree *)
            MaxXInRectangle := c;
          END; (* of MaxXInRectangle *)
        
```

```

FUNCTION MinYInXRange(t: RPSTPtr; x0, x1: KeyRange;
  lowerX: KeyRange; upperX: KeyBound): CondPair;
VAR
  c, cRight: CondPair;
  middleX: KeyRange;
BEGIN
  IF t <> NIL THEN
    IF (x0 <= t↑.p.x) AND (t↑.p.x <= x1) THEN
      (* This node's p pair lies in the x range, and it must
      be the min y in its subtree because of the priority
      queue invariant on y. *)
      BEGIN
        c.valid := TRUE;
        c.p := t↑.p;
      END
    ELSE
      BEGIN
        middleX := (lowerX+upperX) DIV 2;

        IF x0 < middleX THEN c :=
          MinYInXRange(t↑.left, x0, x1, lowerX, middleX)
        ELSE c.valid := FALSE;

        IF middleX <= x1 THEN cRight :=
          MinYInXRange(t↑.right, x0, x1, middleX, upperX)
        ELSE cRight.valid := FALSE;

        IF NOT c.valid OR
          (cRight.valid AND (cRight.p.y < c.p.y)) THEN
          c := cRight;
        END
      END
    ELSE c.valid := FALSE; (* empty subtree *)
  MinYInXRange := c;
END; (* of MinYInXRange *)

FUNCTION EnumerateRectangle(t: RPSTPtr; x0, x1, y1: KeyRange;
  FUNCTION Report(Pair): BOOLEAN;
  lowerX: KeyRange; upperX: KeyBound): BOOLEAN;
VAR
  continue: BOOLEAN;
  middleX: KeyRange;
BEGIN
  IF t <> NIL THEN
    IF t↑.p.y <= y1 THEN
      BEGIN (* node passes y test *)
        IF (x0 <= t↑.p.x) AND (t↑.p.x <= x1) THEN
          continue := Report(t↑.p)
        ELSE continue := TRUE;
        middleX := (lowerX+upperX) DIV 2;
        IF continue AND (x0 < middleX)
          THEN
          continue := EnumerateRectangle(t↑.left, x0, x1, y1,
            Report, lowerX, middleX);
        IF continue AND (middleX <= x1)
          THEN
          continue := EnumerateRectangle(t↑.right, x0, x1, y1,
            Report, middleX, upperX);
        EnumerateRectangle := continue;
      END
    ELSE EnumerateRectangle := TRUE (* node fails y test *)
  ELSE EnumerateRectangle := TRUE; (* empty subtree *)
END; (* of EnumerateRectangle *)

```

Appendix B. Pascal procedures for balanced priority search trees.

```

PROCEDURE BalancedInsertPair(VAR t: BPSTPtr; newPr: Pair;
  useAsP: BOOLEAN);
  BEGIN (* Top-level call has useAsP = TRUE *)
  IF t = NIL THEN
    BEGIN (* put newPr in the q field of a new leaf node *)
    NEW(t);
    t↑.q := newPr;
    t↑.left := NIL;
    t↑.right := NIL;
    t↑.validP := FALSE;
    t↑.duplQ := NOT useAsP;
    t↑.balance := leafBalance
      (* depends on tree family *)
    END
  ELSE
    BEGIN
    IF useAsP AND ((NOT t↑.validP) OR (newPr.y < t↑.p.y)) THEN
      BEGIN (* newPr belongs in t↑.p *)
      DisposeP(t↑);
      t↑.p := newPr;
      t↑.validP := TRUE;
      useAsP := FALSE;
      END;
    IF newPr.x < t↑.q.x
      THEN BalancedInsertPair(t↑.left, newPr, useAsP)
      ELSE BalancedInsertPair(t↑.right, newPr, useAsP);
    AdjustBalanceForInsert(t);
      (* implementation varies by tree family *)
    END;
  END; (* of BalancedInsertPair *)

PROCEDURE BalancedDeletePair(VAR t: BPSTPtr; oldPr: Pair);

TYPE
  ExtractedPair = RECORD
    q: Pair;
    duplAsP: BOOLEAN (* q appears as p field higher in tree *)
  END;

VAR
  n: ExtractedPair;
  d: BPSTPtr;

FUNCTION ExtractMaxQX(VAR t: BPSTPtr): ExtractedPair;
  VAR
    d: BPSTPtr;
  BEGIN (* extract the pair with maximal q.x in t *)
  IF t↑.right = NIL THEN
    BEGIN
    ExtractMaxQX.q := t↑.q;
    ExtractMaxQX.duplAsP := t↑.duplQ;
    DisposeP(t↑);
    d := t;
    t := t↑.left;
    DISPOSE(d);
    END
  ELSE

```

```

BEGIN
  ExtractMaxQX := ExtractMaxQX(t↑.right);
  IF ExtractMaxQX.duplAsP AND t↑.validP
    AND (t↑.p = ExtractMaxQX.q) THEN
    BEGIN
      ExtractMaxQX.duplAsP := FALSE;
      ExtractP(t↑);
      (* re-fill invalidated p field if possible *)
    END;
  AdjustBalanceForNeighborExtract(t);
  (* implementation varies by tree family *)
END
END; (* of ExtractMaxQX *)

BEGIN
  IF t <> NIL THEN
    BEGIN
      IF t↑.q = oldPr THEN
        (* have located node whose .q field is oldPr *)
        BEGIN
          DisposeP(t↑);
          IF (t↑.left = NIL) OR (t↑.right = NIL) THEN
            (* t↑ has at most one subtree and can
              be bypassed *)
            BEGIN
              d := t;
              IF t↑.left = NIL
                THEN t := t↑.right
                ELSE t := t↑.left;
              DISPOSE(d);
            END
          END
        ELSE

          (* t↑ has both subtrees. We must find
            a neighboring pair n.q that can
            replace t↑.q without violating x-order. *)
          BEGIN
            n := ExtractMaxQX(t↑.left);
            t↑.q := n.q;
            t↑.duplQ := n.duplAsP;
            ExtractP(t↑);
            (* re-fill invalidated p field if possible *)
          END;
        END
      ELSE
        BEGIN (* oldPr must be a .q field in a subtree *)
          IF oldPr.x < t↑.q.x
            THEN BalancedDeletePair(t↑.left, oldPr)
            ELSE BalancedDeletePair(t↑.right, oldPr);
          IF t↑.validP AND (t↑.p = oldPr) THEN
            ExtractP(t↑);
            (* re-fill invalidated p field if possible *)
          END;
          AdjustBalanceForDelete(t↑);
          (* implementation varies by tree family *)
        END
      END
    END
  (* ELSE this pair wasn't in the tree so it can't be deleted *);
END; (* of BalancedDeletePair *)

```

```

PROCEDURE RotateRight(VAR t: BPSTPtr);
  VAR
    e, c: BPSTPtr;
  BEGIN (* implements the rotation in Figure 2 *)
    e := t;
    DisposeP(e↑);
    c := e↑.left;
    DisposeP(c↑);
    e↑.left := c↑.right;
    ExtractP(e↑);
    c↑.right := e;
    ExtractP(c↑);
    AdjustBalanceForRotateRight(c);
    (* implementation varies by tree family *)
    t := c;
  END; (* of RotateRight *)

```

```

PROCEDURE DisposeP(VAR t: BPST);
  BEGIN (* DisposeP can cause a temporary violation of the second
    structural invariant, leaving a node in the middle of the tree
    with an invalid p field while one of its children has a
    valid p field. This violation is usually repaired by a
    subsequent invocation of ExtractP *)
  IF t.validP THEN
    BEGIN
      IF t.p.x < t.q.x THEN
        BEGIN (* dispose into left subtree *)
          IF t.p = t.left↑.q THEN
            (* maintains third invariant, depends on
            uniqueness of pairs in t *)
            t.left↑.duplQ := FALSE
          ELSE
            BEGIN
              DisposeP(t.left↑);
              t.left↑.p := t.p;
              t.left↑.validP := TRUE;
            END
          END
        ELSE
          BEGIN (* dispose into right subtree *)
            IF t.p = t.right↑.q THEN
              (* maintains third invariant, depends on
              uniqueness of pairs in t *)
              t.right↑.duplQ := FALSE
            ELSE
              BEGIN
                DisposeP(t.right↑);
                t.right↑.p := t.p;
                t.right↑.validP := TRUE;
              END
            END;
            t.validP := FALSE;
          END
        (* ELSE no p field to dispose *);
      END; (* of DisposeP *)

```

```

PROCEDURE ExtractP(VAR t: BPST);
  CONST Worst = LastKey+1;
  VAR
    leftY,rightY: FirstKey..Worst;
  BEGIN
    leftY := Worst;
    IF t.left <> NIL THEN
      BEGIN
        IF t.left↑.validP THEN leftY := t.left↑.p.y;
        IF NOT t.left↑.duplQ THEN leftY := MIN(leftY, t.left↑.q.y);
        END;
      rightY := Worst;
      IF t.right <> NIL THEN
        BEGIN
          IF t.right↑.validP THEN rightY := t.right↑.p.y;
          IF NOT t.right↑.duplQ THEN rightY := MIN(rightY, t.right↑.q.y);
          END;
        IF leftY < rightY THEN
          BEGIN (* best is left *)
            IF t.left↑.validP AND (leftY = t.left↑.p.y) THEN
              BEGIN (* steal his p field *)
                t.p := t.left↑.p;
                ExtractP(t.left↑);
              END
            ELSE
              BEGIN (* his q field is unduplicated and better,
                so duplicate it *)
                t.p := t.left↑.q;
                t.left↑.duplQ := TRUE;
              END;
            t.validP := TRUE;
          END
        ELSE IF rightY <> Worst THEN
          BEGIN (* best is right *)
            IF t.right↑.validP AND (rightY = t.right↑.p.y) THEN
              BEGIN (* steal his p field *)
                t.p := t.right↑.p;
                ExtractP(t.right↑);
              END
            ELSE
              BEGIN (* his q field is unduplicated and better,
                so duplicate it *)
                t.p := t.right↑.q;
                t.right↑.duplQ := TRUE;
              END;
            t.validP := TRUE;
          END
        ELSE t.validP := FALSE; (* no candidates *)
      END; (* of ExtractP *)

```

REFERENCES

- [1] D. E. KNUTH, *The Art of Computer Programming, Vol. 3, Sorting and Searching*, Addison-Wesley, Reading, MA, 1973, pp. 142-145.
- [2] H. J. OLIVIE, *A new class of balanced search trees; half-balanced binary search trees*, RAIRO Theor. Inform., 16 (1982), pp. 51-71.
- [3] E. MCCREIGHT, *Efficient algorithms for enumerating intersecting intervals and rectangles*, Xerox Palo Alto Research Center Technical Report CSL-80-9, Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA, 1980.
- [4] L. GUIBAS AND J. SAXE, Private communication, 1980.
- [5] H. EDELSBRUNNER, *Dynamic rectangle intersection searching*, Report 47, Information Processing Institute, Technical University of Graz, Graz, Austria, February, 1980.
- [6] ———, *Dynamic data structures for orthogonal intersection queries*, Report 59, Information Processing Institute, Technical University of Graz, Graz, Austria, October, 1980.
- [7] M. I. SHAMOS AND D. HOEY, *Geometric intersection problems*, 17th Annual IEEE Symposium on Foundations of Computer Science, 1975, pp. 208-215.
- [8] H. EDELSBRUNNER, *A time- and space-optimal solution for the planar all intersecting rectangles problem*, Report 50, Information Processing Institute, Technical University of Graz, Graz, Austria, April, 1980.
- [9] J. BENTLEY, *Priority queues with range restrictions*, Bulletin of the European Association of Theoretical Computer Science, #9, H. Maurer, ed., Technical University of Graz, Graz, Austria, October, 1979, pp. 7-8.
- [10] J. L. BENTLEY AND D. WOOD, *An optimal worst-case algorithm for reporting intersections of rectangles*, IEEE Trans. Comp., C-29 (1980), pp. 571-577.
- [11] R. P. BRENT, *Efficient implementation of the first-fit strategy for dynamic storage allocation*, TR-CS-81-05, Dept. Computer Science, Australian National University, Canberra, ACT 2600, Australia, 1981, Australian Computer Science Communications, to appear.
- [12] J. NIEVERGELT AND F. P. PREPARATA, *Plane-sweep algorithms for interesting geometric figures*, Technical Report in preparation, Institut für Informatik, ETH, Zurich.
- [13] J. VUILLEMIN, *A unifying look at data structures*, Comm. ACM, 23 (1980), pp. 229-239.
- [14] R. BAYER, *Symmetric binary B-trees: data structure and maintenance algorithms*, Acta Inform., 1 (1972), pp. 290-306.
- [15] L. J. GUIBAS AND R. SEDGEWICK, *A dichromatic framework for balanced trees*, 19th Annual IEEE Symposium on Foundations of Computer Science, IEEE publication 78CH1397-9, 1978, pp. 8-21.
- [16] R. TARJAN, Private communication, September, 1982.

ON THE COMPLEXITY OF MAINTAINING PARTIAL SUMS*

ANDREW C. YAO†

Abstract. Let $F = \{(r_i, s_i) | 0 \leq i < n\}$ be a file of n records, where r_i are d -dimensional vectors and s_i are elements of a commutative semigroup S . We are interested in the partial sum problem, in which queries of the form “ $\sum_{r_i \leq a} s_i = ?$ ” are to be answered. A space-time tradeoff $t = \Omega(\log n / \log(m \log n/n))$ is established for storing a static two-dimensional file. It will also be shown that, for the one-dimensional problem, any dynamic algorithm must have a worst-case time $\Omega(n \log n / \log \log n)$ in processing a sequence $O(n)$ INSERT and QUERY instructions.

Key words. complexity, lower bound, partial sum, range query, semigroup, space-time tradeoff

1. Introduction. Consider a database that contains a collection of records, each with a key and a number of data fields. Given a range query, which is specified by a set of constraints on keys, the database system is expected to return the set of records, or a certain function of the set of records, whose keys satisfy all the constraints. For example, consider a geographic database in which the record for a city contains among other items its location (as the key) and its population (as a data field). A range query may ask for the total population of cities whose locations are within r miles of a certain point; another possible range query may ask for the number of cities in a certain region. There is an extensive literature on efficient algorithms for handling various types of range queries (see, e.g., Bentley and Mauer [1], Lueker [8], Rivest [9], Willard [10], [11]). As in all data structure problems, the optimality question (i.e., whether one has found the best possible solution) is much harder to answer. Recently, an interesting model was developed by Fredman [3]–[6] to discuss the inherent complexity of some range query problems. Several elegant results were obtained which offered insight into questions such as “why circular range queries seem more difficult to handle than orthogonal range queries,” and “why a sequence of $O(n)$ instructions for d -dimensional orthogonal range queries takes $O(n(\log n)^d)$ time to process.”

In this paper we consider range queries of a special type, the *d-dimensional partial sum queries*, in the framework of Fredman's. The keys are d -dimensional vectors, and a query selects records whose keys are dominated componentwise by a certain vector. We will establish, for $d=2$, a (storage) space–(retrieval) time tradeoff $t = \Omega(\log n / \log(m \log n/n))$ for a static collection of n records. It will also be shown that, for the one-dimensional case, any dynamic algorithm must use $\Omega(n \log n / \log \log n)$ time to process a sequence of $O(n)$ insertions and queries in the worst case.

We remark that the space and time requirements for range queries have received much attention [1], [2], [8], [9], [10], [11]. A discussion of space–time tradeoff constraints for other types of range queries within the present framework can be found in Yao [12]. We also mention that the complexity of the one-dimensional dynamic partial sum query was also studied by Fredman [5] in a somewhat different model; his results are incomparable to ours. Bentley and Shamos [2] discussed some algorithms for partial sum queries, where the problem was called the ECDF problem (for Empirical Cumulative Distribution Function).

* Received by the editors March 11, 1982, and in revised form October 11, 1983. This research was done while the author was visiting the Computer Science Department, IBM San Jose Research Center, 5600 Cottle Road, San Jose, California 95193. This work was supported in part by the National Science Foundation under grant MCS-77-05313-A01.

† Computer Science Department, Stanford University, Stanford, California 94305.

2. Model and results. Let S be a commutative semigroup with an addition operation “+”. Let $d \geq 1$ be an integer, and let Z^d be the set of all d -tuples of integers. A record (\mathbf{r}, s) is a pair of key $\mathbf{r} \in Z^d$ and datum $s \in S$. A file F is a finite collection of records. A partial sum query $\text{QUERY}(\mathbf{a})$ is specified by a vector $\mathbf{a} \in Z^d$. For a file $F = \{(\mathbf{r}_i, s_i) | 0 \leq i < n\}$, the response to $\text{QUERY}(\mathbf{a})$ is defined to be $\text{resp}(\mathbf{a}; F) = \sum_{\mathbf{r}_i \leq \mathbf{a}} s_i$, where $\mathbf{r}_i \leq \mathbf{a}$ means componentwise inequalities; we agree that $\text{resp}(\mathbf{a}; F) = \emptyset$ if there exists no $\mathbf{r}_i \leq \mathbf{a}$.

We are interested in the question “how efficiently can one implement partial sum queries?”. Two types of problems will be considered: *the static problem* in which the file F is fixed, and *the dynamic problem* in which insertions of new records may take place.

The static problem. Let $F = \{(\mathbf{r}_i, s_i) | 0 \leq i < n\}$ be a given file. We wish to store the information in a way that partial sum queries can be answered quickly. Call an expression $\sum_i \lambda_i s_i$ a *positive linear function* if λ_i are nonnegative integers. A *storage scheme* \mathcal{S} for F is a family of positive linear functions $\{z_1, z_2, \dots, z_m\}$ such that, for any $\mathbf{a} \in Z^d$, there exists an identity

$$(1) \quad \text{resp}(\mathbf{a}; F) = \sum_{j \in V} \mu_j z_j$$

valid for all values of $s_i \in S$, for some integers $\mu_j \geq 0$ and $V \subseteq \{1, 2, \dots, m\}$. We will interpret the right-hand side of (1) as \emptyset if $V = \emptyset$. Let $t_F(\mathbf{a}; \mathcal{S})$ be the minimum $|V|$ for which $\text{resp}(\mathbf{a}; F)$ can be written in the form of (1); let $t_F(\mathcal{S}) = \max_{\mathbf{a}} t_F(\mathbf{a}; \mathcal{S})$. We will omit the subscripts F in $t_F(\mathbf{a}; \mathcal{S})$ and $t_F(\mathcal{S})$ when there is no danger of ambiguity. We say that \mathcal{S} uses *space* m and *time* $t(\mathcal{S})$.

We emphasize that the keys \mathbf{r}_i in F are considered fixed, and identity (1) need only be valid for this set of keys. However, the identity must be true for all possible values of $s_i \in S$.

All the above discussions are dependent on the semigroup S under consideration. A storage scheme \mathcal{S} for $F = \{(\mathbf{r}_i, s_i) | 0 \leq i < n\}$ with $s_i \in S$ may not be a storage scheme for $F' = \{(\mathbf{r}_i, s'_i) | 0 \leq i < n\}$ with $s'_i \in S'$, as will be clear from the next two examples.

Example 1. Let $d = 1$, and S be the set of real numbers under the ordinary addition “+”. Assume $f = \{(i, s_i) | 0 \leq i < n\}$. Then the collection $\mathcal{S} = \{z_1, z_2, \dots, z_n\}$ with $z_i = \sum_{0 \leq j < i} s_j$ is a storage scheme. It is easy to verify that both the space $m = n$ and time $t(\mathcal{S}) = 1$ are minimum for any storage scheme.

Example 2. Let $d \geq 1$ be arbitrary, and $S = \{b\}$ with an addition operation defined by $b + b = b$. For any file $F = \{(\mathbf{r}_i, s_i) | 0 \leq i < n\}$ with $s_i \in S$, the collection $\mathcal{S} = \{z\}$ with $z = s_0$ is always a storage scheme.

The dynamic problem. Initially, the file F is empty. We wish to process a sequence σ of instructions $\alpha_1, \alpha_2, \dots, \alpha_p$, where each α_i is either of the two types: $\text{INSERT}(\mathbf{r}, s)$, $\text{QUERY}(\mathbf{a})$. The instruction $\text{INSERT}(\mathbf{r}, s)$ requires that $F \leftarrow F \cup \{(\mathbf{r}, s)\}$, and $\text{QUERY}(\mathbf{a})$ asks that $\text{resp}(\mathbf{a}; F)$ be returned for the current F .

In the model we are considering, there is an infinite array of variables z_1, z_2, \dots . An algorithm \mathcal{A} specifies how the instructions are to be implemented using these variables. To process an instruction $\alpha_i = \text{INSERT}(\mathbf{r}, s)$, one carries out a sequence of operations $\beta_1, \beta_2, \dots, \beta_b$, where each β_u is either of the form $z_j \leftarrow s$ or $z_j \leftarrow \lambda z_k + \lambda' z_{k'}$ ($\lambda, \lambda' \geq 0$ integers). To process an instruction $\alpha_i = \text{QUERY}(\mathbf{a})$, one carries out $\beta_1, \beta_2, \dots, \beta_b$, where β_u is of the form $z_j \leftarrow \lambda z_k + \lambda' z_{k'}$ ($\lambda, \lambda' \geq 0$ integers), return (z_j) , or return (\emptyset) . The algorithms are fully adaptive: the number of operations and the choice of the operations used to process an instruction can depend on the key

values of the queries that have been processed so far. Note that QUERY (**a**) takes at least $t_F(\mathbf{a}; \mathcal{S})$ operations to process, where F is the current file and \mathcal{S} is the collection of contents of the variables used (one operation for return (z_j), and $t_F(\mathbf{a}; \mathcal{S}) - 1$ to get z_j). The cost $C(\sigma; \mathcal{A})$ is the total number of operations used to process the sequence σ .

As we discussed in Example 2, the problems may become trivial due to the trivial nature of the semigroup S . We now define a class of semigroups for which the problem is nontrivial. A commutative semigroup S is said to be *faithful* if, for every $T_1, T_2 \subseteq \{1, 2, \dots, n\}$ and every integer $\delta_i, \delta'_j > 0$,

$$\sum_{i \in T_1} \delta_i s_i = \sum_{j \in T_2} \delta'_j s_j$$

cannot be an identity for all $s_1, s_2, \dots, s_n \in S$ unless $T_1 = T_2$. It is easy to verify that the set of real numbers under the ordinary addition is faithful; so is the set of real numbers under $\max\{x, y\}$ as the “addition” operation; so is the set $S = \{0, 1\}$ under the logical “OR” as its addition.

On the other hand the semigroup $\{0, 1\}$ with modulo 2 addition is not faithful.

We summarize below the main results of this paper. (All the logarithms will be in base 2 unless explicitly indicated otherwise.)

THEOREM 1. *There exists a constant $\gamma > 0$ such that the following is true. Consider the static partial sum problem for $d = 2$ and a faithful commutative semigroup S . For each $n > 2$, there exists a file F of n records such that any storage scheme using space $m \geq n$ must have a time $t \geq \gamma \cdot \log n / \log(m \log n / n)$.*

THEOREM 2. *There exists a constant $\gamma' > 0$ such that the following is true. Consider the dynamic partial sum problem for $d = 1$ and a faithful commutative semigroup S . For any algorithm \mathcal{A} , there exists for each n a sequence σ of $2n$ INSERT and QUERY instructions such that $C(\sigma; \mathcal{A}) \geq \gamma' \cdot n \log n / \log \log n$.*

THEOREM 3. *Let $d > 1$ be fixed. Suppose there is an algorithm for the dynamic $(d - 1)$ -dimensional partial sum problem, with an $O(\nu(n))$ processing time per instruction (INSERT or QUERY) when the current file contains n records. Then, for any given file of n records with d -dimensional keys, there exists a storage scheme that uses space $O(n\nu(n))$ and time $O(\nu(n))$.*

We remark that Theorem 3 is valid without the faithfulness assumption. Also we wish to point out that, when S is a group and the subtraction operation is allowed in the implementation, the response to an orthogonal query $\sum_{\mathbf{a} < \mathbf{r}_i \leq \mathbf{b}} s_i$ can be expressed as a linear combination of partial sums (see Bentley and Shamos [2] for this observation). For example, when $d = 2$,

$$\sum_{\mathbf{a} < \mathbf{r}_i \leq \mathbf{b}} s_i = \sum_{\mathbf{r}_i \leq \mathbf{b}} s_i + \sum_{\mathbf{r}_i \leq \mathbf{a}} s_i - \sum_{\mathbf{r}_i \leq (b_1, a_2)} s_i - \sum_{\mathbf{r}_i \leq (a_1, b_2)} s_i,$$

where $\mathbf{a} = (a_1, a_2)$ and $\mathbf{b} = (b_1, b_2)$. Theorem 3 is then also true for orthogonal queries in this case.

3. Two-dimensional static-partial sums. Let $n = 2^k$ be a power of 2. For each integer $0 \leq j < n$, let j^R be the integer with binary representation $\beta_0 \beta_1 \beta_2 \dots \beta_{k-1}$, where $\beta_{k-1} \beta_{k-2} \dots \beta_1 \beta_0$ is the k -bit binary representation of j . Let B_n denote the set $\{\mathbf{b}_j | 0 \leq j < n\}$, where $\mathbf{b}_j = (j, j^R)$. Consider the file $F_n = \{(\mathbf{b}_j, s_j) | \mathbf{b}_j \in B_n\}$, where s_j are elements of a faithful commutative semigroup S . The aim of this section is to prove the following theorem.

THEOREM 4. *There exists a positive constant γ (independent of S) such that the following is true. Let $n = 2^k$ where $k > 3$ is an integer. Let \mathcal{S} be any storage scheme using*

space m for file F_n , where $m \geq n$. Then there exist at least $n/4$ $l \in \{0, 1, 2, \dots, n-1\}$ with the property that, for some $\mathbf{a}_l = (l, h(l))$,

$$t(\mathbf{a}_l; \mathcal{S}) \geq \gamma \cdot \frac{\log n}{\log\left(\frac{m}{n} \log n\right)}.$$

It is perhaps curious to note that the set B_n , when viewed as a permutation, arose in another context of space-time tradeoff [7].

Theorem 4 implies Theorem 1 for the case when n is a power of 2. Clearly, this then implies Theorem 1 for all n . We will also use Theorem 4 in § 4 in the proof of Theorem 2.

3.1. Reductions. We prove Theorem 4 in a slightly different form. For each $\mathbf{a} \in Z^2$, let $A(\mathbf{a}) = B_n \cap \{\mathbf{b} | \mathbf{b} \leq \mathbf{a}\}$. A structure \mathcal{T} for B_n is a family of subsets $\{T_1, T_2, \dots, T_m\}$, where $T_i \subseteq B_n$, such that every $A(\mathbf{a})$ can be written as $\cup_{i \in V(\mathbf{a})} T_i$ for some $V(\mathbf{a}) \subseteq \{1, 2, \dots, m\}$; let $t'(\mathbf{a}; \mathcal{T})$ be the minimum l for which $V(\mathbf{a})$ of size l exists. In particular, $t'(\mathbf{a}; \mathcal{T}) = 0$ if $A(\mathbf{a}) = \emptyset$.

THEOREM 4'. *There exists a positive constant γ such that the following is true. Let $m \geq n = 2^k$ where $k > 3$ is an integer. Let \mathcal{T} be a structure for B_n with $|\mathcal{T}| = m$. Then there exist at least $n/4$ $l \in \{0, 1, 2, \dots, n-1\}$ such that there exist $h(l)$ with*

$$t'((l, h(l)); \mathcal{T}) \geq \gamma \cdot \frac{\log n}{\log\left(\frac{m}{n} \log n\right)}.$$

We first show that Theorem 4' implies Theorem 4. Given any storage scheme \mathcal{S} for F_n , we will construct a structure \mathcal{T} for B_n such that $|\mathcal{T}| = |\mathcal{S}|$, and $t'(\mathbf{a}; \mathcal{T}) \leq t(\mathbf{a}; \mathcal{S})$ for any $\mathbf{a} \in Z^2$. This clearly will demonstrate that Theorem 4' implies Theorem 4.

For any positive linear function $z = \sum_j \lambda_j s_j$, let $W(z) = \{(j, j^R) | \lambda_j > 0\} \subseteq B_n$. For a given storage scheme $\mathcal{S} = \{z_1, z_2, \dots, z_m\}$, define $\mathcal{T} = \{W(z_i) | 1 \leq i \leq m\}$. For any $\mathbf{a} \in Z^2$, write

$$\text{resp}(\mathbf{a}; F_n) = \sum_{i \in V(\mathbf{a})} \delta_i z_i$$

where $\delta_i > 0$ and $|V(\mathbf{a})| = t(\mathbf{a}; \mathcal{S})$. Using the definition of $\text{resp}(\mathbf{a}, F_n)$, we can write the above equation as

$$\sum_{(j, j^R) \in A(\mathbf{a})} s_j = \sum_{i \in V(\mathbf{a})} \delta_i z_i = \sum_j \lambda_j s_j,$$

for some λ_j , where $\lambda_j > 0$ iff $j \in W(z_i)$ for some $i \in V(\mathbf{a})$. This means

$$A(\mathbf{a}) = \cup_{i \in V(\mathbf{a})} W(z_i)$$

because of the faithfulness of S . It follows that $t'(\mathbf{a}; \mathcal{T}) \leq t(\mathbf{a}; \mathcal{S})$.

The remainder of § 3 is devoted to the proof of Theorem 4'. We will use the symbol μ to stand for m/n , and, without loss of generality, assume that $\mu \geq 2$. This latter assumption will be used in the proof of Lemma 3 below. We organize our proof into three lemmas.

Note that the density of the points of B_n in the $n \times n$ square is $1/n$. The first lemma asserts that these points are uniformly distributed in a certain sense.

LEMMA 1. *Let $[a, b) \times (c, d) \subseteq [0, n] \times [0, n]$, where $a < b, c < d$ are real numbers. If $(b-a)(d-c) \geq 8n$, then there exists at least one point of B_n in $[a, b) \times (c, d)$.*

Proof. Clearly, $b - a \geq 4$. Let r be the unique integer satisfying $(b - a)/4 \leq 2^r < (b - a)/2$. If we partition the set $\{0, 1, 2, \dots, n - 1\}$ into 2^{k-r} consecutive equal parts, one of them must be entirely contained in $[a, b]$; suppose it is $Y = \{j2^r + i \mid 0 \leq i < 2^r\}$. Let $Y^R = \{y^R \mid y \in Y\}$; then the integers in Y^R all have binary representations of the form $\alpha_0\alpha_1 \dots \alpha_{r-1}\beta_r\beta_{r+1} \dots \beta_{k-1}$, where $\alpha_0\alpha_1 \dots \alpha_{r-1}$ is arbitrary and $\beta_{k-1}\beta_{k-2} \dots \beta_r$ is the $(k - r)$ -bit binary representation of j . Thus, the integers in Y^R are equally spaced at 2^{k-r} distance apart. As $2^{k-r} \leq n / ((b - a)/4) \leq \frac{1}{2}(d - c)$, there exists $y \in Y$ such that $y^R \in (c, d)$, i.e., $(y, y^R) \in B_n \cap ([a, b] \times (c, d))$. \square

Define a special class of subsets of B_n by $D_{ij} = ([0, i] \times [0, j]) \cap B_n$, where i and j are nonnegative integers. We call a pair (i, j) canonical pair if $(i, i^R) \in D_{ij}$ and $(j^R, j) \in D_{ij}$. In other words, (i, j) is canonical if every (i', j') with $D_{i',j'} = D_{ij}$ satisfies $i' \geq i$ and $j' \geq j$. Clearly, for every D_{ij} , there is a unique canonical pair (i', j') such that $D_{i',j'} = D_{ij}$. From now on, when we mention D_{ij} , it is understood that (i, j) is always a canonical pair.

To prove Theorem 4', let \mathcal{T} be a structure for B_n with $|\mathcal{T}| = m$. Without loss of generality, we assume that every member T of \mathcal{T} is of the form D_{ij} . (For otherwise, we can replace T by the smallest D_{ij} containing T , and obtain a structure \mathcal{T}' with $t'(\mathbf{a}; \mathcal{T}') \leq t'(\mathbf{a}; \mathcal{T})$.)

For integers $r \leq s$, let $N_{r,s}$ denote the number of $D_{ij} \in \mathcal{T}$ such that $i \in [r, s]$. Let $\eta \geq 0$. Consider integral positions $\lfloor n/2 \rfloor \leq l < n$. We say that a position l is η -favorable if, for each integer $\delta \geq 0$, one has $N_{l-\delta, l} \leq \eta(\delta + 1)$.

LEMMA 2. *There exist at least $n/4$ (4μ) -favorable positions l with $\lfloor n/2 \rfloor \leq l < n$.*

Proof. Construct a sequence of disjoint intervals $[i_1, j_1], [i_2, j_2], \dots$ in the following way: Let $j_1 < n$ be the largest integer that is not (4μ) -favorable, and i_1 be an integer such that $N_{i_1, j_1} > 4\mu(j_1 - i_1 + 1)$; inductively, for integer $p > 1$, let j_p be the maximum integer q , $\lfloor n/2 \rfloor \leq q < i_{p-1}$, that is not (4μ) -favorable, and let i_p be an integer such that $N_{i_p, j_p} > 4\mu(j_p - i_p + 1)$; do this until no such j_p can be found. Let $[i_1, j_1], [i_2, j_2], \dots, [i_u, j_u]$ be the sequence constructed. Then

$$\left| \bigcup_{p=1}^u \{i_p, i_{p+1}, \dots, j_p\} \right| < \frac{1}{4\mu} \sum_{p=1}^u N_{i_p, j_p} \leq \frac{1}{4\mu} m = \frac{n}{4}.$$

It follows that there are at least $n/4$ integers l ($\lfloor n/2 \rfloor \leq l < n$) not contained in $\bigcup_{p=1}^u [i_p, j_p]$. These l must all be (4μ) -favorable; otherwise some of them should have been selected as j_p . The lemma follows. \square

LEMMA 3. *Let l ($\lfloor n/2 \rfloor \leq l < n$) be a (4μ) -favorable position. Then there exists an integer $0 \leq q < n$ such that, for $\mathbf{a} = (l, q)$, $A(\mathbf{a})$ cannot be the union of less than $\lfloor \log n / 64(\log(\mu \log n)) \rfloor$ of the members of \mathcal{T} .*

Theorem 4' follows immediately from Lemmas 2 and 3. A proof of Lemma 3 will be given in § 3.3. To give an illustration of the intuitive idea of the proof, we will first prove a weaker version of Lemma 3.

3.2. A weaker version. In this subsection we prove a weaker form of Lemma 3. Let $\mu = m/n$ be fixed. We shall exhibit a vector $\mathbf{a} = (l, y)$, which will be called a query vector, such that $A(\mathbf{a})$ cannot be the union of less than $\Omega(\log \log n)$ of the members of \mathcal{T} . (It is easy to make y an integer in addition, but we will not insist on that here.) This is a weaker form, as Lemma 3 states an $\Omega(\log n / \log \log n)$ lower bound in this situation.

Let us imagine Alice is playing a game against Bob, who knows what \mathcal{T} consists of. Alice wants to exhibit a query vector \mathbf{a} for which Bob cannot find a less than size- $\Omega(\log \log n)$ subcollection of \mathcal{T} whose union gives $A(\mathbf{a})$. Initially, Alice does not know anything about \mathcal{T} , and Bob knows nothing about the value of \mathbf{a} . As the game

proceeds, Bob will gradually reveal the identity of the members of \mathcal{T} , and Alice will successively narrow down and announce her range of the query vector \mathbf{a} . Each time Alice narrows down her range further, the query $A(\mathbf{a})$ becomes a little harder to answer; Alice will make sure that for Bob to answer any query in this range, namely to give a subcollection of \mathcal{T} whose union is $A(\mathbf{a})$, he has to include at least a certain minimum number of members in \mathcal{T} that have already been revealed, independent of what the rest of \mathcal{T} are. We now describe the rules of the game. The game can be visualized as being played on the two-dimensional $x-y$ board $[0, n] \times [0, n]$, with the points of B_n scattered on it. We will use the notation $\mathcal{T}(x, x')$ to stand for the subcollection of $D_{is} \in \mathcal{T}$ with $x \leq i < x'$. (Note that $|\mathcal{T}(r, s+1)| = N_{r,s}$ for integers r, s , where $N_{r,s}$ was as defined in § 3.1.)

Let $l_0 = l+1$ and I_0 be the open interval $(0, n)$. The game proceeds in stages $j = 1, 2, \dots$. In stage j , Alice picks a point $\mathbf{b}^{(j)} = (l_j, y_j) \in B_n$ where $y_j \in I_{j-1}$ and $0 \leq l_j < l_{j-1}$. (Clearly, $l_j \leq l$ for all $j \geq 1$.) Bob then reveals the collection $\mathcal{T}(l_j, l_{j-1})$. Alice now picks an open interval $I_j \subseteq I_{j-1}$ such that $y_j < y$ for all $y \in I_j$ and that no $D_{is} \in \mathcal{T}(l_j, l_0)$ satisfies $s \in I_j$, and announces the choice of I_j to Bob. One way to think about this is to say that, along the vertical line $x = l_j$, the point $\mathbf{b}^{(j)} = (l_j, y_j)$ will lie below the vertical segment $\{l_j\} \times I_j$, and that for any $D_{is} \in \mathcal{T}(l_j, l_0)$, the horizontal line $y = s$ will not pass through the vertical segment $\{l_j\} \times I_j$. The game stops when Alice is unable to find a next $\mathbf{b}^{(j)}$ satisfying the requirements.

The goal of Alice is to prolong the game as much as possible, and Bob's is the opposite. If we can prescribe a strategy for Alice that guarantees that the game will not stop before J stages, then we will have proved a lower bound J as implied immediately by the following statement: If during a particular session, the game stops after J stages, then for any $\mathbf{a} = (l, y)$ with $y \in I_j$, the set $A(\mathbf{a})$ cannot be represented by the union of less than J of the members of \mathcal{T} . We will now prove the above statement.

Without loss of generality, assume that $J \geq 1$. The intuitive idea is that, among the points in $A(\mathbf{a})$, each of the J points $\mathbf{b}^{(j)} = (l_j, y_j)$ has to be covered by a distinct $D_{is} \in \mathcal{T}(l_j, l_{j-1})$. Formally, we prove by induction on $j = 1, 2, \dots, J$ the following statement: For any $\mathbf{a} = (l, y)$ where $y \in I_j$, if $A(\mathbf{a})$ is the union of the members of a subcollection $\mathcal{T}' \subseteq \mathcal{T}$, then at least j of the members of \mathcal{T}' must be in $\mathcal{T}(l_j, l_0)$. Noting that any D_{is} contains the point (i, i^R) , one can easily verify the case $j = 1$. For the inductive step, let $j > 1$ and assume that we have proved the cases for all smaller values. Let $\mathbf{a} = (l, y)$ with $y \in I_j$, and suppose \mathcal{T}' is a collection whose union is $A(\mathbf{a})$. As $y \in I_j \subseteq I_{j-1}$, there are at least $j-1$ members of \mathcal{T}' that are in $\mathcal{T}(l_{j-1}, l_0)$. We will show that there is at least one D_{is} from \mathcal{T}' that is in $\mathcal{T}(l_j, l_{j-1})$. This clearly will complete the induction proof. Let $D_{is} \in \mathcal{T}'$ be a member that contains the point $\mathbf{b}^{(j)}$. It is obvious that $i \geq l_j$. Also $i < l_0$; otherwise D_{is} will contain the point (i, i^R) which is not in $A(\mathbf{a})$. We have thus either $D_{is} \in \mathcal{T}(l_{j-1}, l_0)$ or $D_{is} \in \mathcal{T}(l_j, l_{j-1})$. In the former case we have $s > y'$ for any $y' \in I_{j-1}$ by the construction of I_{j-1} . It follows that $s > y$, and hence the point $(s^R, s) \in D_{is}$ will not be contained in $A(\mathbf{a})$; this contradicts the fact $D_{is} \in \mathcal{T}'$. We must conclude that $D_{is} \in \mathcal{T}(l_j, l_{j-1})$. This completes the inductive step.

We will now describe a strategy for Alice to play the game. For each j let us write I_j as $(w^{(j)}, v^{(j)})$, and the I'_j and I''_j denote the open intervals that are the lower half and the upper half of I_j . That is, let $I'_j = (w^{(j)}, (w^{(j)} + v^{(j)})/2)$, and $I''_j = ((w^{(j)} + v^{(j)})/2, v^{(j)})$. In stage $j \geq 1$, Alice considers the motion of the vertical segment $\{\eta\} \times I'_{j-1}$ as η ranges from $l_{j-1} - 0.1$ towards 0, and jots down the first point of B_n that the segment encounters; call this point (l_j, y_j) , and announce it to Bob. After Bob reveals the collection $\mathcal{T}(l_j, l_{j-1})$, Alice makes a list of the members of $\mathcal{T}(l_j, l_0)$ as $D_{i_1, s_1}, D_{i_2, s_2}, \dots, D_{i_g, s_g}$. Let $V_j \subseteq \{s_1, s_2, \dots, s_g\}$ be the subset of numbers that fall into the

interval I_{j-1} . Then V_j divide I_{j-1} into disjoint open intervals. Alice now picks a longest such open interval, calls it I_j and announces it to Bob. Alice stops the game after J stages when she finds $[0, l_j) \times I'_j$ contains no point of B_N .

So far we have not used Lemma 1 and Lemma 2. We will now, with the help of the lemmas, show that by using the above strategy, Alice can be guaranteed that the game will last at least $\Omega(\log \log n)$ stages. This of course then implies the existence of $\mathbf{a} = (l, y)$ such that $A(\mathbf{a})$ cannot be the union of less than $\Omega(\log \log n)$ of the members of \mathcal{F} .

Let us use the notation $\|K\|$ for the total length of K , if K is the union of a finite set of disjoint intervals. Suppose that the game stops after J stages. Let $d_j = l_0 - l_j$. Clearly, $d_0 = 0$ and $\|I_0\| = n$. By Lemma 1, we have $(l_{j-1} - l_j) \cdot \|I'_{j-1}\| \leq 8n$. Using the fact that $\|I'_{j-1}\| = \|I_{j-1}\|/2$, we obtain that, for all $1 \leq j \leq J$,

$$(2) \quad d_j - d_{j-1} \leq \frac{16n}{\|I_{j-1}\|}.$$

Now, notice that the size of V_j is at most $4\mu \cdot d_j$. It follows that $\|I_j\| \geq \|I''_{j-1}\|/(4\mu \cdot d_j)$. Thus, for all $1 \leq j \leq J$,

$$(3) \quad \|I_j\| \geq \frac{\|I_{j-1}\|}{8\mu \cdot d_j}.$$

Furthermore, we claim that

$$(4) \quad l_j < \frac{16n}{\|I_j\|}.$$

Otherwise, by Lemma 1, Alice should have been able to locate a point in the rectangle $[0, l_j) = I_j$, and to start stage $J + 1$.

We shall now prove that, for sufficiently large n , $J \geq \log_c \log_c n - 2$, where $c = 20\mu$. Let $n_0 = c^{c^{100}}$, and $n > n_0$ be any large integer of the form 2^k . Let us assume that $J < \log_c \log_c n - 2$, and will derive a contradiction. It is straightforward to use (2), (3) to verify by induction that, for $1 \leq j \leq J$, one has $d_j < c^{c^{j-1}}$, $\|I_j\| > n/c^c$. For $j = J$, this leads to $d_J < n/4$, $\|I_J\| > 64$. Thus, $l_J = l_0 - d_J > n/2 - n/4 = n/4 \geq 16n/\|I_J\|$, which is a contradiction to (4). This completes the proof of the weak form of Lemma 3.

To prove Lemma 3 in its full strength, we have to refine the above arguments in two ways. Firstly, the division of I_{j-1} into intervals I'_{j-1} and I''_{j-1} of the same size is somewhat arbitrary. The benefit of having a larger I'_{j-1} would be that a point $\mathbf{b}^{(j)}$ can be found with smaller $l_{j-1} - l_j$, but this has to be balanced by the fact that a larger I'_{j-1} (and hence a smaller I''_{j-1}) may narrow down the range of query vector more in the next stage of the game and cause a larger $l_j - l_{j+1}$. Thus there is some freedom on the choice of the ratio of $\|I'_{j-1}\|$ to $\|I''_{j-1}\|$, but in order to be beneficial, it has to be carefully chosen dependent on the modification of the proof in other parts. Secondly, in the above proof Alice has perhaps given away too much information about the range of her query vector; Bob may concentrate on putting all his $D_{is} \in \mathcal{F}(l_j, l_{j-1})$ with $s \in I''_{j-1}$. If we change the rules of the game a little, in such a way that Alice can hold on to more intervals, and reserve the right to choose query vector in any interval, then Bob may have to divide his $D_{is} \in \mathcal{F}(l_j, l_{j-1})$ with s among more intervals. In this fashion, the decrease of the size of intervals I_j as a function of j will be much slower, which would prolong the game.

In the next subsection we will give a proof of Lemma 3, utilizing the above ideas. At each stage j , we will have a family of intervals \mathcal{F}_j , and the construction of intervals

in \mathcal{I}_j will involve splitting intervals into unequal sizes. The proof contains many numerical quantities, and will be presented in a more direct fashion than the above arguments, so as to facilitate the verification of the proof. We remark that the symbols used in the following proof may not denote exactly the same quantity as in the preceding proof; the notations will be defined anew.

3.3. A proof of Lemma 3.

Proof. Let $r = \lfloor \log n / 64(\log(\mu \log n)) \rfloor$. Without loss of generality, we assume that $r \geq 1$. This means in particular $n \geq 10^4$. Also recall that we have assumed that $\mu = m/n \geq 2$.

For $0 \leq j \leq r$, define $k_j = \lfloor n / ((10,000\mu)^j (j!)^4) \rfloor$. Let $d_0 = 0$ and $d_j = \lceil 80nj^2 / (k_{j-1}) \rceil$ for $1 \leq j \leq r$. The following inequalities are straightforward to verify (see Appendix):

$$(5) \quad k_j > 10 \quad \text{for } 0 \leq j \leq r,$$

$$(6) \quad 8\mu d_j k_j \leq \frac{n}{8j^2} \quad \text{for } 1 \leq j \leq r,$$

$$(7) \quad \frac{(d_j - d_{j-1})k_{j-1}}{8j^2} \geq 8n \quad \text{for } 1 \leq j \leq r.$$

Let $l_j = l + 1 - d_j$ for $0 \leq j \leq r$. We are going to construct a sequence $\mathcal{I}_0, \mathcal{I}_1, \mathcal{I}_2, \dots, \mathcal{I}_r$, where each \mathcal{I}_j is a finite family of disjoint open intervals. They will be shown to possess the following properties:

P1. For $0 \leq j \leq r$, \mathcal{I}_j is nonempty; furthermore, each $I \in \mathcal{I}_j$ has length > 1 ;

P2. Let $y \in I \in \mathcal{I}_j$, where $1 \leq j \leq r$ and y is an integer. If $A(\mathbf{a}) = T_1 \cup T_2 \cup \dots \cup T_w$, where $\mathbf{a} = (l, y)$ and $T_i \in \mathcal{T}$, then there are at least j T_α of the form D_{is} with $i \in [l_j, l]$.

This will prove the lemma, since one can pick by P1 an integer $y \in I \in \mathcal{I}_r$, and one knows by P2 $t'(\mathbf{a}; \mathcal{T}) \geq r$ where $\mathbf{a} = (l, y)$.

We construct \mathcal{I}_j inductively. Define $\mathcal{I}_0 = \{(0, n)\}$. Let $0 < j \leq r$, and suppose $\mathcal{I}_0, \mathcal{I}_1, \dots, \mathcal{I}_{j-1}$ have been constructed; we will construct \mathcal{I}_j . We first introduce some notation. For an open interval $I = (a, b)$, define two open intervals $I^{j\text{-bottom}} = (a, a + (b - a) / 8j^2)$, and $I^{j\text{-top}} = (a + (b - a) / 8j^2, b)$. Let

$$V_j = \{s \mid \exists D_{is} \in \mathcal{T} \text{ with } i \in [l_j, l]\}.$$

For each $I \in \mathcal{I}_{j-1}$, consider the points of V_j that divide the interval $I^{j\text{-top}}$ into disjoint open intervals; let \mathcal{I}_I be the set of such intervals that have length $\geq k_j$. Define $\mathcal{I}_j = \bigcup_{I \in \mathcal{I}_{j-1}} \mathcal{I}_I$.

It may help one understand the above construction and the discussions to come, if one visualizes \mathcal{I}_j as a collection of intervals on the vertical line $x = l_j$ in the two-dimensional x - y key space.

It remains to verify that properties P1 and P2 are satisfied. We will use the notation $\|K\|$ to denote the measure (or, length) of K , if K is the union of a finite number of disjoint intervals. P1 is obviously true for $j = 0$. To show P1 for $j > 0$, it suffices to prove

$$(8) \quad \left\| \bigcup_{I \in \mathcal{I}_j} I \right\| > 0,$$

as it implies that \mathcal{I}_j is nonempty; the other part of P1 follows from inequality (5) and the fact that each $I \in \mathcal{I}_j$ has length $\geq k_j$ by construction. To prove (8), define $M_i = \bigcup_{I \in \mathcal{I}_{i-1}} I - \bigcup_{I \in \mathcal{I}_i} I$. Observe that $\bigcup_{I \in \mathcal{I}_j} I$ differs from $[0, n] - \bigcup_{i=1}^j M_i$ by at most a finite

number of points. Thus

$$(9) \quad \left\| \bigcup_{I \in \mathcal{F}_j} I \right\| \geq n - \sum_{i=1}^j \|M_i\|.$$

Now note that M_i is the union of a finite number of points plus two types of intervals: those that are $I^{i-\text{bottom}}$ for some $I \in \mathcal{F}_{i-1}$, and those that have length $< k_i$ and are contained in $I^{i-\text{top}}$ with at least one endpoint in V_i . It follows that

$$(10) \quad \|M_i\| \leq \frac{1}{8i^2} \left\| \bigcup_{I \in \mathcal{F}_{i-1}} I \right\| + 2|V_i|k_i.$$

Since position l is (4μ) -favorable, we have

$$(11) \quad |V_i| \leq 4\mu d_i.$$

From (10), (11) and (6), we obtain for $i \geq 1$

$$(12) \quad \|M_i\| \leq \frac{1}{8i^2} n + 8\mu d_i k_i \leq \frac{1}{8i^2} n + \frac{1}{8i^2} n \leq \frac{n}{4i^2}.$$

From (9) and (12), we obtain

$$\left\| \bigcup_{I \in \mathcal{F}_j} I \right\| \geq n \left(1 - \frac{1}{4} \sum_{i=1}^{\infty} \frac{1}{i^2} \right) = n \left(1 - \frac{1}{4} \frac{\pi^2}{6} \right) > \frac{1}{8} n.$$

This proves (8), and hence property P1.

To prove P2, we proceed by induction. It is obviously true for $j = 1$. Let $j > 1$, and assume that we have proved P2 for $j - 1$; we will prove it for j . Let $y \in I \in \mathcal{F}_j$ be an integer, and let $\mathbf{a} = (l, y)$. Assume that $A(\mathbf{a}) = T_1 \cup T_2 \cup \dots \cup T_u$ where $T_v \in \mathcal{T}$. We will prove that there are at least j T_v of the form D_{is} with $i \in [l_j, l]$.

By the construction of \mathcal{F}_j , there exists an $I' \in I_{j-1}$ such that $I \subseteq I'$. By the induction hypothesis, there must be at least $j - 1$ T_v of the form D_{is} with $i \in [l_{j-1}, l]$. Therefore, we need only to show that there exists a T_v of the form D_{is} with $i \in [l_j, l_{j-1})$.

Consider the square $[l_j, l_{j-1}) \times I'^{j-\text{bottom}}$. Note that, by (7),

$$(l_{j-1} - l_j) \|I'^{j-\text{bottom}}\| = (d_j - d_{j-1}) \frac{k_{j-1}}{8j^2} \geq 8n.$$

It follows by Lemma 1 that there exists a point

$$(x, y') \in B_n \cap ([l_j, l_{j-1}) \times I'^{j-\text{bottom}}).$$

As $(x, y') \in A(\mathbf{a})$, there must be a $T_v = D_{is}$ containing the point (x, y') . Clearly $i \geq x \geq l_j$ and $s \geq y'$. We want to show that $i \in [l_j, l_{j-1})$ by ruling out the other possibilities. If $i > l$, then D_{is} will contain a point not in $A(\mathbf{a})$, namely, (i, i^R) , which is not allowed. If $i \in [l_{j-1}, l]$, then by the construction of \mathcal{F}_{j-1} , either $s \geq u'$ or $s \leq u$, where $I' = (u, u')$. As $s \geq y' > u$, this means we must have $s \geq u'$ and hence $s > y$. This last inequality implies that the point $(s^R, s) \notin A(\mathbf{a})$, which contradicts the fact that $(s^R, s) \in D_{is} \subseteq A(\mathbf{a})$. We have thus proved that $i \in [l_j, l_{j-1})$. This completes the induction proof of Property 2. We have proved Lemma 3. \square

We have completed the proof of Theorem 4. As mentioned earlier, this also proves Theorem 1.

4. One-dimensional dynamic partial sums. In this section we will prove Theorem 2. Without loss of generality, we will assume $n \geq 16$ and is a power of 2. Let \mathcal{A} be an algorithm for the one-dimensional dynamic partial sum problem. Consider the following

sequence σ of instructions:

INSERT $(0^R, s_0)$, QUERY (g_0) , INSERT $(1^R, s_1)$, QUERY (g_1) , \dots

\dots , INSERT (j^R, s_j) , QUERY (g_j) , \dots , INSERT $((n-1)^R, s_{n-1})$, QUERY (g_{n-1}) ,

where $0 \leq j < n$ is chosen such that QUERY (g_j) requires the maximum number of operations to process at that time. We will show that the time used by \mathcal{A} to process this sequence is $\Omega(n \log n / \log \log n)$.

Consider the two-dimensional static partial sum problem for the file $F_n = \{(\mathbf{b}_j, s_j) | 0 \leq j < n\}$ where $\mathbf{b}_j = (j, j^R)$. Let us construct a storage scheme \mathcal{S} from the way \mathcal{A} processes σ . Let z_1, z_2, \dots, z_m be the family of all positive linear functions (of s_i) that have been computed during the processing of σ . Then $\mathcal{S} = \{z_1, z_2, \dots, z_m\}$ is a storage scheme. If $m \geq n \log n$, then the time used by \mathcal{A} on σ is $\Omega(n \log n)$, and we have proved the theorem. Thus we can assume $m < n \log n$. By Theorem 4, there exists at least $\lfloor n/4 \rfloor l$ such that $t((l, h(l)); \mathcal{S}) = \Omega(\log n / \log \log n)$ for some $h(l)$. Now, consider the current set \mathcal{S}_l of the contents of variables just after the instruction INSERT (l^R, s_l) has been processed. The response to QUERY $(h(l))$ at that time should be exactly $\text{resp}((l, h(l)); F_n)$ in the two-dimensional static problem. Since $\mathcal{S}_l \subseteq \mathcal{S}$, the time needed by \mathcal{A} to process QUERY $(h(l))$ is at least $t((l, h(l)); \mathcal{S}) = \Omega(\log n / \log \log n)$. This means QUERY (g_l) takes time $\Omega(\log n / \log \log n)$ to process, as QUERY (g_l) is chosen to be the hardest query to process at that time. It follows that the time used by \mathcal{A} to process σ is at least $\lfloor n/4 \rfloor \cdot \Omega(\log n / \log \log n) = \Omega(n \log n / \log \log n)$.

We have proved Theorem 2.

5. Relations between static and dynamic problems. We will prove Theorem 3. Let $F = \{(\mathbf{r}_i, s_i) | 0 \leq i < n\}$ be a file with $\mathbf{r}_i \in Z^d$ and $s_i \in S$. We will construct a storage scheme \mathcal{S} using space $m = O(n\nu(n))$ and time $O(\nu(n))$.

Write $\mathbf{r}_i = (x_i, \hat{\mathbf{r}}_i)$ where x_i is the first component of \mathbf{r}_i , and $\hat{\mathbf{r}}_i \in Z^{d-1}$ is the vector of the other components. Without loss of generality, we can assume $x_0 \leq x_1 \leq x_2 \leq \dots \leq x_{n-1}$. Consider the processing of the following sequence σ for the $(d-1)$ -dimensional dynamic problem using algorithm \mathcal{A} : INSERT $(\hat{\mathbf{r}}_0, s_0)$, INSERT $(\hat{\mathbf{r}}_1, s_1)$, \dots , INSERT $(\hat{\mathbf{r}}_{n-1}, s_{n-1})$. Let $\mathcal{S} = \{z_1, z_2, \dots, z_m\}$ be the family of all positive linear functions of s_i that have been computed during this process. Then $m = O(n\nu(n))$ because the total number of operations used by \mathcal{A} is $O(n\nu(n))$. It remains to prove that $t_F(\mathcal{S}) = O(\nu(n))$.

Let $\mathbf{a} = (l, \hat{\mathbf{a}})$ be any element in Z^d . Find i such that $x_i \leq l < x_{i+1}$ (we agree that $x_{-1} = -\infty$ and $x_n = +\infty$). Consider the processing of the $(d-1)$ -dimension sequence σ using \mathcal{A} again. If one asks QUERY $(\hat{\mathbf{a}})$ immediately after INSERT $(\hat{\mathbf{r}}_i, s_i)$, the answer at that time should be $\text{resp}(\mathbf{a}; F)$. Since QUERY $(\hat{\mathbf{a}})$ can be answered in $O(\nu(n))$ operations, $\text{resp}(\mathbf{a}; F)$ must be a positive combination of no more than $O(\nu(n))$ $z_j \in \mathcal{S}$. This proves $t_F(\mathbf{a}; \mathcal{S}) = O(\nu(n))$ and hence $t_F(\mathcal{S}) = O(\nu(n))$.

We have finished the proof of Theorem 3.

6. Open problems. We list below a few open problems that seem to deserve further study.

(A) It is easy to see that $O(n \log n)$ time is sufficient to process $O(n)$ instructions for the dynamic one-dimensional partial sum problem. Can one close the gap between this upper bound and the lower bound $\Omega(n \log n / \log \log n)$ given by Theorem 2? Similarly, what is the true tradeoff for the static two-dimensional partial sum problem? Is it possible to achieve $m = O(n)$, $t = O(\log n / \log \log n)$?

(B) What is the space-time tradeoff for static partial sum queries when $d > 2$? It is known [11] that $m = O(n(\log n)^{d-1})$, $t = O((\log n)^{d-1})$ can be achieved. Can one prove that, if $m = O(n)$, then $t = \Omega((\log n)^{d-1})$?

(C) What can be said when S is a group instead of a semigroup (i.e., subtraction allowed)? Some results in this direction can be found in Fredman [5], but we know much less about range queries in the group model.

Appendix. Proof of inequalities (5)–(7). Let $c = 10^4$. Recall $n \geq 10^4$, $\mu \geq 2$, and

$$r = \left\lfloor \frac{\log n}{64(\log(\mu \log n))} \right\rfloor,$$

$$k_j = \left\lfloor \frac{n}{(c\mu)^j(j!)^4} \right\rfloor \quad \text{for } j \geq 0,$$

$$d_0 = 0, \quad d_j = \left\lceil \frac{80j^2n}{k_{j-1}} \right\rceil \quad \text{for } j \geq 1.$$

We now prove inequalities (5), (6), and (7) in § 3.

Inequality (5). $k_j > 10$ for all $0 \leq j \leq r$.

Proof. As k_j is nonincreasing in j , it suffices to prove $k_r > 10$. By elementary manipulation

$$\begin{aligned} \log((c\mu)^r(r!)^4) &\leq r \log(c\mu) + 4r \log r \\ &\leq \frac{\log n}{64 \log \mu} (\log \mu + 16) + 4 \frac{\log n}{64 \log \log n} \log \log n \\ &\leq \left(\frac{1}{64} + \frac{1}{4} + \frac{1}{16} \right) \log n \\ &\leq \log \sqrt{n}. \end{aligned}$$

This proves

$$(c\mu)^r(r!)^4 \leq \sqrt{n}.$$

Hence

$$k_r \geq \lfloor \sqrt{n} \rfloor > 10. \quad \square$$

Inequality (6). $8\mu d_j k_j \leq n/8j^2$ for $1 \leq j \leq r$.

Proof. Using inequality (5), we obtain

$$d_j \leq 1 + \frac{80nj^2}{\frac{10}{11}n / ((c\mu)^{j-1}((j-1)!)^4)} \leq 90j^2(c\mu)^{j-1}((j-1)!)^4.$$

This leads to

$$8\mu k_j d_j \leq 8\mu \cdot 90j^2 \frac{n}{(c\mu)^j j^4} \leq \frac{n}{8j^2}. \quad \square$$

Inequality (7). $(d_j - d_{j-1})k_{j-1}/8j^2 \geq 8n$ for $1 \leq j \leq r$.

Proof. For $j = 1$, it is trivial to verify it. Let $1 < j \leq r$, then

$$\frac{d_j}{d_{j-1}} \geq \frac{k_{j-2}}{2k_{j-1}} \geq \frac{1}{4}(j-1)^4(c\mu) > 10.$$

Thus,

$$(d_j - d_{j-1})k_{j-1} \geq \frac{9}{10} d_j k_{j-1} \geq \frac{9}{10} \frac{80j^2 n}{k_{j-1}} k_{j-1} \geq 64j^2 n.$$

Inequality (7) follows. \square

REFERENCES

- [1] J. L. BENTLEY AND H. A. MAUER, *Efficient worst-case data structure for range searching*, Acta Inform., 13 (1980), pp. 155–168.
- [2] J. L. BENTLEY AND M. I. SHAMOS, *A problem in multivariate statistics: algorithms, data structure and applications*, Proc. of the 15th Allerton Conference on Communications, Control, and Computing (1977), pp. 193–201.
- [3] M. L. FREDMAN, *Lower bounds on the complexity of some optimal data structures*, this Journal, 10 (1981), pp. 1–10.
- [4] ———, *A lower bound on the complexity of orthogonal range queries*, J. Assoc. Comput. Mach., 28 (1981), pp. 696–705.
- [5] ———, *The complexity of maintaining an array and computing its partial sums*, Assoc. Comput. Mach., 29 (1982), pp. 250–260.
- [6] M. L. FREDMAN AND D. J. VOLPER, *Query time versus redundancy trade-offs for range queries*, J. Comput. System Sci., 23 (1981), pp. 355–365.
- [7] T. LENGAUER, *Upper and lower bounds on time-space tradeoffs in a pebble game*, Stanford Computer Science Department Technical Report STAN-CS-79-745, Stanford CA, July 1979.
- [8] G. S. LUEKER, *A data structure for orthogonal range queries*, Proc. 19th IEEE Symposium on Foundations of Computer Science, 1978, pp. 28–34.
- [9] R. L. RIVEST, *Partial match retrieval algorithms*, this Journal, 5 (1976), pp. 19–50.
- [10] D. E. WILLARD, *Polygon retrieval*, this Journal, 11 (1982), pp. 149–165.
- [11] ———, *New data structures for orthogonal range queries*, TR-22-78, Center for Research in Computing Technology, Harvard Univ., Cambridge, MA, 1978.
- [12] A. C. YAO, *On the space-time tradeoff for answering range queries*, to appear.

AN EFFICIENT ALGORITHM FOR FINDING MULTICOMMODITY FLOWS IN PLANAR NETWORKS*

KAZUHIKO MATSUMOTO†, TAKAO NISHIZEKI† AND NOBUJI SAITO†

Abstract. This paper presents an efficient algorithm for finding multicommodity flows in planar graphs. Suppose that G is an undirected planar graph with all sources and sinks on the boundary of the outer face and that a real-valued demand is given for each source-sink pair. The algorithm decides whether G has multicommodity flows, each from a source to a sink and of a given demand, and actually finds them if G has. It spends $O(kn + n^2(\log n)^{1/2})$ time and $O(kn)$ space if G has n vertices and k source-sink pairs.

Key words. algorithm, cut-condition, planar graph, polynomial time, max flow-min cut theorem, multicommodity flows, network

1. Introduction. The network flow problem and its variants have been extensively studied. The original and most basic problem is that of finding the maximum flow of a single commodity in an arbitrary directed graph. The most basic theorem of flow theory is the max flow-min cut theorem of Ford and Fulkerson [4] which holds for single-commodity and two-commodity flows. There are efficient algorithms for finding a maximum single-commodity flow; the $O(|E||V| \log |V|)$ time algorithm of Sleator and Tarjan [13], [14] is the theoretically best known one for sparse graphs. (V is the set of vertices and E is the set of edges.) Two-commodity flows in undirected graphs can be found by solving two single-commodity flow problems, so it can be done in $O(|E||V| \log |V|)$ time [7], [12].

The situation is different with regard to flows of more than two commodities. No true polynomial time algorithm is known for the multicommodity flow problem on general graphs. Like all network flow problems, the multicommodity flow problem (planar or nonplanar) can be formulated as a linear program. Thus it can be solved in pseudo-polynomial time by the ellipsoid method of Khachiyan [9]. Unfortunately this method seems to be inefficient in practice. The simplex method is more practical, but experience has shown that for many specific problems special purpose algorithms can be devised which work better than the simplex method.

A specific case of the multicommodity flow problem in which the network is planar has a number of important applications, such as the control of communication or traffic in networks, and routing in VLSI. Several papers have been published on this problem [2], [11], [15].

In this paper we concentrate on planar undirected graphs which arise in many applications, and give an efficient algorithm for finding multicommodity flows in these graphs. Suppose that G is an undirected planar graph with all sources and sinks on the boundary of the outer face and that a real-valued demand is given for each source-sink pair. If G has n vertices and k source-sink pairs, the algorithm decides in $O(n^2)$ time whether G has multicommodity flows, each from a source to a sink satisfying a given demand, and actually finds them in $O(kn + n^2(\log n)^{1/2})$ time using $O(kn)$ space. In this paper we use the Unit Cost RAM [1] as the machine model, assuming that each arithmetic operation costs one unit of time.

In our algorithm we apply a shortest path algorithm to the dual of a given planar graph [1], [3], [5]. In this sense our algorithm is similar to those in [2], [6], [8], [11]. However the well-known algorithm of "topmost" augmenting path [4], [6], [8], which

* Received by the editors April 1, 1982, and in final revised form October 15, 1983.

† Department of Electrical Communications, Faculty of Engineering, Tohoku University, Sendai 980, Japan.

finds efficiently a single-commodity flow in a planar graph, cannot be adapted directly to our multicommodity flow problem. Recently Okamura and Seymour have shown that the max flow–min cut theorem holds true also for the case of multicommodity flows in the same planar graphs as ours [10]. Indeed we employ their proof technique in our algorithm, modifying it in many points in order to guarantee the time and space complexity.

2. Preliminaries. A flow network $N = (G, P, c)$ is a triplet, where:

(i) $G = (V, E)$ is a finite undirected simple graph with vertex set V and edge set E ;

(ii) P is the set of *source–sink pairs* (s_i, t_i) , where source s_i and sink t_i are distinguished vertices in G .

(iii) $c: E \rightarrow R^+$ is the *capacity function*. (R (or R^+) denotes the set of (positive) real numbers.)

A network $N = (G, P, c)$ is *planar* if G is planar, and is a *k-network* if N has k source–sink pairs, that is, $|P| = k$. Fig. 1 illustrates two planar 3- and 4-networks.

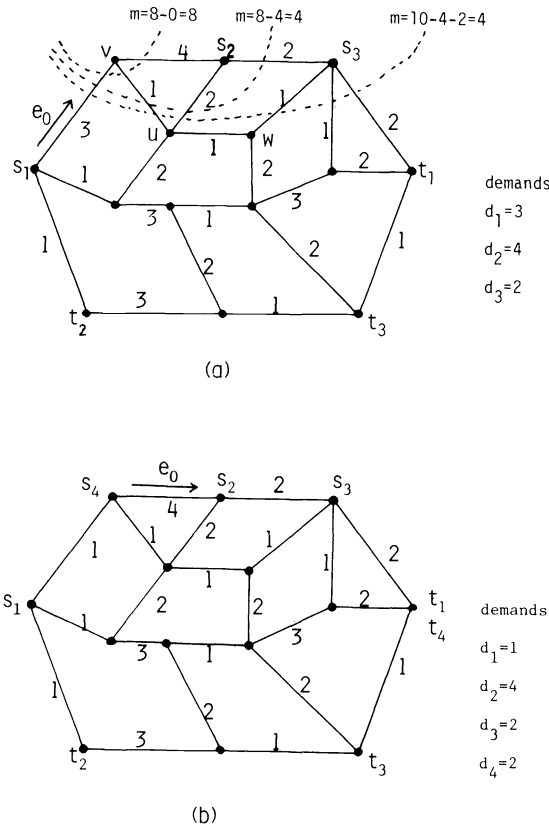


FIG. 1. Two planar networks. (Numbers next to edges are capacities.)

Each source–sink pair (s_i, t_i) of N is given a nonnegative *demand* $d_i \geq 0$. Although G is undirected, we orient the edges of G arbitrarily so that the sign of a value of a flow function can indicate the real direction of the flow in an edge. A set of functions $\{f_1, f_2, \dots, f_k\}$ with each $f_i: E \rightarrow R$ is *k-commodity flows* of demands d_1, d_2, \dots, d_k if it satisfies:

(a) the *capacity rule*: for each $e \in E$

$$\sum_{i=1}^k |f_i(e)| \leq c(e);$$

(b) the *conservation rule*: each f_i satisfies

$$\text{IN}(f_i, v) = \text{OUT}(f_i, v) \quad \text{for each } v \in V - \{s_i, t_i\},$$

and

$$\text{OUT}(f_i, s_i) - \text{IN}(f_i, s_i) = \text{IN}(f_i, t_i) - \text{OUT}(f_i, t_i) = d_i,$$

where $\text{IN}(f_i, v)$ is the total amount of the flow f_i of commodity i entering v , that is, $\text{IN}(f_i, v) = \sum f_i(e) - \sum f_i(e')$, the first sum being over all the edges e entering v and with $f_i(e) > 0$, and the second over all the edges e' emanating from v and with $f_i(e') < 0$; $\text{OUT}(f_i, v)$, similarly defined, is the total amount of the flow of commodity i emanating from v .

Figure 2 illustrates three-commodity flows satisfying the given demands $d_1 = 3$, $d_2 = 4$ and $d_3 = 2$ in the network of Fig. 1(a). Arrows indicate actual directions of flows through edges. Numbers next to arrows are amounts of flows.

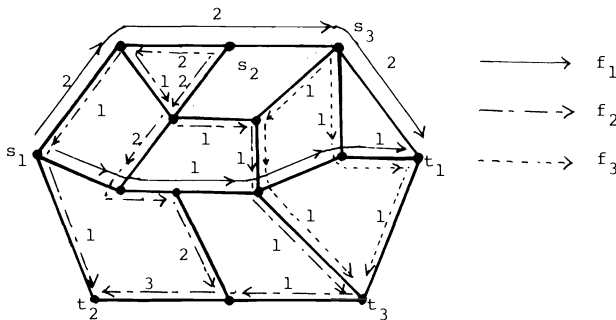


FIG. 2. Three-commodity flows in the network of Fig. 1(a).

Although a planar graph G may be altered during the execution of our algorithm, we generically denote by B the boundary of the outer face of G . We use B also for the set of edges on B when there is no possibility of confusion. The boundary B is a (simple) cycle if G is 2-connected, but is a closed walk in general. A *bridge*, i.e. an edge whose deletion disconnects a connected graph, may appear twice in B . We also assume that the vertices on B are v_0, v_1, \dots, v_b , taken in clockwise order.

We denote by $E(X, Y)$ the set of edges with one end in $X \subset V$ and the other in $Y \subset V$. If $X \subset V$, then $E(X) = E(X, V - X)$ is called a *cut*. Define:

$$c(X, Y) = \sum_{e \in E(X, Y)} c(e); \text{ and}$$

$$c(X) = c(X, V - X) \text{ (the capacity of a cut).}$$

We denote by $d(X, Y)$ the sum of the demands of all source-sink pairs with a source or sink in X and the other in Y . Define $d(X) = d(X, V - X)$. Clearly $E(X) = E(V - X)$, $c(X) = c(V - X)$, and $d(X) = d(V - X)$.

We say that a network N satisfies the *cut condition* for the given demands if $c(X) \geq d(X)$ for every $X \subset V$. The cut condition is necessary for the existence of k -commodity flows satisfying given demands in a k -network, but not always sufficient. However Okamura and Seymour have proved the following theorem.

THEOREM 1[10]. *Let $N = (G, P, c)$ be a planar k -network, and let all the sources and sinks be on the boundary B of the outer face of a planar graph G . Then N has k -commodity flows satisfying the demands if and only if N satisfies the cut condition.*

We denote by $G|X$ the graph obtained from G by deleting the vertices in $V - X$ together with the edges adjacent to the vertices in $V - X$. Since the following two lemmas are easy to prove, we omit the proofs. (These lemmas hold even if G is not planar.)

LEMMA 1[10]. *Let $G = (V, E)$ be a connected graph. A network $N = (G, P, c)$ satisfies the cut condition if and only if $c(X) \geq d(X)$ for each X such that both $G|X$ and $G|(V - X)$ are connected.*

LEMMA 2[10]. *If $G = (V, E)$ is a graph and $X, Y \subset V$, then*

$$c(X \cap Y) + c(X \cup Y) = c(X) + c(Y) - 2c(X - Y, Y - X), \quad \text{and}$$

$$d(X \cap Y) + d(X \cup Y) = d(X) + d(Y) - 2d(X - Y, Y - X).$$

3. Test of feasibility. In this section we present an algorithm for deciding whether network N has multicommodity flows satisfying given demands, i.e., for testing feasibility. In what follows, we assume that $N = (G, P, c)$ is a planar network, G is a planar connected graph with n vertices, and all the sources and sinks are on the boundary B of the outer face of G .

First we define some terms. For $X \subset V$, define the *margin* $m(X)$ of a cut $E(X)$ as $m(X) = c(X) - d(X)$. For $e, e' \in B$, define $m(e, e')$ as follows:

$$m(e, e') = \min \{m(X) \mid X \subset V, E(X) \cap B = \{e, e'\}\},$$

where $m(e, e') = \infty$ if there exists no $X \subset V$ such that $E(X) \cap B = \{e, e'\}$. That is, $m(e, e')$ is the minimum margin of cuts containing edges e and e' of B . It should be noted that e and e' are not always distinct in the definition above, and that $m(e, e) = \infty$ unless e is a bridge.

Consider the planar network N depicted in Fig. 1(a) to illustrate the terms above. If $X = \{u, v, w, s_2\}$ then $c(X) = 10$, $d(X) = 4$ and $m(X) = 6$. If $X = \{s_1, t_2\}$ then $m(X) = 0$. If $e = (s_1, v)$ and $e' = (s_2, s_3)$, then $m(e, e') = m(\{v, s_2\}) = 4$.

Combining Theorem 1 and Lemma 1, we obtain the following lemma.

LEMMA 3. *A planar k -network $N = (G, P, c)$ has k -commodity flows satisfying the demands if and only if $m(e, e') \geq 0$ for every $e, e' \in B$.*

Proof. We have to prove the implication in both directions. First assume that the network has k -commodity flows satisfying the demands. Then by Theorem 1, the network satisfies the cut condition. The cut condition says that $c(X) \geq d(X)$ for all X , so $m(X) \geq 0$ for all X , so $m(e, e') \geq 0$ for any e and e' .

To prove the converse assume that $m(e, e') \geq 0$ for all e and e' in B . This implies immediately that the cut condition is satisfied for any set X such that $|E(X) \cap B| = 1$ or 2. The cut condition is also satisfied for any set X such that $|E(X) \cap B| = 0$, because then the vertices of B are contained in X or $V - X$. Since all sources and sinks are on the boundary B , we have $d(X) = 0$, which implies the cut condition for these sets. Because of planarity, any set X that has the property that both $G|X$ and $G|(V - X)$ are connected satisfies $|E(X) \cap B| = 0, 1, \text{ or } 2$. Therefore by Lemma 1 the network satisfies the cut condition for all X , and so by Theorem 1 there are k -commodity flows satisfying the demands. Q.E.D.

To test the feasibility, one must check whether every $e, e' \in B$ satisfies $m(e, e') \geq 0$. For the purpose one must compute $c(X)$ and $d(X)$ for all $X \subset V$ with $|E(X) \cap B| = 1, 2$. Since there may exist an exponential number of these X 's, the straightforward

method cannot guarantee the polynomial time boundedness. However we can test the feasibility in $O(n^2 \log^* n)$ time as follows.

Since all the sources and sinks are on the boundary of the outer face, clearly $d(X) = d(Y)$ whenever $E(X) \cap B = E(Y) \cap B$. Hence if $e, e' \in B$ are fixed then $d(X)$ is constant for all X with $E(X) \cap B = \{e, e'\}$. Denote the constant by $d(e, e')$, and let

$$c(e, e') = \min \{c(X) \mid X \subset V \text{ and } E(X) \cap B = \{e, e'\}\},$$

then

$$m(e, e') = c(e, e') - d(e, e').$$

Thus we shall show that $c(e, e')$ and $d(e, e')$ can be computed in the claimed time.

One can compute $d(e, e')$ for a fixed $e \in B$ and all $e' \in B$ in $O(k + b)$ time. Remember that $|P| = k$ and $|B| = b + 1$. These values can be easily updated for the new e next to the current e on B . Thus we can compute $d(e, e')$ for all $e, e' \in B$ in $O(k + b^2)$ time.

We now show how to compute $c(e, e')$. Construct a dual graph $G^* = (V^*, E)$ of $G = (V, E)$, and consider the capacity function c as a length function of G^* . Then the minimum value $c(e, e')$ of cuts containing e and e' is equal to the length of a shortest nontrivial path joining e and e' in G^* . The famous Dijkstra's algorithm finds the shortest paths in G^* from a particular vertex to all other vertices in $O(|E| \log |V^*|)$ time [1], [3]. Thus we can compute $c(e, e')$ in $O(n \log n)$ time for a fixed $e \in B$ and all $e' \in B$ by applying the algorithm to G^* with a simple modification. Note that $|E| = O(n)$ and $|V^*| = O(n)$ since G is planar. Thus all $c(e, e')$ can be computed in $O(bn \log n)$ time. Recently Frederickson gave two shortest path algorithms for planar graphs [5]: one finds in $O(n(\log)^{1/2})$ time the shortest paths from a particular vertex to all other vertices; the other finds in $O(n^2 \log^* n)$ time the shortest paths between all pairs of vertices. Using these algorithms, one can improve the bound above: one can compute all $c(e, e')$ in $O(\min \{n^2 \log^* n, bn(\log n)^{1/2}\})$ time.

Hence we have shown that the feasibility can be tested in $O(\min \{n^2 \log^* n, bn(\log n)^{1/2}\})$ time.

Remark. R. Hassin of Tel-Aviv University and an anonymous referee¹ have pointed out that the bound above can be improved by the simple expedient of adding new, exterior edges of capacity zero as follows. Let the vertices on B that are source or sink be $v_{i_1}, v_{i_2}, \dots, v_{i_j}$ taken in clockwise order on B . The new edges are $(v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \dots, (v_{i_j}, v_{i_1})$. Clearly $|B'| = j$ if B' is the new outer boundary. Since $j \leq \min \{2k, n\}$, the feasibility can be tested in $O(\min \{n^2 \log^* n, kn(\log n)^{1/2}\})$ time, an advantage whenever $k = o(n \log^* n / (\log n)^{1/2})$.

4. Algorithm DELTAFLOW. The details of our algorithm MULTIFLOW will be given in the next section. In order to explain the core of MULTIFLOW we present, in this section, an "algorithm" DELTAFLOW which finds k -commodity flows in a planar network N , but does not run necessarily in polynomial time.

procedure DELTAFLOW (N);

begin

1. select an appropriate edge e_0 on the boundary B incident with a source or a sink (assume w.l.o.g. that $e_0 = (s_i, v)$);
2. determine an amount D as an appropriate positive number $\leq c(e_0)$;
3. push D units of flow f_i through e_0 ;

¹ We are grateful to them for this.

4. let $N' = (G', P', c')$ be a new $(k + 1)$ -network such that $P' = P \cup \{(s_{k+1}, t_{k+1})\}$, where $s_{k+1} = v$ and $t_{k+1} = t_i$; $c'(e) = c(e)$ if $e \in E - e_0$, and $c'(e_0) = c(e_0) - D$; and $G' = (V, E')$ where $E' = E$ if $c'(e_0) > 0$, or $E' = E - e_0$ otherwise;
5. define the new demands d'_j ($1 \leq j \leq k + 1$) as follows: $d'_i = d_i - D$, $d'_{k+1} = D$, and $d'_j = d_j$ if $1 \leq j \leq k$ and $j \neq i$;
6. apply DELTAFLOW recursively to the new network N' to find $(k + 1)$ -commodity flows f_1, f_2, \dots, f_{k+1} of demands d'_1, \dots, d'_{k+1} in N' ;
7. superimpose three flows, f_i, f_{k+1} and the D units of f_i pushed through edge e_0 , into a new single flow f_i , that is, define $f_i: E \rightarrow R$ as follows:

$$\begin{aligned} f_i(e) &:= f_i(e) + f_{k+1}(e) && \text{if } e \in E - e_0, \\ f_i(e_0) &:= f_i(e_0) + f_{k+1}(e_0) \pm D && \text{if } e_0 \in E', \\ f_i(e_0) &:= \pm D && \text{if } e_0 \notin E', \end{aligned}$$

where the sign \pm depends on the orientation of e_0 ;

8. output f_1, f_2, \dots, f_k as the k -commodity flows in N ;
- end.**

See Fig. 1 for an illustration. If two units of flow f_1 are pushed through e_0 , then the 3-network in Fig. 1(a) results in a new 4-network in Fig. 1(b).

The new planar network N' needs to satisfy the cut condition for the new demands d'_1, \dots, d'_{k+1} so that algorithm DELTAFLOW works well. The next lemma gives the requirement for value D , where we define $m(e_0; Q)$ for a path Q on B as follows: $m(e_0; Q) = \min \{m(e_0, e) | e \in Q\}$ if Q is not empty; $m(e_0; Q) = \infty$ otherwise.

LEMMA 4. (a) Let N be a planar network satisfying the cut condition, and let edge $e_0 = (s_i, v) \in B$ be incident with a source s_i of demand d_i . Let Q be the path on B joining v and t_i and not containing edge $e_0 = (v, s_i)$. If

$$D = \min \{c(e_0), d_i, m(e_0; Q)/2\},$$

then network N' satisfies the cut condition for demands d'_1, \dots, d'_{k+1} . (For N in Fig. 1(a) $D = \min \{3, 3, \frac{4}{2}\} = 2$.)

(b) For every $X \subset V$ the margin $m(X)$ does not increase during the execution of DELTAFLOW(N). In particular, once $m(X)$ becomes 0, it remains unchanged thereafter.

Proof. Assume that e_0 is not deleted, that is, $E' = E$. (The proof for the other case is similar.) We show that $m(X) = c(X) - d(X)$ remains nonnegative for every $X \subset V$ such that $|E(X) \cap B| = 1, 2$. Clearly $c(X)$ or $d(X)$ changes the value only if $e_0 \in E(X)$. We can assume without loss of generality that $t_i \notin X$; otherwise consider $m(V - X)$. Denote by $m'(X)$ the margin in the new network N' . If $s_i \notin X$ and $v \in X$,

$$m'(X) = (c(X) - D) - (d(X) + D) = m(X) - 2D.$$

If $s_i \in X$ and $v \notin X$,

$$m'(X) = (c(X) - D) - (d(X) - D) = m(X).$$

Thus $m(X)$ never increases. It decreases (exactly by $2D$) only if there exists an edge $e \in Q$ such that $E(X) \cap B = \{e_0, e\}$. Hence we have $m'(X) \geq 0$ since $m(X) \geq 0$ and

$$D = \min \{c(e_0), d_i, m(e_0; Q)/2\}.$$

The first two terms of the right-hand side above are necessary to guarantee that $c'(e_0)$ and d'_i are nonnegative. Q.E.D.

Although we can show that there exists $e_0 = (v_j, v_{j+1}) \in B$ such that $D > 0$ and $v_j = s_i$ or t_i for some i and j , we do not prove it here. Instead we will prove a stronger result, Lemma 6, in § 6.

If the first two lines of DELTAFLOW are refined as above, the obtained multicommodity flows satisfy the capacity and conservation rules and have the given demands whenever the algorithm terminates. The sum of capacities over all the edges in G' decreases by D units compared with that in G . However this fact does not immediately imply that DELTAFLOW terminates finitely or within polynomial time. Thus there are three obstructions to the correctness or polynomial boundedness of DELTAFLOW:

- (1) If edge e_0 is selected arbitrarily, DELTAFLOW does not always terminate in polynomial time: when there exists an edge e of infinite capacity on the boundary B , DELTAFLOW possibly lets a flow go and return infinitely many times through e .
- (2) The number of source–sink pairs increases, and the representation of flow functions would require much space.
- (3) The new graph G' may be disconnected even if G is connected.

5. Algorithm MULTIFLOW. In this section we give an algorithm MULTIFLOW for finding multicommodity flows in planar networks, which spends $O(kn + n^2(\log n)^{1/2})$ time and $O(kn)$ space. MULTIFLOW uses the operation of DELTAFLOW, but is improved on the three obstructions mentioned at the end of the preceding section in the following way.

Obstruction (1). In order to make the algorithm run in polynomial time, we select both edge e_0 and flow f_i which is pushed through e_0 , as follows. First select an arbitrary edge, say $e_0 = (v_0, v_1)$, on B ; apply the operation of DELTAFLOW for e_0 with respect to each flow having v_0 as a source (or sink), in the order that the corresponding sink (or source) appears on B in clockwise order. For details, see procedure PUSH (N, e_0) given later in this section. Next select the edge *clockwise* next to e_0 on B as the new e_0 , and apply the same procedure as above for the new e_0 . Repeat this procedure until there exists no source–sink pair. We will show later in § 6 that MULTIFLOW terminates before e_0 traverses all the edges once in each of its two directions. Thus we can guarantee the polynomial time boundedness of MULTIFLOW.

Obstruction (2). Although MULTIFLOW also makes a new source–sink pair (s_{k+1}, t_{k+1}) , it does not introduce the new flow function f_{k+1} , but simply attaches to the pair a number indicating the kind of commodity between s_{k+1} and t_{k+1} . As shown later in the succeeding section, the number of source–sink pairs is at most $O(k+n)$ throughout the execution of MULTIFLOW.

Obstruction (3). If graph G' is disconnected by the deletion of edge e_0 , then we find multicommodity flows in each connected component of G' . Note that G' consists of two components.

We are now ready to present algorithm MULTIFLOW which improves on the three obstructions.

procedure MULTIFLOW (N);

begin

if there exist $e, e' \in B$ with $m(e, e') < 0$

then print “No multiflows of demands” and stop;

for each $e \in E$ and $i (1 \leq i \leq k)$ **do** $f_i(e) := 0$; {initialization}

$p := k$; {the number of source–sink pairs}

$e_0 :=$ an arbitrary edge (v_0, v_1) on B ;

 ROTATE (N, e_0)

end.

```

procedure ROTATE ( $N, e_0$ );
begin
  {this procedure rotates edge  $e_0$  around  $B$  in clockwise order}
  delete all source–sink pairs  $(s_i, t_i)$  such that  $s_i = t_i$  or  $d_i = 0$ ;
  if network  $N$  has a source–sink pair then
    begin
      PUSH ( $N, e_0$ );
      {procedure PUSH ( $N, e_0$ ) pushes flows through  $e_0 = (v_0, v_1)$  by applying
      the operation of DELTAFLOW to each flow having  $v_0$  as a source, in the
      order that the corresponding sink appears on  $B$  in clockwise order. Note
      that  $c(e_0)$  may decrease when the procedure terminates.}
      if  $c(e_0) > 0$  then
        begin
           $e_0 :=$  the edge clockwise next to  $e_0$  on  $B$ ;
          ROTATE ( $N, e_0$ )
        end
      else  $\{e_0$  is saturated, i.e.,  $c(e_0) = 0.\}$ 
        begin
           $G := G - e_0$ ; {delete  $e_0$  from  $G$ }
          if  $G$  is connected then
            begin
              let  $e_0$  be the edge on the new boundary  $B$  of  $G$ , joining  $v_0$  and
              the clockwise next vertex on  $B$ ;
              ROTATE ( $N, e_0$ )
            end
          else for each connected component  $G_j$  ( $j = 1, 2$ )
            of  $G$  do
              begin
                let  $N_j$  be the subnetwork of  $N$  with graph  $G_j$ ;
                let  $e_0$  be the edge on the boundary of the outer face  $B_j$  of  $G_j$ ,
                joining  $v_0$  or  $v_1$  and the clockwise next vertex on  $B_j$ ;
                ROTATE ( $N_j, e_0$ )
              end
            end
          end
        end
      end;

```

Before presenting procedure PUSH (N, e_0), consider how to decide which flows and what amounts can be pushed through e_0 . Let $e_0 = (v_0, v_1)$. Suppose that sources (or sinks) s_1, s_2, \dots, s_l are assigned to v_0 , and that the corresponding sinks (or sources) are t_1, t_2, \dots, t_l , taken in clockwise order from v_1 . Let Q_1 be the path on B clockwise going from v_1 to t_1 , and Q_i the path from t_{i-1} to t_i ($2 \leq i \leq l$). (See Fig. 3.) Note that Q_i is possibly empty. Given $m(e_0, e')$ for all $e' \in B$, one can compute $m(e_0, Q_1)$, and so decide amount D_1 of the flow between s_1 and t_1 to be pushed through e_0 , in $O(|Q_1| + 1)$ time (See Lemma 4(a)). When D_1 units of the flow are pushed through e_0 , $m(e_0, e')$ decreases by $2D_1$ if $e' \in Q_1$, and remains unchanged if $e' \in Q_2 \cup \dots \cup Q_l$. (See the proof of Lemma 4.) In order to decide amount D_2 of the flow between s_2 and t_2 to be pushed through e_0 , a trivial algorithm alters the values $m(e_0, e')$ for $e' \in Q_1$ and finds the minimum of $m(e_0, e')$ over all $e' \in Q_1 \cup Q_2$, so it spends $O(|Q_1| + |Q_2| + 1)$ time. Thus a straightforward algorithm, repeating this procedure, would require $O(n^2)$

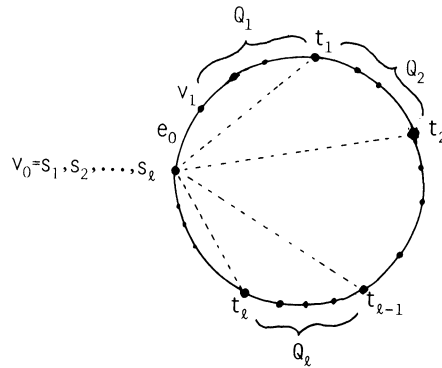


FIG. 3. Illustration for Q_1, Q_2, \dots, Q_l .

time to decide all D_i . However, by updating the value $x(i) = \min \{c(e_0), m(e_0, e')/2 | e' \in Q_1 \cup \dots \cup Q_i\}$ from $x(i-1)$, one can decide each $D_i, 1 \leq i \leq l$, in only $O(|Q_i|+1)$ time as shown in PUSH (N, e_0) below. Thus, given $m(e_0, e')$ for all $e' \in B$, one can decide all $D_i, 1 \leq i \leq l$, in $O(n+l)$ time, since

$$\sum_{i=1, \dots, l} (|Q_i|+1) \leq |B|+l = O(n+l).$$

procedure PUSH (N, e_0) ;

begin

{this procedure pushes flows through edge $e_0 = (v_0, v_1)$ }

$x := \min \{c(e_0), m(e_0; Q_1)/2\}; \{x = x(1)\}$

$i := 1$

while $(x > 0)$ **and** $(i \leq l)$ **do**

if $d_i \leq x$ **then** {the flow between s_i and t_i can be entirely pushed through e_0 }

begin

$D := d_i$;

$s_i := v_1$; {surrogate source}

$c(e_0) := c(e_0) - D_i$; {residual capacity}

{update $x(i+1)$ from $x(i)$ }

if $i = l$ **then** $x := 0$

else $x := \min \{x - d_i, m(e_0; Q_{i+1})/2\}$;

$i := i + 1$ {clockwise next pair}

end;

else {the flow between s_i and t_i can be partly pushed through e_0 }

begin

$D_i := x$;

$d_i := d_i - D_i$; {residual demand}

$c(e_0) := c(e_0) - D_i$;

create a new source-sink pair (s_{p+1}, t_{p+1}) of the same commodity as that between s_i and t_i where $s_{p+1} = v_1$ and $t_{p+1} = t_i$;

$d_{p+1} := D_i$; {demand of new pair}

$p := p + 1$; {the number of pairs increases by one}

$x := 0$;

$i := i + 1$

end;

```

j := i - 1; {the first j flows can be pushed through edge e_0}
for i := 1 to j do
  begin
    let (s_i, t_i) be a source-sink pair for commodity r, where 1 ≤ r ≤ k;
    f_r(e_0) := f_r(e_0) ± D_i;
    {the sign ± depends on both the orientation of edge e_0 and whether v_0 is
     source s_i or sink t_i}
  end;
end;

```

6. Time and space of MULTIFLOW. In this section we first establish our claim on the time and space complexity of algorithm MULTIFLOW. Then the correctness of MULTIFLOW immediately follows from Lemma 4(a).

6.1 Some lemmas. When procedure PUSH (N, e_0) is executed with $e_0 = (v_0, v_1)$, one of the following three cases occurs depending on which term attains the minimum in the equation $D = \min \{c(e_0), d_i, m(e_0; Q)/2\}$ (see Lemma 4(a)). Either (1) edge e_0 becomes saturated and will be deleted by ROTATE, or (2) the $s_i - t_i$ flow of demand d_i is entirely pushed through e_0 , so source s_i disappears from v_0 and a “surrogate source” s_i is constructed at v_1 , or (3) a “bottleneck” edge e is known to exist on B somewhere between v_1 and t_i , that is, there exists $X \subset V$ such that $m(X) = 0$, $v_1 \in X$, and $E(X) \cap B = \{e_0, e\}$.

LEMMA 5. *Suppose that there exists $X \subset V$ such that $m(X) = 0$, $v_0 \in X$, and $E(X) \cap B = \{(v_b, v_0), (v_b, v_{i+1})\}$ ($i < b$). Denote by R the path on B going from v_0 to v_{i+1} in clockwise order. If procedure MULTIFLOW (N) is executed with first assigning edge (v_0, v_1) to variable e_0 , then at least one edge on R is deleted on a first traversal of R .*

Proof. Assume that there is network N for which the lemma is not true for some $X \subset V$, that is, case (1) never occurs on a first traversal of R . Moreover assume that X is minimum in cardinality among all these X 's. Suppose that case (3) never occurs. Then there would be no source or sink of a positive demand in X , so $d(X) = 0$ when PUSH ($N, (v_b, v_{i+1})$) terminates. By Lemma 4(b) $m(X) = 0$ at that time since $m(X) = 0$ when procedure MULTIFLOW (N) started. However $c(X) > 0$ since no edge on R is deleted. This is a contradiction. Thus we have shown that case (3) occurs for an edge on R .

Assume that $e = (v_j, v_{j+1})$ is the first one of such edges on R , where $0 \leq j \leq i$. Then all the vertices v_0, \dots, v_{j-1} are not sources or sinks of positive demands, and there exists $Y \subset V$ such that $m(Y) = 0$, $v_{j+1} \in Y$ and $E(Y) \cap B = \{e, e'\}$ for some $e' = (v_b, v_{l+1}) \in B$ with $l < b$. Furthermore we may assume that Y contains no sources or sinks corresponding to sinks or sources assigned to v_j . Therefore $d(X - Y, Y - X) = 0$. Combining the equation with $m(X) = m(Y) = 0$ through Lemma 2, we have

$$m(X \cap Y) + m(X \cup Y) = -2c(X - Y, Y - X) \leq 0.$$

On the other hand Lemma 4(a) implies that the current network N satisfies the cut condition, and hence $m(X \cap Y), m(X \cup Y) \geq 0$. If $j = i$, then $c(X - Y, Y - X) \geq c((v_b, v_{i+1})) > 0$, contradicting the equation above. Thus $j < i$. Then $\emptyset \neq X \cap Y \subset X - \{x_j\}$ and $m(X \cap Y) = 0$. Therefore the assumption on the minimality of X implies that at least one edge is deleted on the first traversal of the path on B clockwise going from v_{j+1} to the first vertex not in $X \cap Y$. Clearly this edge is on R , a contradiction. Q.E.D.

LEMMA 6. *Algorithm MULTIFLOW (N) assigns no single edge to the variable e_0 more than once for each of its two orientations.*

Proof. Assume that $N = (G, P, c)$ is a network for which the lemma is not true, and that G has a minimum number of edges among such networks; clearly the number is positive. We may assume without loss of generality that edge (v_0, v_1) is first assigned to e_0 .

We now show that at least one edge, other than the last edge (v_b, v_0) , is deleted on a first traversal of B . Suppose to the contrary that no edge is deleted. If case (3) never happens for any edge in $B - \{(v_b, v_0)\}$, then MULTIFLOW (N) would terminate before the first traversal of $B - \{(v_b, v_0)\}$ has been completed, contrary to the assumption. Thus case (3) occurs for an edge $e = (v_j, v_{j+1})$ on $B - \{(v_b, v_0)\}$. Then there exists $X \subset V$ such that $m(X) = 0$, $v_{j+1} \in X$ and $E(X) \cap B = \{(v_j, v_{j+1}), (v_b, v_{i+1})\}$ ($l < b$). Denote by R the path on B going from v_{j+1} to v_{i+1} in clockwise order. By Lemma 5 at least one edge must be deleted on the first traversal of path R . This contradicts the supposition.

Let $e = (v_i, v_{i+1}) \in B$ be the first edge deleted from the graph, where $i < b$. Note that MULTIFLOW (N) has assigned to e_0 each of edges $(v_0, v_1), \dots, (v_i, v_{i+1})$ once so far. Assume that network N results in $N' = (G', P', c')$ when procedure PUSH (N, e) finishes, where $G' = G - e$. Then the following two cases happen.

Case 1. G' is connected. Reapply procedure MULTIFLOW to the new network $M = N'$ by first assigning (v_0, v_1) to e_0 , and consider the behavior by dividing the time period into two parts: (a) while e_0 is $(v_0, v_1), \dots$, or (v_{i-1}, v_i) ; (b) after e_0 becomes (v_i, u) , where u is the vertex clockwise next to v_i on the new outer boundary of G' .

Consider the period (a). By Lemma 4(b) margins have never increased in MULTIFLOW (N) . Since $e \neq (v_b, v_0)$, no "surrogate" source or sink has been constructed at v_0 in MULTIFLOW (N) . Therefore no flow can be pushed through edges $(v_0, v_1), \dots, (v_{i-1}, v_i)$ in MULTIFLOW (M) . Therefore the network M is not altered at all during the period (a), that is, $M = N'$ when e_0 becomes (v_i, u) .

Thus during (b) the behavior of MULTIFLOW (M) is identical with that of MULTIFLOW (N) . Since the number of edges of G' is one less than that of G , the assumption of the minimality of G implies that MULTIFLOW (M) assigns no single edge of G' to e_0 more than once for each of its orientations. Hence MULTIFLOW (N) assigns no single edge of G to variable e_0 more than once for each of its orientations, contrary to the assumption.

Case 2. G' is disconnected. Let G_1 and G_2 be the two connected components in G' . We may assume that G_1 contains v_i and G_2 v_{i+1} . Let $N_i = (G_i, P_i, c_i)$ be the resulting networks ($i = 1, 2$). The behaviour of procedure MULTIFLOW (N) is identical with a combination of two behaviors: the behavior of MULTIFLOW (N_1) beginning at (v_0, v_1) ; and the behavior of MULTIFLOW (N_2) beginning at (v_{i+1}, v_{i+2}) . Since both G_1 and G_2 have fewer edges than G , MULTIFLOW (N_1) and MULTIFLOW (N_2) assign no single edge of G_1 or G_2 to e_0 more than once for each of its orientations, contrary to the assumption. Q.E.D.

Lemma 6 guarantees the polynomial boundedness and so termination of MULTIFLOW.

Figure 4 illustrates a partial traversal of variable e_0 in the network N of Fig. 1(a). The deleted edges are drawn in dashed lines. Number i and an arrow next to an edge indicate that MULTIFLOW (N) assigns the edge to e_0 for the orientation of the arrow in the i th execution of ROTATE (and PUSH). An edge has been assigned to e_0 once for each of the two orientations.

6.2 Data structure and space. A graph G is represented by the adjacency lists, in each of which the edges adjacent to a vertex are stored in the order of the planar

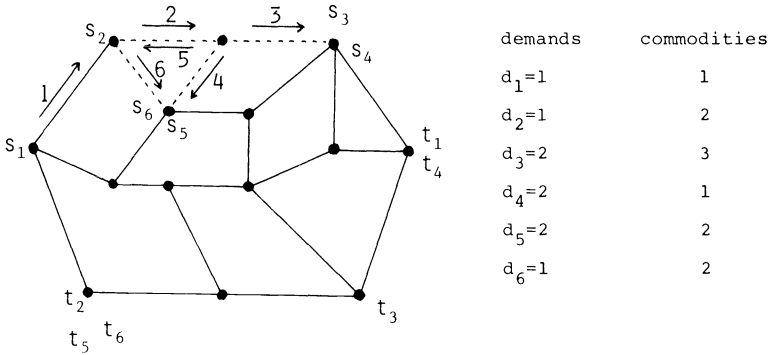


FIG. 4. A network after the first six executions of PUSH.

embedding, clockwise around the vertex. Thus, given an edge e , the edge clockwise next to e around an end of e can be directly accessed. Since G is planar, $|E| = O(n)$. Therefore G can be represented in $O(n)$ space.

Each of the k flow functions f_1, f_2, \dots, f_k is represented by an array of size $|E|$. Thus the representation of k -commodity flows uses $O(kn)$ space.

The set P of source-sink pairs is represented by a multigraph $G_P = (V(B), P)$, each edge of which corresponds to a source-sink pair. Two numbers are associated with each pair (s_i, t_i) : a real-valued demand d_i , and an integer $r(i)$ ($1 \leq r(i) \leq k$) indicating the kind of commodity between s_i and t_i . As shown in Lemma 7 below, the number of source-sink pairs is at most $O(k+n)$ throughout the execution of MULTIFLOW. Thus G_P is represented in $O(k+n)$ space. Hence MULTIFLOW uses $O(kn)$ space in total.

LEMMA 7. *The number of source-sink pairs is at most $O(k+n)$ throughout the execution of MULTIFLOW.*

Proof. Consider procedure PUSH (N, e_0) . Let j be the integer decided in PUSH (N, e_0) . For every i , $1 \leq i \leq j-1$, s_i is moved to v_1 . Only for j may a new source-sink pair be created. Therefore one execution of PUSH (N, e_0) increases the number of source-sink pairs at most one. Lemma 6 implies that MULTIFLOW calls PUSH (N, e_0) at most $2|E|$ times in total. Thus we have established the claim. Q.E.D.

6.3 Computation time. Consider the computation time for one execution of procedure PUSH (N, e_0) . We must compute $m(e_0, e')$ for all $e' \in B$. As shown in § 3, one can compute them in $O(k+n(\log n)^{1/2})$ time by applying one of Frederickson's shortest path algorithms to the dual of G . We have shown in the preceding section that, given $m(e_0, e')$ for all $e' \in B$, one can decide all D_i , $1 \leq i \leq j$, in $O(n+l)$ time. By Lemma 7 l is at most $O(k+n)$. Clearly, PUSH (N, e_0) updates $c(e_0)$ and $f_r(e_0)$ in $O(l) = O(k+n)$ time. Hence we have shown that one execution of PUSH (N, e_0) can be done in $O(k+n(\log n)^{1/2})$ time.

Since MULTIFLOW calls PUSH (N, e_0) $O(n)$ times by Lemma 6, PUSH (N, e_0) spends $O(kn+n^2(\log n)^{1/2})$ time in total. MULTIFLOW (N) decides the feasibility in $O(n^2 \log^* n)$ time as shown in § 3, and the **for** statement in MULTIFLOW spends $O(kn)$ time. The remaining time is for procedure ROTATE (N, e_0) . One execution of ROTATE (N, e_0) can be done in at most $O(n)$ time, exclusive of the time spent by PUSH (N, e_0) called there. Since ROTATE (N, e_0) is called $O(n)$ times by Lemma 6, it spends $O(n^2)$ time in total. Thus we have shown that Algorithm MULTIFLOW spends $O(kn+n^2(\log n)^{1/2})$ time in total.

We now have the following theorem.

THEOREM 2. *Algorithm MULTIFLOW correctly finds multicommodity flows of given demands in a planar network $N = (G, P, c)$ if all the sources and sinks are on the boundary of the outer face of a planar graph G . It spends $O(kn + n^2(\log n)^{1/2})$ time and $O(kn)$ space if there are n vertices and k source-sink pairs.*

7. Conclusion. We have presented an efficient algorithm for finding multicommodity flows in a planar undirected graph, which spends $O(kn + n^2(\log n)^{1/2})$ time and $O(kn)$ space if a graph has n vertices and k source-sink pairs. It is interesting that the values of the obtained flows are half integers if the capacities and demands are all integers. Using our algorithm, one can design a heuristic algorithm for finding multicommodity flows in a general planar graph in which not all sources and sinks are on the boundary of the outer face. We expect to examine the practicality of the heuristics. Finally we remark: neither a claim similar to Theorem 1 holds for planar *directed* graphs, nor our algorithm correctly works for them, even in the case of two-commodity flows. Figure 5 depicts a planar directed graph in which all sources and sinks lie on the outer boundary. Although the graph satisfies the cut condition of the directed version, it has no two-commodity flows realizing the given demands.

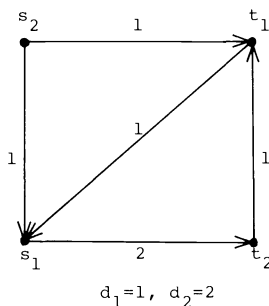


FIG. 5. A planar directed graph. (The number associated with each edge represents the capacity of the edge.)

Acknowledgment. We thank the referees for their technical and grammatical comments which helped us to shorten and improve this paper.

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. DIAZ AND G. DE GHELLINCK, *Multicommodity maximum flow in planar networks (the D-algorithm approach)*, CORE discussion paper No. 7212, Center for Operations Research and Econometrics, Louvain-la-Neuve, Belgium, 1972.
- [3] E. W. DIJKSTRA, *A note on two problems in connexion with graphs*, Numer. Math., 1 (1959), pp. 269–271.
- [4] L. R. FORD AND D. R. FULKERSON, *Maximal flow through a network*, Canad. J. Math., 8 (1956), pp. 399–404.
- [5] G. N. FREDERICKSON, *Shortest path problems in planar graphs (preliminary version)*, Proc. 24th Symposium on Foundation of Computer Science, Tucson, Nov. 1983, pp. 242–247.
- [6] R. HASSIN, *Maximum flow in (s, t) planar networks*, Inform. Proc. Lett., 13 (1981), p. 107.
- [7] A. ITAI, *Two-commodity flow*, J. Assoc. Comput. Mach., 25 (1978), pp. 596–611.
- [8] A. ITAI AND Y. SHILOACH, *Maximum flows in planar networks*, this Journal, 8 (1979), pp. 135–150.
- [9] L. G. KHACHIYAN, *A polynomial algorithm in linear programming*, Dokl. Akad. Nauk SSSR N. S. 244:5 (1979), pp. 1093–1096 [English transl., Soviet Math. Dokl. 20:1 (1979), pp. 191–194].

- [10] H. OKAMURA AND P. D. SEYMOUR, *Multicommodity flows in planar graphs*, J. Combin. Theory B, 31 (1981), pp. 75–81.
- [11] M. SAKAROVITCH, *The multicommodity flow problem*, Doctoral thesis, Operations Research Center, Univ. California, Berkeley, 1966.
- [12] ———, *Two commodity network flows and linear programming*, Math. Prog., 4 (1973), pp. 1–20.
- [13] D. D. SLEATOR, *An $O(nm \log n)$ algorithm for maximum network flow*, Ph.D. dissertation, Computer Science Dept., Stanford Univ., Stanford, CA, 1980.
- [14] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 26 (1983), pp. 362–390.
- [15] D. T. TANG, *Bi-path networks and multicommodity flows*, IEEE Trans. Circuit Theory, CT-11 (1964), pp. 468–474.

TRADE-OFFS BETWEEN DEPTH AND WIDTH IN PARALLEL COMPUTATION*

UZI VISHKIN† AND AVI WIGDERSON‡

Abstract. A new technique for proving lower bounds for parallel computation is introduced. This technique enables us to obtain, for the first time, nontrivial tight lower bounds for shared-memory models of parallel computation that allow several processors to have simultaneous access to the same memory location. Specifically, we use a concurrent-read concurrent-write model of parallel computation. It has p processors, each has access to a common memory of size m (also called *communication width* or *width* in short). The input to the problem is located in an additional read-only portion of the common memory.

For a wide variety of problems (including parity, majority and summation) we show that the time complexity T (depth) and the communication width m are related by the trade-off curve $mT^2 = \Omega(n)$, (where n is the size of the input), *regardless* of the number of processors. Moreover, for every point on this curve with $m = O(n/\log^2 n)$ we give a matching upper bound with the *optimal* number of processors.

We extend our technique to prove $mT^3 = \Omega(n)$ trade-off for a class of "simpler" functions (including Boolean OR) on a weaker model that forbids simultaneous write access. We also state and give a proof of a new result by Beame [B-83] that achieves a tight lower bound for the OR in this model, namely $mT^2 = \Omega(n)$. These results improve the lower bound of Cook and Dwork [CD-82] when communication is limited.

Key words. synchronous parallelism, parallel time complexity, communication width, trade-offs between complexity measures, lower bounds

1. Introduction. Consider the following informal problem: there are a large number of people (or processing units), each knows n numbers a_1, a_2, \dots, a_n . They all wish to compute the sum of these numbers. If they cannot communicate, there is no way to avoid sequential ($\Omega(n)$ time) summation by each person separately. On the other hand, it is shown in the paper that with only one communication channel (one cell of shared memory) this time can be reduced to $O(\sqrt{n})$. With n (resp. 2^n) shared memory cells the time can be reduced further to $O(\log n)$ (resp. $O(1)$). This exemplifies that a communication facility is essential for any utilization of parallelism, and that its size directly affects the performance of the algorithm.

The size of the common memory required by a given parallel algorithm will be determined by two principal factors.

(a) *Input availability.* The size of the input, in the case that the input is placed in the common memory, or the need to transfer input data in the case that the input is initially distributed among the local memories.

(b) *Cooperation between processors.* The transmission of intermediate results between processors, utilized to obtain fast processing time.

Here we propose to concentrate on point (b). For this reason we put the input in a "read only" common memory.

In this paper we will concentrate on parallel RAMs (PRAMs), in particular, the Concurrent-Read Concurrent-Write PRAM (CRCW PRAM) and the Concurrent-

* Received by the editors May 17, 1983, and in revised form November 15, 1983. This paper is based on *Trade-offs between depth and width in parallel computation* by U. Vishkin and A. Wigderson, appearing in the 24th Annual Symposium on Foundations of Computer Science, November 7-9, 1983, Tucson, AZ, pp. 146-153. © 1983 IEEE.

† Department of Computer Science, School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel. This research was conducted while this author was at Courant Institute, New York University, New York, New York.

‡ Computer Science Division, University of California, Berkeley, California 94720. This research was conducted while this author was in the Electrical Engineering and Computer Science Departments, Princeton University, Princeton, New Jersey.

Read Exclusive-Write PRAM (CREW PRAM). Both models are precisely defined in § 2. In the above models processors communicate via a shared memory. Therefore the size of the communication facility of the machine, here called *communication width* (or *width* in short) is simply the number of shared memory cells. We consider the width m a resource, together with the size p (the number of processors) and the depth T (the running time), and we seek trade-offs between the three.

One of the subtleties in proving lower bounds for these models, is that information may be communicated by the fact that no processor writes into a common memory cell. We introduce a novel technique to deal with this difficulty.

For a large class of functions, which includes Parity and Majority, we prove $T = \Omega((n/m)^{1/2})$ on the CRCW PRAM, where n is the size of the input. This lower bound is tight for all values of width $m = O(n/\log^2 n)$. This is the first time nontrivial tight lower bounds are achieved for a model that allows concurrent write access. The only known lower bound on the CRCW PRAM model is given in Stockmeyer and Vishkin [SV-82]. They show, using a result of Furst, Saxe, and Sipser [FSS-81], that it is impossible to compute parity in this model in constant time using a polynomial number of processors. There is, however, a large gap between this lower bound and the best upper bound known for a polynomial number of processors, which is $O(\log n / \log \log n)$. (See [CSV-82]).

For another class of functions, which includes the functions AND and OR, we prove a lower bound of $T = \Omega((n/m)^{1/3})$ on the CREW PRAM. This lower bound extends the $\Omega(\log n)$ of Cook and Dwork for small values of m , and further discerns the power of CRCW PRAM from the CREW PRAM. At this point we state, and give a proof, of a new result by Beame that achieves a tight lower bound for computing the OR in this model. For a different class of functions (that include OR) he proves $T = \Omega((n/m)^{1/2})$.

Both our lower bounds hold regardless of the number of processors, while the upper bounds are achieved with the smallest possible number of processors.

Our study of values of m which are smaller than input size requires us to add a read only input tape to the model, as is done in the study of space bounded Turing machines. The interest in those values is not solely theoretical—it is well founded in practice. For example the “Ethernet” can be considered as a PRAM with only one shared memory cell. Also, the papers Gottlieb et al. [GGKMRS-82], Kuck [K-77] and Vishkin [V-82] imply that minimizing the size of shared memory (that can be accessed in parallel) may amount to hardware feasibility of the parallel machine.

The paper is organized as follows: precise definitions and the lower bounds are given in § 2. Section 3 contains the upper bounds and § 4 concludes the paper and suggests further research directions. To improve the readability of § 2, some of the proofs were deferred to the appendix.

2. Lower bounds. In the first subsection we give precise definitions of the models of computation when the communication width $m = 1$, and of the types of functions we are interested in. Subsections 2.2 and 2.3 contain the lower bound proofs for the concurrent-write and exclusive-write models respectively when $m = 1$. In the last subsection we show how to extend the lower bounds for arbitrary communication width.

2.1. Definitions.

DEFINITION 2.1. A CRCW PRAM (1) consists of a set $\Pi = \{p_1, p_2, \dots\}$ of processors, a number n of inputs, n read-only input cells $X(1), X(2), \dots, X(n)$, one common memory cell C , an alphabet Σ and an execution time T .

Each processor p_i has a set of states Q_i and functions

$\rho_i: Q_i \rightarrow \{1, 2, \dots, n\}$ —the next input cell to be read,

$\sigma_i: Q_i \rightarrow \Sigma$ —the symbol to be written into C , and

$\delta_i: Q_i \times \Sigma \times \Sigma \rightarrow Q_i$ —the state transition function.

At each time period $t=0, 1, \dots, T$ each processor p_i is in a state $q_i^t \in Q_i$, and the cell C contains a symbol $s^t \in \Sigma$. At time $t=0$ the input cell $X(i)$ contains the input $x_i (\in \Sigma) (1 \leq i \leq n)$, the cell C contains a designated symbol $b_0 \in \Sigma$, and every processor p_i is in an initial state $q_i^0 \in Q_i$. In general

$$q_i^{t+1} = \delta_i(q_i^t, X(j), s^t), \text{ where } j = \rho_i(q_i^t), \text{ and}$$

$$s^{t+1} = \begin{cases} s^t & \text{if for every } i \sigma_i(q_i^{t+1}) = s^t \text{ (no one writes),} \\ \sigma_i(q_i^{t+1}), & i \text{ is the smallest s.t. } \sigma_i(q_i^{t+1}) \neq s^t. \end{cases}$$

The value $f(x_1, x_2, \dots, x_n)$ of the function f computed by the PRAM (1) is the contents s^T of C at time T .

DEFINITION 2.2. A CREW PRAM (1) is defined exactly like the CRCW PRAM (1), with only one exception—at each time period t there can be at most one processor that writes, i.e. at most one i s.t. $\sigma_i(q_i^{t+1}) \neq s^t$.

Remarks. These lower bound models allow Π, Σ and Q_i to be infinite, and allow the processors be nonuniform (i.e. have different programs for different values of n). Also note that we use the convention that a processor *writes* if it tries to *change* the contents of C , (and in the CRCW it must be the one with the smallest serial number doing so).

DEFINITION 2.3. Let Σ be a set, $\mathbf{I} \subseteq \Sigma^n$ and $f: \mathbf{I} \rightarrow \Sigma$ some function. An input $x = x_1, x_2, \dots, x_n \in \mathbf{I}$ is said to be k -sensitive w.r.t. f if for every subset $J \subseteq \{1, 2, \dots, n\}, |J| = k - 1$ there exists another input $y = y_1, y_2, \dots, y_n \in \mathbf{I}$ s.t. $x_j = y_j$ for all $j \in J$, and $f(x) \neq f(y)$. If k is the largest integer s.t. every input (resp. some input) $x \in \mathbf{I}$ is k -sensitive w.r.t. f , then f is said to be k -sensitive everywhere (resp. k -sensitive somewhere).

Examples. Consider the functions Parity, Majority, OR: $\{0, 1\}^n \rightarrow \{0, 1\}$.

Parity is n -sensitive everywhere.

Majority is $\lceil n/2 \rceil$ -sensitive everywhere.

OR is only 1-sensitive everywhere for all n , but it is n -sensitive somewhere (the all zeros input is n -sensitive w.r.t. OR).

2.2. Lower bounds for CRCW PRAM (1).

THEOREM 2.1. Let M be a CRCW PRAM (1) that computes a k -sensitive everywhere function f in time T . Then $T = \Omega(\sqrt{k})$.

COROLLARY 2.1. Let M be a CRCW PRAM (1) that computes the Parity, Majority (Sum, Max) function on n bits (integers) in time T . Then $T = \Omega(\sqrt{n})$.

Proof. Parity, sum and max are n -sensitive everywhere. Majority is $\lceil n/2 \rceil$ -sensitive everywhere. \square

Let us informally discuss the difficulties we are facing in trying to prove Theorem 2.1. Consider the behavior of the machine in time period t . There are two possible cases:

Case 1. No processor writes into C ($s^t = s^{t-1}$.)

Case 2. Some processor (say p_j) writes into C ($s^t \neq s^{t-1}$).

We have to analyze the information that is transferred in each case. Consider first Case 1. As Cook and Dwork [CD-82] point out, information is transferred in this case, namely the information that nobody wrote. They show how this information can be used in an algorithm for the OR function, that is faster than the obvious one. The way they keep track of this elusive information is heavily based on the fact that their

model does not allow simultaneous write access to the same memory cell. (Indeed, their lower bound does not hold for the CRCW PRAM). As our model allows simultaneous write access, we had to choose an approach which is different from theirs.

The information that is transferred in Case 2 seems even more slippery. We know what was written into C , and in addition we know that no processor with serial number smaller than j tried to write. (Note that as Σ may be infinite, the writer can encode its serial number in the symbol it writes.) This case is much simpler in the exclusive write model, since there, if someone writes, there can be no other processor that tries to write!

At this point we need some notation. Let \mathbf{I} denote the (nonempty) set of all possible inputs (the domain). Fix a time period t and let $\beta = s^0 s^1 \cdots s^{t-1}$ be the string of successive symbols in C in time periods $0, 1, \dots, t-1$. β is called the history through time t . Denote by $\mathbf{I}_\beta \subseteq \mathbf{I}$ the subset of inputs that have history β through time t .

Our analysis will be based on the observation that Cases 1 and 2 consist each of two subcases. Fix β , a history through time t .

Case 1a. There is no input in \mathbf{I}_β for which some processor writes at time t .

Case 1b. There is an input in \mathbf{I}_β for which some processor writes at time t .

Case 2a. There is no input in \mathbf{I}_β for which some processor with smaller serial number than j writes at time t .

Case 2b. There is an input in \mathbf{I}_β for which a processor with smaller serial number than j writes at time t .

It turns out that Cases 1a and 2a are simple to analyze. Intuitively, in Case 1a no new information is transferred as β itself contains the information that no one will write at time t . Similarly, in Case 2a, β contains the information that no processor with a smaller serial number than the writer could have written, so the only new piece of information is the new symbol in C , s^t .

Now, rather than confronting the elusive information that is transferred in Cases 1b and 2b, we avoid (or circumvent) it, and hence coin the name *circumvention* for this technique. Showing that we can restrict ourselves to the “easy to analyze” cases is the heart of our argument.

Let $\mathbf{I}(\{(i_1, y_1), \dots, (i_l, y_l)\}) = \{x \in \mathbf{I} \mid x_{i_j} = y_j, 1 \leq j \leq l\}$ denote the set of all inputs (n -tuples) whose projection on the l -tuple (i_1, i_2, \dots, i_l) is (y_1, y_2, \dots, y_l) .

Remark. We switch here from qualifying inputs by their history (“range” qualification) to qualifying them by their values at given coordinates (domain qualification). This yields a simpler and more intuitive proof than our original one which used range qualification. However, we believe that range qualification is more powerful, and that it may be used to prove lower bounds when domain qualification fails.

The following iterative definition will generate an “easy to analyze” set of inputs, i.e. inputs for which Cases 1b and 2b never occur. For every t , D^t will contain pairs of “fixed” input positions and their values, and $E^t = \mathbf{I}(D^t)$.

Let $E^0 = \mathbf{I}$ and $D^0 = \emptyset$. Consider time period t and define E^t, D^t according to the following:

Case 1. There is no processor p_j and no input $x \in E^{t-1}$ such that p_j writes on x at time t . Then

$$E^t \leftarrow E^{t-1}, \quad D^t \leftarrow D^{t-1}.$$

Case 2. There is a processor p_j and an input $x \in E^{t-1}$ s.t. p_j writes on x at time t . Let p_l and $y \in E^{t-1}$ be so that p_l writes on y at time t , and l is the smallest serial number of any processor that writes at time t on any input in E^{t-1} . Let i_1, i_2, \dots, i_u and $y_{i_1}, y_{i_2}, \dots, y_{i_u}$ be the sets of input cells and their contents (respectively) that were

read by p_i up to time t . (Clearly $u \leq t$.) Then

$$E^t \leftarrow E^{t-1} \cap \mathbf{I}(\{(i_1, y_{i_1}), \dots, (i_u, y_{i_u})\}),$$

$$D^t \leftarrow D^{t-1} \cup \{(i_1, y_{i_1}), \dots, (i_u, y_{i_u})\}.$$

It is easy to see that for every $0 \leq t \leq T$

- (1) $|D^t| \leq |D^{t-1}| + t, |D^0| = 0$ and hence $|D^t| \leq t(t+1)/2$.
- (2) $E^t = \mathbf{I}(D^t), D^{t-1} \subseteq D^t, E^t \subseteq E^{t-1}$.
- (3) $E^t \neq \emptyset$.

In particular we have:

LEMMA 2.1. $E^T \neq \emptyset$ and $|D^T| \leq T(T+1)/2$.

Remark. The definition above generates a set E^T of “easy to analyze” inputs, regardless of the function being computed. Therefore we believe that this technique can be used to prove lower bounds for the computation of other functions in this model.

LEMMA 2.2. Let M be an CRCW PRAM (1) computing a function f , and let E^T be defined as above for M . Then for every $x, y \in E^T, f(x) = f(y)$.

A rigorous proof of this lemma is given in the appendix. The idea is to show inductively on t , that any processor which writes at time t on some input in E^T , will have exactly the same computation through time t on every input in E^T .

Proof of Theorem 2.1. Recall that M computes a k -sensitive everywhere function f in time T . Suppose that $T(T+1)/2 < k$. Then $|D^T| > k$, and so by Definition 2.3, there must be inputs x and y in E^T s.t. $f(x) \neq f(y)$. This contradicts Lemma 2.2. Therefore $T(T+1)/2 \geq k$, so $T = \Omega(\sqrt{k})$. \square

2.3. Lower bounds for the CREW PRAM (1). Consider the OR function of n bits. As mentioned earlier, the OR is just 1-sensitive everywhere, so the results in the previous subsection imply only a constant time lower bound for it on the CRCW PRAM (1). Indeed, there is a two step algorithm for the OR on this model as follows. In the first step, the common memory cell C is initialized with “0.” In the second step, a processor p_i reads the i th input position and writes a “1” into C iff the value it read was “1.”

It is clear why this algorithm is not valid for a CREW PRAM. Note, however, that if the domain consists only of inputs which have at most one position containing a “1,” a write conflict cannot occur, and the algorithm is valid for the CREW PRAM. For this reason we will restrict ourselves here to functions with a full domain (i.e. $\mathbf{I} = \Sigma^n$). The main result in this subsection is the following theorem.

THEOREM 2.2. Let N be a CREW PRAM (1) that computes a k -sensitive somewhere function g in time T . Then $T = \Omega(k^{1/3})$.

COROLLARY 2.2. If g is the OR function on n bits, then $T = \Omega(n^{1/3})$.

In an earlier version of this paper we conjectured that the lower bound of Corollary 2.2 can be improved to $T = \Omega(\sqrt{n})$. This was recently proved by Beame [B-83]. In fact, he proved the following stronger theorem.

THEOREM 2.3. (Beame). Let N be a CREW PRAM (1) that computes a function $g: \{0, 1\}^n \rightarrow \{0, 1\}$ in time T . If there exists an input $e \in \mathbf{I}$ s.t. $|\{x \in \mathbf{I}: g(x) = g(e)\}| \leq |\mathbf{I}|/r$, then $T = \Omega(\sqrt{\log_2 r})$.

It immediately follows that:

COROLLARY 2.3 (Beame). If g is the OR function on n bits, then $T = \Omega(\sqrt{n})$.

The proof of Theorem 2.3 has the same structure as that of Theorem 2.2. However, while we focus on the sensitivity of inputs in the lower bound argument, Beame focuses on a different parameter, namely the number of inputs with the same image. His proof is of independent interest, and we include it in the appendix.

We return to the proof of Theorem 2.2. The idea is to use the framework of the previous subsection, namely to construct a set of inputs E^T , and show that for the computed function to be constant on E^T , T must be large. This task was relatively easy for everywhere sensitive functions, since we did not have to worry about the contents of E^T , as every input is sensitive. To use the sensitivity of inputs in a somewhere sensitive function in a similar argument we must make sure that E^T contains at least one sensitive input. This motivates the following inductive definition of the sets D^t, E^t .

Let $g: \mathbf{I} \rightarrow \Sigma$ be the function being computed and $e = e_1, e_2, \dots, e_n \in \mathbf{I}$ be a k -sensitive input w.r.t. g . Set $E^0 = \mathbf{I}$ and $D^0 = \emptyset$. Consider time period t and define E^t, D^t as follows:

Case 1. There is no processor p_j and no input $x \in E^{t-1}$ such that p_j writes on x at time t . Then

$$E^t \leftarrow E^{t-1}, \quad D^t \leftarrow D^{t-1}.$$

Case 2. There is a (unique) processor p_j that writes on $e \in E^{t-1}$ at time t . Let i_1, i_2, \dots, i_u and $e_{i_1}, e_{i_2}, \dots, e_{i_u}$ be the sets of input cells and their contents (respectively) that were read by p_j up to time t . (Clearly $u \leq t$). Then

$$D^t \leftarrow D^{t-1} \cup \{(i_1, e_{i_1}), \dots, (i_u, e_{i_u})\},$$

$$E^t \leftarrow E^{t-1} \cap \mathbf{I}(\{(i_1, e_{i_1}), \dots, (i_u, e_{i_u})\}).$$

Case 3. There exists $x \in E^{t-1}, x \neq e$ s.t. some p_j writes on x at time t , but no processor writes on e at time t . Let R'_0 be a set of positions s.t. if $y \in E^{t-1}$ and $y_i = e_i$ for all $i \in R'_0$, then no processor writes on y at time t . In this case we fix the positions R'_0 with values of e :

$$D^t \leftarrow D^{t-1} \cup \{(i, e_i) | i \in R'_0\},$$

$$E^t \leftarrow E^{t-1} \cap \mathbf{I}(\{(i, e_i) | i \in R'_0\}).$$

It is easy to see inductively that $e \in E^t$ for all t , and so $e \in E^t$. Our main problem is to obtain an upper bound on $|R'_0|$.

LEMMA 2.3. *For every $t, |R'_0| \leq t(t+1)/2$.*

This lemma is the heart of the lower bound. Since the proof is long, it is deferred to the appendix.

LEMMA 2.4. *For every $t, |D^t| \leq t(t+1)(t+2)/6$ and $e \in E^t$.*

Proof. By simple induction on t . \square

LEMMA 2.5. *For every $x, y \in E^T, g(x) = g(y)$.*

Proof. Exactly the same as the proof of Lemma 2.2. \square

Proof of Theorem 2.2. Recall that N computes a k -sensitive somewhere function g in time T . Suppose $T(T+1)(T+2)/6 < k$. Then by Lemma 2.4 $|D^T| > k$. Since $e \in E^T$, by Definition 2.3 there must be a $y \in E^T$ s.t. $g(y) \neq g(e)$, which contradicts Lemma 2.5. \square

2.4. Arbitrary communication width. What happens when the communication width is larger than 1? The CRCW PRAM(m) is defined similarly to the CRCW PRAM(1), only now there are m common memory cells $C(1), C(2), \dots, C(m)$ to which the processors have concurrent read/write access. In a similar fashion the CREW PRAM(m) can be defined. Our results are summarized in the following theorem.

THEOREM 2.4. *Let M be a CRCW PRAM(m) that computes a k -sensitive everywhere function $f: \mathbf{I}(\subseteq \Sigma^n) \rightarrow \Sigma$ in time T . Then $T = \Omega(\sqrt{k/m})$. In particular, if $f \in \{\text{Parity, Majority, Sum, Max}\}$, $T = \Omega(\sqrt{n/m})$. Let N be a CREW PRAM(m) that*

computes a k -sensitive somewhere function $g: \Sigma^n \rightarrow \Sigma$. Then $T = \Omega((k/m)^{1/3})$. In particular, if $g \in \{\text{AND}, \text{OR}\}$, $T = \Omega((n/m)^{1/3})$.

The only difficulty in extending our technique to prove Theorem 2.4 is in the definition of the “easy to analyze” cases. For example, one can construct a machine for which the following happens: There are inputs for which both $C(1)$ and $C(2)$ are written into. However, if we choose an input for which the smallest numbered processor writes into $C(1)$, no one will write into $C(2)$ and vice versa.

We overcome this difficulty by conceptually serializing the write access into different cells as follows: Each time unit t is sliced into m slices, so that in the i th slice only cell $C(i)$ may be written into. Then, at the i th slice of time period t we can refer not only to the contents of all cells at previous time periods, but also to the contents of cells 1 to $i-1$ at time period t . (Note that the machine is not affected by this conceptual slicing. Indeed, it shows that our results hold even in a stronger model that allows the processors to access all common memory cells at each time unit.). As a result we are able to define sets E^t and D^t , $0 \leq t \leq T$, $1 \leq i \leq m$, inductively in a similar fashion to the previous subsections for the CRCW PRAM(m) and the CREW PRAM(m) respectively. The only refinement is that instead of defining E^t from E^{t-1} , we define E^t from E^{t-1} when $i > 1$, and E^t from $E^{(t-1)m}$.

The analysis of the previous subsections carries through in a straightforward manner w.r.t the final sets, E^{Tm} and D^{Tm} . This includes the proof of the following two lemmas and the conclusion of the theorem from them.

LEMMA 2.6. In the CRCW PRAM(m), $|D^{Tm}| \leq m(T+1)T/2$.

In the CREW PRAM(m), $|D^{Tm}| \leq mT(T+1)(T+2)/6$.

LEMMA 2.7. In the CRCW PRAM(m), for every $x, y \in E^{Tm}$, $f(x) = f(y)$.

In the CREW PRAM(m), for every $x, y \in E^{Tm}$, $g(x) = g(y)$.

We conclude this subsection with two observations:

(1) The ideas outlined above can be used to extend also Beame’s theorem (Theorem 2.3) for arbitrary communication width, as follows.

THEOREM 2.5. Let N be a CRCW PRAM(m) that computes a function $g: \{0, 1\}^n \rightarrow \{0, 1\}$ in time T . If there exists an input $e \in I$ s.t. $|\{x \in I: g(x) = g(e)\}| \leq |I|/r$, then $T = \Omega(\sqrt{(\log_2 r)/m})$. In particular, if g is the OR function, then $T = \Omega(\sqrt{n/m})$.

(2) Two other concurrent-write models of parallel computation that appeared in the literature ([SV-81], [ShV-82]). They differ from our CRCW PRAM in the way they resolve write conflicts. In the first all processors that access the same memory location should write the same value. In the second there is no such restriction, but we do not know in advance which processor succeeds in writing. We conclude this section by mentioning that those two models are weaker than ours, and therefore our results for the CRCW PRAM hold for them as well.

3. Upper bounds. All upper bounds can be achieved in the weakest version of a PRAM, namely the Exclusive-Read Exclusive-Write PRAM (EREW PRAM). It is similar to the CREW PRAM, only that in this model any simultaneous access of a shared memory cell is forbidden. The algorithms are simple and will be described informally. They will be given only for the problem of summing n numbers. It is easy to see that they hold for computing any associative function.

Consider first the EREW PRAM(1) model. The n numbers a_1, a_2, \dots, a_n are initially stored in the read-only input tape. Let L_j be a local memory cell of processor p_j and C is the common memory cell. The algorithm is described in Fig. 1.

Clearly, only $p = O(\sqrt{n})$ processors are active in this algorithm, and the sum is computed in $O(\sqrt{n})$ time. Since sequential time for summation is $\Omega(n)$, a

| Time | p_1 | p_2 | p_3 | p_4 | \dots |
|------|--|--|--|---|---------|
| 1 | $L_1 \leftarrow a_1$ $C \leftarrow L_1$ | $L_2 \leftarrow a_2$ | $L_3 \leftarrow a_4$ | $L_4 \leftarrow a_7$ | \dots |
| 2 | | $L_2 \leftarrow L_2 + a_3$ $C \leftarrow C + L_2$ | $L_3 \leftarrow L_3 + a_5$ | $L_4 \leftarrow L_4 + a_8$ | \dots |
| 3 | | | $L_3 \leftarrow L_3 + a_6$ $C \leftarrow C + L_3$ | $L_4 \leftarrow L_4 + a_9$ | \dots |
| 4 | | | | $L_4 \leftarrow L_4 + a_{10}$ $C \leftarrow C + L_4$ | \dots |
| . | | | | | |
| . | | | | | |

FIG. 1. *Summation with one common memory cell.*

straightforward lower bound of $\Omega(n/p)$ exists for any parallel machine with p processors. Hence the number of processors is optimal up to a constant factor.

Consider now the same problem for the CRCW PRAM(m), where $m = O(n/\log^2 n)$. We show how to achieve $O(\sqrt{n/m})$ time with $O(\sqrt{nm})$ processors. The algorithm has two phases:

(1) Partition the n inputs into m subsets of size roughly n/m each. Assign to each subset $\sqrt{n/m}$ processors and one common memory cell. For each subset the sum is computed in the respective memory cell using the algorithm above in time $O(\sqrt{n/m})$.

(2) Sum up the m values in the common memory using $m(\cong \sqrt{nm})$ processors in $O(\log m)$ time in the obvious way.

As before, the number of processors used is optimal up to a constant factor. This upper bound establishes that our lower bound for Parity on the CRCW PRAM(m) and Beame's lower bound for the OR on the CREW PRAM(m) are tight.

We conclude by mentioning what is known when the communication width is larger than the input size. If the input values are taken from a finite domain, the sum can be computed in constant time using exponential width and number of processors. If those two resources are bounded by a polynomial in n , the best upper bound known is $O(\log n / \log \log n)$ [CSV-82].

4. Conclusions and open problems. Using communication based arguments to prove lower bounds in computer science is an old idea. The crossing-sequence [HU-79] technique in Turing machines essentially measures communication between work-tape cells. This technique was extended to measure communication between two halves of a VLSI circuit [Y-81], [LS-81], [PS-82] and obtain Time-Area trade-offs.

We consider this paper to be a first step towards understanding the central role played by communication in efficient parallel computation. The view of communication as a resource in parallel machines gives rise to many questions. We mention a few below.

(1) Our lower bound for the OR on the CREW PRAM, combined with that of Cook and Dwork, covers the whole range of m . On the other hand, the lower bound for the parity functions on the CRCW PRAM(m) becomes trivial when $m \cong n$. The case where m is only bounded by a polynomial in n is of particular interest, since a lower bound on the time here will give a lower bound on the depth of polynomial size parity circuits.

(2) Consider parallel RAMs in which processors are allowed to be probabilistic or nondeterministic. In the deterministic version of the CRCW PRAM (1) which we studied here, both the Parity and the Max functions have an $\Omega(\sqrt{n})$ lower bound on the time. If we allow nondeterminism, the maximum of n numbers can be computed in constant time. However, we conjecture that the lower bound still holds for Parity even in the nondeterministic model.

(3) Study Time–Width–Processors trade-offs for other functions.

Appendix.

LEMMA 2.2. For every $x, y \in E^T$, $f(x) = f(y)$.

Proof of Lemma 2.2. We use the following notation. For an input $x \in \mathbf{I}$,

- $q_i^t(x)$ and $s^t(x)$ are respectively the state of p_i and the contents of C in time t for the input x .
- $R_j^t(x) = \{\rho_j(q_j^r(x)) \mid 0 \leq r < t\}$ is the set of input cells read by p_j through time t . Set $R_0^t(x) = \emptyset$.
- $w^t(x)$ is the index of the processor that writes at time t on input x . If there is no such processor at time t for x , $w^t(x) = 0$.
- $W^t(x) = \bigcup_{r=1}^t R_j^r(x)$ where $j = w^r(x)$, is the set of input cells read by all writers through time t .
- $FW^t(x) = \{w^r(x) \mid t \leq r \leq T, w^r(x) \neq 0\}$ is the set of future writers from time period t on.

Let x and y be elements in E^T . It is sufficient to show that $s^T(x) = s^T(y)$. We prove by induction on t , that $w^t(x) = w^t(y)$, $W^t(x) = W^t(y)$, $s^t(x) = s^t(y)$, and that for every $j \in FW^t(x)$, $q_j^t(x) = q_j^t(y)$ and $R_j^t(x) = R_j^t(y)$.

$t = 0$. For every processor j , $q_j^0(x) = q_j^0(y) = q_j^0$, $R_j^0(x) = R_j^0(y) = \emptyset$. Also, $s^0(x) = s^0(y) = b_0$, $W^0(x) = W^0(y) = \emptyset$ and $w^0(x) = w^0(y) = 0$.

$t > 0$. Assume the claim holds for every $r < t$. Let $j \in FW^t(x)$. Let $i(x) = \rho_j(q_j^{t-1}(x))$, $i(y) = \rho_j(q_j^{t-1}(y))$. By the induction hypothesis $i(x) = i(y)$ and hence $R_j^t(x) = R_j^t(y)$. Since $j \in FW^t(x)$, $R_j^t(x) \subseteq W^t(x)$ which using $x, y \in E^T$ implies that $x_{i(x)} = y_{i(x)}$. From this and the induction hypothesis we get $q_j^t(x) = q_j^t(y)$. Let $\sigma_j(x) = \sigma_j(q_j^t(x))$ and $\sigma_j(y) = \sigma_j(q_j^t(y))$. Then $\sigma_j(x) = \sigma_j(y)$. There are two cases to consider now,

Case 1. $s^t(x) = s^{t-1}(x)$. By the construction of E^t and induction, $s^t(y) = s^t(x) = s^{t-1}(x)$, $w^t(x) = w^t(y) = 0$ and $W^t(x) = W^t(y) = W^{t-1}(x)$.

Case 2. $s^t(x) \neq s^{t-1}(x)$. Let $j = w^t(x)$. Again by construction of E^t , there can be no $l < j$ s.t. $\sigma_l(q_l^t(y)) \neq s^{t-1}(y)$, and since $\sigma_j(x) = \sigma_j(y)$ we have $w^t(y) = j = w^t(x)$, $s^t(y) = s^t(x)$, and $W^t(y) = W^t(x)$. \square

LEMMA 2.3. For every t , $|R_0^t| \leq t(t+1)/2$.

Proof of Lemma 2.3. Denote by Z^+ the set of nonnegative integers, and i, j, k, l denote only positive integers. Also, for a subset $S \subseteq \{1, 2, \dots, n\}$ and inputs $x, y \in \mathbf{I}$,

- $x \equiv y \pmod{S}$ means $x_i = y_i$ for all $i \in S$.
- $\mathbf{I}_y(S) = \{x \in \mathbf{I} \mid x \equiv y \pmod{S}\}$.

Claim. Given an integer t , a set $S \subseteq \{1, 2, \dots, n\}$, a function $h: \mathbf{I}_e(S) \rightarrow Z^+$, and sets $S_j \subseteq \{1, 2, \dots, n\}$ for every positive $j \in h(\mathbf{I}_e(S))$ that satisfy

- (1) $S_j \cap S = \emptyset$ for all j .
- (2) $|S_j| \leq t$ for all j .
- (3) $h(e) = 0$.

(4) $h(x) = j$ and $y \equiv x \pmod{S_j}$ implies $h(y) = j$.

(5) $h(x) = j$, $h(y) = k$ and $j \neq k$ implies that there exists an $i \in S_j \cap S_k$ s.t. $x_i \neq y_i$. Then there exists a set $R \subseteq \{1, 2, \dots, n\}$ s.t. $|R| \leq t(t+1)/2$ and $h(\mathbf{I}_e(S \cup R)) = 0$.

Connection between the claim and the lemma. Recall that we wanted to prove the existence of $t(t+1)/2$ input positions s.t. fixing them with values of e will ensure that

no one writes at time t in Case 3. Let R_j^t be defined as in the proof of Lemma 2.2. Then let S be the set of fixed input positions through time t , ($S = D^{t-1}$, $\mathbf{I}_e(S) = E^{t-1}$), $S_j = R_j^t - S$ for all j , and let the function $h: \mathbf{I}_e(S) \rightarrow Z^+$ be defined by $h(x) = j$ if p_j is the (unique) processor that writes on x at time t , and $h(x) = 0$ if no one writes on x at time t . Let us verify that properties (1)–(5) hold.

(1) By the definition of S_j .

(2) $|S_j| = |R_j^t - S| \leq |R_j^t| \leq t$.

(3) We deal here only with case 3, in which no processor writes on e .

(4) Since $x, y \in \mathbf{I}_e(S)$ and $x \equiv y \pmod{S_j}$, $x \equiv y \pmod{R_j^t}$. With an almost identical proof to that of Lemma 2.2 we can prove that $q_j^t(x) = q_j^t(y)$, i.e. p_j will arrive at the same state at time t for both inputs x and y . In particular, p_j will write on x if and only if it will write on y at time t .

(5) Suppose not. Then define an input z by $z_i = x_i$ if $i \in S_j$, $z_i = y_i$ if $i \in S_k - S_j$, and $z_i = e_i$ for the remaining values of i . Clearly $z \in \mathbf{I}_e(S) = E^{t-1}$. Therefore both p_j and p_k write at time t on z , contradicting the definition of the CREW PRAM.

Now we can take $R_0^t = R$, which completes the proof of the lemma. \square

Proof of Claim. The proof is by induction on t .

$t = 0$. In this case $h(\mathbf{I}_e(S)) \equiv \{0\}$. Otherwise, for some $x \in \mathbf{I}_e(S)$ there exists $j > 0$ s.t. $h(x) = j$, then (4) also $h(e) = j$, contradiction to (3).

$t > 0$. If $h(\mathbf{I}_e(S)) \equiv \{0\}$ we are done. Assume that for some $x \in \mathbf{I}_e(S)$, $h(x) = l > 0$. Set $S' = S \cup S_l$, and h' be the restriction of h to $\mathbf{I}_e(S')$, and $S'_j = S_j - S_l$ for all $j \in h'(\mathbf{I}_e(S'))$. Then we have the following:

(1') $S' \cap S'_j = \emptyset$ for all j . Clear.

(2') $|S'_j| \leq t - 1$ for all $j \in h'(\mathbf{I}_e(S'))$. Since $l, j \in h(\mathbf{I}_e(S))$, by (5) $S_j \cap S_l \neq \emptyset$, and therefore $|S'_j| = |S_j - S_l| \leq |S_j| - 1 \leq t - 1$.

(3') $e \in \mathbf{I}_e(S')$ and $h'(e) = 0$. Clear.

(4') $h'(x) = j$, $y \equiv x \pmod{S'_j}$ implies $h'(y) = j$. Since $x, y \in \mathbf{I}_e(S')$, $y \equiv x \pmod{S_l}$ and therefore $y \equiv x \pmod{S_j}$. Hence $j = h'(x) = h(x) = h(y) = h'(y)$.

(5') $h'(x) = j$, $h'(y) = k$, $j \neq k$ implies that there is an $i \in S'_j \cap S'_k$ s.t. $x_i \neq y_i$. Since $h(x) = j$, $h(y) = k$ there must be such an i in $S_j \cap S_k$. However, since $x \equiv y \equiv e \pmod{S_l}$ i must belong to $S'_j \cap S'_k$.

By the induction hypothesis, there exists a set R' s.t. $|R'| \leq (t-1)t/2$ and $h'(\mathbf{I}_e(S' \cup R')) = 0$. Set $R = R' \cup S_l$. Then clearly $|R| \leq t(t+1)/2$ and $h(\mathbf{I}_e(S \cup R)) = h(\mathbf{I}_e(S' \cup R')) = h'(\mathbf{I}_e(S' \cup R')) = 0$. \square

THEOREM 2.3. *Let N be a CREW PRAM (1) that computes a function g in time T such that $\exists e \in \mathbf{I}$ ($\mathbf{I} = \{0, 1\}^n$) for which $|\{x \in \mathbf{I} | g(x) = g(e)\}| \leq |\mathbf{I}|/r$. Then $T = \Omega(\sqrt{\log_2 r})$.*

Set $E^0 = \mathbf{I}$ and $F^0 = \mathbf{I} - E^0 = \emptyset$. Consider time period t . For any j let P_j^t be the set of input positions read by processor p_j up to time t and define E^t and its complement F^t as follows:

Case 1. There is no processor p_j and no input $x \in E^{t-1}$ such that p_j writes on x at time t : Then $E^t \leftarrow E^{t-1}$, $F^t \leftarrow F^{t-1}$.

Case 2. No processor writes on e at time t but there is an $x \in E^{t-1}$ such that some p_j writes on x at time t : For every input $x \in E^{t-1}$ which causes a processor p_j to write at time t define

$$C_x^t = \mathbf{I}(\{(i, x_i) | i \in P_j^t\}).$$

Each C_x^t is specified by the values in at most t input cells since $|P_j^t| \leq t$. It is clear that any $x \in E^{t-1}$ which causes a write at time t is in some C_x^t . Also any $y \in E^{t-1} \cap C_x^t$ will cause a write at time t since processor p_j at time t will not be able to distinguish y from x . Thus if we eliminate the elements of these ‘‘cubes’’ from E^{t-1} no writes will occur at time t .

The ‘‘cubes’’ also satisfy an additional property. If $C'_x \cap C'_y \neq \emptyset$ then their shared specifying positions must agree in value. Therefore, if $C'_x \neq C'_y$ then C'_x and C'_y must be specified by different input cells and so correspond to different processors. It follows then that $C'_x \cap C'_y \subseteq \mathbf{I} - E^{t-1} = F^{t-1}$ otherwise there would be a simultaneous write which is not allowed.

Thus if we designate the distinct cubes as $\{C'_i\}$ then

$$E^t \leftarrow E^{t-1} - \left(\bigcup_i C'_i \right) \quad \text{and}$$

$$F^t \leftarrow F^{t-1} \cup \left(\bigcup_i C'_i \right) \quad \text{where } \forall i \neq j, C'_i \cap C'_j \subseteq F^{t-1}.$$

Case 3. There is a (unique) processor p_j that writes on $e \in E^{t-1}$ at time t : Then we require that the input agree with e in the positions of P^t_j . We may regard this as requiring that the input be in the cube which is the subset of the input specified by these $|P^t_j| \leq t$ values. Equally well this may be regarded as excluding from the input all values which are in the cubes specified by the other $2^{|P^t_j|} - 1$ possible settings of values in these positions. If we call these excluded cubes $\{C'_i\}$ as in Case 2, it is immediate that $\forall i \neq j, C'_i \cap C'_j = \emptyset \subseteq F^{t-1}$. Then as in Case 2 we have

$$E^t \leftarrow E^{t-1} - \left(\bigcup_i C'_i \right) \quad \text{and} \quad F^t \leftarrow F^{t-1} \cup \left(\bigcup_i C'_i \right).$$

LEMMA 2.8. For any $t \geq 0$ and any ‘‘cube’’ C^s which is specified by at most s cells of the input \exists an integer r such that $|C^s \cap F^t| = r|\mathbf{I}| / (2^{s+t(t+1)/2})$.

Proof. By induction on t

$t = 0$: $F^t = \emptyset$ so the claim is true with $r = 0$.

Assume the claim for $t - 1$: $F^t = F^{t-1} + \sum_i (C'_i - F^{t-1})$ since $\forall i \neq j, C'_i \cap C'_j \subseteq F^{t-1}$ (we use additive notation for disjoint union). Therefore

$$\begin{aligned} |C^s \cap F^t| &= \left| C^s \cap (F^{t-1} + \sum_i (C'_i - F^{t-1})) \right| \\ &= |C^s \cap F^{t-1}| + \sum_i |(C^s \cap C'_i) - F^{t-1}| \\ &= |C^s \cap F^{t-1}| + \sum_i (|C^s \cap C'_i| - |(C^s \cap C'_i) \cap F^{t-1}|). \end{aligned}$$

If we only sum over nonempty intersections then $C^s \cap C'_i$ is a subset of the input which restricts the input only by specifying input positions and which is specified by at most $s + t$ of them. Thus we may designate $C_i^{s+t} = C^s \cap C'_i$. Therefore

$$\begin{aligned} |C^s \cap F^t| &= |C^s \cap F^{t-1}| + \sum_i (|C_i^{s+t}| - |C_i^{s+t} \cap F^{t-1}|) \\ &= \frac{p|\mathbf{I}|}{2^{s+t(t-1)/2}} + \sum_i \frac{q_i|\mathbf{I}|}{2^{s+t}} - \sum_i \frac{r_i|\mathbf{I}|}{2^{s+t+t(t-1)/2}} \quad \text{where } p, q_i, r_i \text{ are integers.} \end{aligned}$$

This follows by the inductive hypothesis for the first and last terms and because of the form of the middle terms.

Since all of the denominators divide $2^{s+t(t+1)/2}$ the claim holds for t and the lemma is proved. \square

LEMMA 2.9. $e \in E^T$ and $\forall x \in E^T, g(x) = g(e)$.

Proof of Theorem 2.3. If we apply Lemma 2.8 with $s=0$ then $C^s = \mathbf{I}$ and we see that $|F^T|$ is an integral multiple of $|\mathbf{I}|/2^{T(T+1)/2}$. Since $E^T = \mathbf{I} - F^T$ it follows that $|E^T|$ is also a multiple of this number. Now $e \in E^T$ so we have $|E^T| > 0$ and thus $|E^T| \geq |\mathbf{I}|/2^{T(T+1)/2}$. By our assumption on g and by Lemma 2.9 we need $|E^T| \leq |\mathbf{I}|r$. Therefore $2^{T(T+1)/2} \geq r$ and so $T = \Omega(\sqrt{\log_2 r})$. \square

Acknowledgments. We are indebted to a careful referee who found a serious mistake in our original Lemma 2.3, which claimed a stronger result. We thank Paul Beame for letting us include his results here. We also wish to thank Al Borodin and Dick Lipton for helpful comments on an early version of this manuscript.

REFERENCES

- [AHU-74] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA 1974.
- [B-83] P. BEAME, personal communication, 1983.
- [CD-82] S. A. COOK AND C. DWORK, *Bounds on the time for parallel RAM's to compute simple functions*, Proc. 14th ACM Symposium on Theory of Computing, 1982, pp. 231–233.
- [CSV-82] A. K. CHANDRA, L. J. STOCKMEYER AND U. VISHKIN, *Complexity theory for unbounded fan-in parallelism*, Proc. 23rd Annual IEEE Symposium on Foundations of Computer Science, 1982, pp. 1–13.
- [FW-78] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, Proc. 10th ACM Symposium on Theory of Computing, 1978, pp. 114–118.
- [FSS-81] M. FURST, J. B. SAXE AND M. SIPSER, *Parity, circuits and the polynomial-time hierarchy*, Proc. 22nd Annual IEEE Symposium on Foundations of Computer Science, 1981, 260–270.
- [GGKMRS-83] A. GOTTLIEB, R. GRISHMAN, C. P. KRUSKAL, K. P. MCAULIFFE, L. RUDOLPH AND M. SNIR, *The NYU ultracomputer—Designing a MIMD Shared Memory Parallel Machine*, IEEE Trans. Comput., C-32 (1983), pp. 175–189.
- [Go-78] L. M. GOLDSCHLAGER, *A unified approach to models of synchronous parallel machines*, Proc. 10th ACM Symposium on Theory of Computing, 1978, pp. 89–94.
- [HU-79] J. HOPCROFT AND J. ULLMAN, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA, 1979, pp. 314–318.
- [K-77] D. J. KUCK, *A survey of parallel machine organization and programming*, Computing Surveys, 9 (1977), pp. 29–59.
- [LS-81] R. LIPTON AND R. SEDGWICK, *Lower bounds for VLSI*, Proc. 13th ACM Symposium on Theory of Computing, 1981, pp. 300–307.
- [PS-82] C. PAPADIMITRIOU AND M. SIPSER, *Communication complexity*, Proc. 14th ACM Symposium on Theory of Computing, 1982, 196–200.
- [SV-81] Y. SHILOACH AND U. VISHKIN, *Finding the maximum, merging and sorting in a parallel computation model*, J. Algorithms, 2 (1981), pp. 88–102.
- [ShV-82] ———, *An $O(\log n)$ parallel connectivity algorithm*, J. Algorithms, 3 (1982), pp. 57–67.
- [SV-82] L. J. STOCKMEYER AND U. VISHKIN, *Simulation of parallel random access machines by circuits*, RC 9362, IBM T. J. Watson Research Center, Yorktown Heights, NY 1982; this Journal, 5 (1984), pp.409–422.
- [V-82] U. VISHKIN, *Parallel-design space distributed—implementation space (PDDI) general purpose computer*, RC 9541, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1982 Theoret. Comput. Sci., to appear.
- [W-83] A. WIGDERSON, *Studies in computational complexity*, Ph.D. thesis, Princeton Univ., Princeton, NJ, May 1983.
- [Y-81] A. YAO, *The entropic limitations on VLSI computations*, Proc. 13th ACM Symposium on Theory of Computing, 1981, pp. 308–311.

ON THE MOVEMENT OF ROBOT ARMS IN 2-DIMENSIONAL BOUNDED REGIONS*

JOHN HOPCROFT†, DEBORAH JOSEPH‡ AND SUE WHITESIDES§

Abstract. The *mover's problem* is the following: can an object in 3-dimensional space be moved from one given position to another while avoiding obstacles? It is known that the general version of this problem involving objects with movable joints is PSPACE hard, even for a simple tree-like structure moving in a 3-dimensional region. In this paper, we investigate a 2-dimensional mover's problem in which the object is a robot arm with an arbitrary number of joints. In particular, we give a polynomial time algorithm for moving an arm confined within a circle from one given configuration to another. We also give a polynomial time algorithm for moving the arm from its initial position to a position in which the end of the arm reaches a given point within the circle. Finally, we show that 148 circles suffice to cover the boundary of the reachable region of a joint in an arm enclosed in a circle and that the boundary can be computed in polynomial time.

Key words. robotics, manipulators, mechanical arms, algorithms, polynomial time

1. Introduction. With current interests in industrial automation and robotics, the problem of designing efficient algorithms for moving 2- and 3-dimensional objects subject to certain geometric constraints is becoming increasingly important. The *mover's problem* is to determine, given an object X , an initial position P_i , a final position P_f and a constraining region R , whether X can be moved from position P_i to position P_f while keeping X within the region R . Polynomial time algorithms (Lozano-Perez and Wesley [3], Reif [5], Schwartz and Sharir [6]) are known in the case where X is a rigid 2- or 3-dimensional polyhedral object, and R is a region described by linear constraints.

A more difficult problem, which is related to problems in robotics, assumes that the object X has joints and is hence nonrigid. Schwartz and Sharir [7] give a polynomial time algorithm, the degree of the polynomial being exponential in the number of joints, for moving X from position P_i to P_f within a region R . Unfortunately, an algorithm with running time polynomial independent of the number of joints is unlikely, as Reif [5] has shown that the problem of deciding whether an arbitrary hinged object can be moved from one position to another in a 3-dimensional region is PSPACE complete.

This paper investigates variants of the mover's problem that we believe are of interest. We begin in §§ 2 and 3 by considering the problem of folding a *carpenter's ruler*—that is, a sequence of line segments hinged together consecutively. This problem arises because a natural strategy for moving an arm in a confining region is to fold it up as compactly as possible at the beginning of the motion. Unfortunately, deciding whether an arbitrary carpenter's ruler (whose link lengths are not necessarily equal) can be folded into a given length is NP-complete. Because of this, it turns out to be at least NP-hard to decide whether or not the end of an arbitrary *arm* (i.e., a carpenter's ruler with one end fixed) can be moved from one position to another while staying within a given 2-dimensional region.

In §§ 4–6 we consider the problem of moving an arm inside a circular region, and we are able to give polynomial time algorithms for changing configurations and reaching

* Received by the editors August 1, 1983, and in revised form January 19, 1984. This work was supported in part by the Office of Naval Research under contract N00014-76-C-0018, the National Science Foundation under grant MCS81-01220, and an NSF Postdoctoral Fellowship, and a Dartmouth College Junior Faculty Fellowship.

† Computer Science Department, Cornell University, Ithaca, New York 14853.

‡ Computer Science Department, University of Wisconsin, Madison, Wisconsin 53706.

§ School of Computer Science, McGill University, Montreal, Quebec, Canada H3A 2K6.

points. Also, we show that circles covering the boundary of the set of points reachable by a joint can be computed in polynomial time.

2. Folding a ruler. In this section, we ask how hard it is to fold a carpenter’s ruler consisting of a sequence of n links L_1, \dots, L_n that are hinged together at their endpoints. These links, which are line segments of integral lengths, may rotate freely about their joints and are allowed to cross over one another. We assume that the endpoints of the links are consecutively labeled A_0, \dots, A_n and for $1 \leq i \leq n$, we let l_i denote the length of link L_i . (See Fig. 2.1.) We define the Ruler Folding problem to be the following:

Given: Positive integers n, l_1, \dots, l_n and k .

Question: Can a carpenter’s ruler with lengths l_1, \dots, l_n be folded (each pair of consecutive links forming either a 0° or 180° angle at the joint between them) so that its folded length is at most k ?

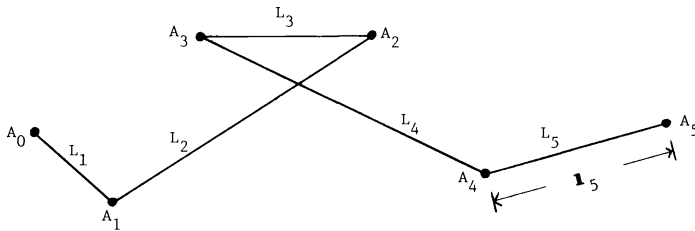


FIG. 2.1. A typical ruler with five links.

By a reduction from the NP-complete PARTITION problem (see Garey and Johnson [1]) we can easily show that the RULER FOLDING problem is also NP-complete. The PARTITION problem asks whether, given a set S of n positive integers l_1, \dots, l_n , there is a subset $S' \subseteq S$ such that

$$\sum_{l_i \in S'} l_i = \sum_{l_j \in S - S'} l_j.$$

THEOREM 2.1. *The RULER FOLDING problem is NP-complete.*

Proof. Given an instance of the PARTITION problem with $S = \{l_1, \dots, l_n\}$, let $d = \sum_{i=1}^n l_i$. Then the desired subset S' of S exists if and only if a ruler with links of length $2d, d, l_1, \dots, l_n, d, 2d$ (in consecutive order) can be folded into an interval of length at most $2d$. To see that this is the case, imagine that the ruler is being folded into the real line interval $[0, 2d]$, and notice that both the initial endpoint A_0 of link L_1 (the third link in our ruler) and the terminal endpoint A_n of link L_n (the third from last link) must be placed at integer d . The set S' in the PARTITION problem then corresponds to the set of links L_i whose initial endpoints A_{i-1} appear to the left of their terminal endpoints A_i in a successful folding of the ruler. \square

The RULER FOLDING problem and the PARTITION problem share not only the property of being NP-complete, but also the property of being solvable in pseudo-polynomial time. The time complexity of the RULER FOLDING problem is bounded by a polynomial in the number of links, n , and the maximum link length, m . In fact, it is possible to find the *minimum* folding length in time proportional to $n * m$ by a dynamic programming scheme. However, in order to carry out this scheme we need to know that a ruler with maximum link length m can always be folded to have length at most $2m$.

LEMMA 2.1. *A ruler with lengths l_1, \dots, l_n can always be folded into length at most $2m$, where $m = \max \{l_i | 1 \leq i \leq n\}$.*

Proof. Place link L_1 into the interval $[0, 2m]$ with A_0 at 0. Having placed links L_1, L_2, \dots, L_{i-1} into the interval, position L_i as follows: Place L_i with A_i to the left of A_{i-1} , if possible. Otherwise, place L_i with A_i to the right of A_{i-1} . To see that this is possible, suppose that p is the position of A_{i-1} and note that if A_i cannot be placed to the left of A_{i-1} , then $p \leq 1_i \leq m$. Hence A_i can surely be placed to the right of A_{i-1} . \square

Using this result, we can now give an $O(m^2 * n)$ dynamic programming algorithm for determining the minimum folding length of a ruler, where n is the number of links in the ruler and m is the maximum length of any given link.

ALGORITHM 2.1. Ruler folding in minimum length. Given a ruler with links L_1, \dots, L_n , compute the maximum link length m . Then, for each $k, 1 \leq k \leq 2m$, construct a table with rows numbered 0 to n and columns numbered 0 to k . For a given k , a T is placed in row i , column j if the linkage L_1, L_2, \dots, L_{i-1} fits in $[0, k]$ with the endpoint A_i at integer j . Row 0 is filled in by writing a T in each column j . Once row $i-1$ has been filled in, fill in row i by writing a T in each column j for which the linkage L_1, \dots, L_i fits in $[0, k]$ with endpoint A_i at integer j . To do this, examine row $i-1$ to obtain the possible locations for A_{i-1} . The last row of the completed table contains a T if and only if the ruler can be folded into $[0, k]$. Find the smallest k for which the table contains a T in the last row, and read the table from bottom to top to reconstruct the desired folds.

The next example shows that $2m$ is, in some sense, the best upper bound for the minimum folding length.

Example 2.1. A ruler with minimum folding length $2m - \epsilon$. Consider a ruler which has $n = 2k - 1$ links L_1, \dots, L_n . (See Fig. 2.2.) Suppose that links with odd subscripts have length m and that links with even subscripts have length $m - \epsilon$, where $\epsilon = m/k$. It is easy to check that this ruler cannot be folded into length less than $2m - \epsilon$.

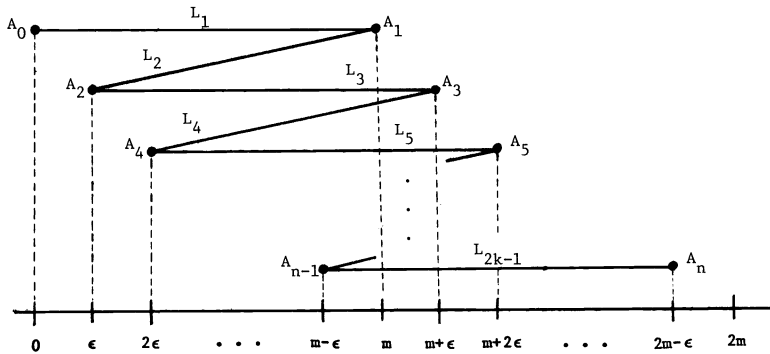


FIG. 2.2. The ruler of Example 2.1.

Having established some basic results about folding rulers, we now return to the original problem of moving such objects.

3. Moving an arm in two dimensions. The remainder of this paper is concerned with moving a ruler that has one endpoint, A_0 , pinned down. We will refer to such a ruler as an *arm*.

Unrestricted movement. It is easy to find out what points can be reached by the free end of an arm placed in the plane. The answer is given in the next lemma, whose simple proof we omit. (The lemma extends readily to three dimensions.)

LEMMA 3.1. Let L_1, \dots, L_n be an arm positioned in 2-dimensional space, and let $r = \sum_{i=1}^n l_i$, the sum of the lengths of the links. Then the set of points that A_n can reach is a disc of radius r centered at A_0 —unless some l_i is greater than the sum of the other lengths. In that case, the set of points A_n can reach is an annulus with center A_0 , outer radius r , and inner radius $l_i - \sum_{j \neq i} l_j$.

Restricted movement. If an arm is constrained to avoid certain specified objects during its motions, then determining whether A_n can reach some given point p is difficult. The following example suggests that a reduction of RULER FOLDING can be used to show that this problem is NP-hard even for walls consisting of a few straight line segments.

Example 3.1. A hard decision problem. We want to know whether the arm shown in Fig. 3.1 can be moved so that A_n reaches the given point p . The arm consists of a “ruler” with links of integral lengths attached to a “chain” of very short links. The chain links are short enough to turn freely inside the tunnel, which is sufficiently narrow that links of the ruler can rotate very little once they are inside. Since the ruler cannot change its shape very much while moving through the tunnel, it must be foldable into length at most k in order to move through the gap of width k . Thus, point p can be reached if and only if the ruler can be folded into length at most k .

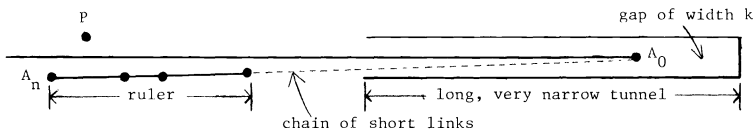


FIG. 3.1. A point that is hard to reach.

We would like to find natural classes of regions for which questions concerning the movement of arms are decidable in polynomial time. Certainly the simplest such region is the inside of a circle, since there are no corners in which an “elbow” might be caught. We believe that studying motions inside a circle sheds light on the underlying movements of the arm without the complexities that arise in situations where a link can jam in a corner. For the remainder of this paper, we will discuss polynomial algorithms for moving an arm within a circle. In a subsequent paper, we hope to treat more general situations.

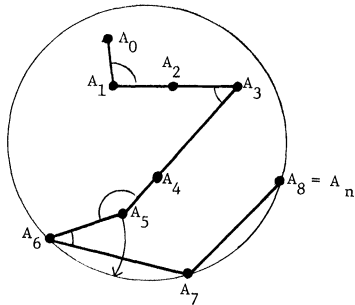
4. Changing configurations inside a circle. In this section, we solve the problem of moving an arm from one given configuration to another inside a circular region. Simply determining whether this can be done turns out to be a matter of checking that links whose “orientations” differ in the two configurations can be reoriented. This checking can be done in time proportional to the number of links. Assuming that it is feasible to change configurations, we show how to move the arm to its desired final position by first moving it to a certain “normal form” and then putting each link into place, correcting its orientation if necessary. Correcting orientation involves destroying and then restoring the positions of previous links. Our algorithm consists of a sequence of “simple motions” (which we are about to define), and the length of this sequence is on the order of the cube of the number of links.

Simple motions. A definition of a “simple motion” is needed in order to make clear the sense in which our algorithms for moving an arm are polynomial. This definition should not limit the positions the arm can reach nor should it complicate the algorithms and proofs. With these considerations in mind, we define a “simple

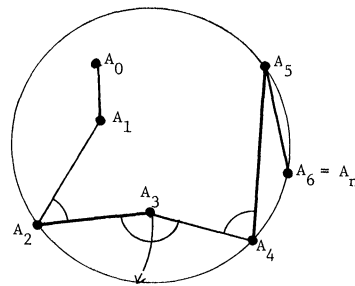
motion" of an arm as follows. (There are many other definitions which would give similar results.)

DEFINITION 4.1. A *simple motion* of an arm is a continuous motion during which at most four joint angles change. (The angle between the first link and some reference line through the fixed point A_0 may be one of these.) Moreover, a changing angle is not allowed both to increase and to decrease during one simple motion.

Figure 4.1 illustrates some simple motions of the type we use. Note that in the motions shown, the joints where angles are changing are connected together by straight sections of the arm. This is true of all the simple motions we will use.



A_5 is moving to the circle by a simple motion. The *locations* of $A_0, A_1, A_6, A_7,$ and A_8 remain fixed. The *angles* at $A_1, A_3, A_5,$ and A_6 are changing.



A_3 is moving to the circle by a simple motion. The *locations* of $A_0, A_1,$ and A_2 remain fixed. $A_4, A_5,$ and A_6 move first counter-clockwise, then clockwise around the circle. Only the angles at $A_2, A_3,$ and A_4 are changing.

FIG. 4.1. Examples of simple motions.

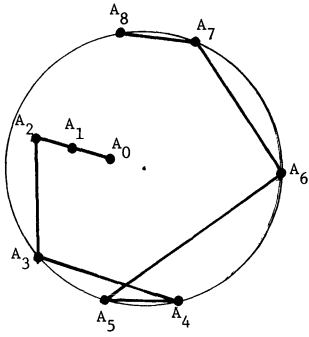
Normal form. It is convenient to begin by showing that any arm positioned within a circle can be moved by a short sequence of simple motions into a normal form that has as many joints as possible positioned on the circle. We immediately dispense with the case in which the distance from A_0 to the circle is greater than the length of the entire arm, since in this case the circle is irrelevant.

DEFINITION 4.2. Suppose A_0 is fixed at some point distance d_0 from the circle, and suppose that j is the smallest integer such that $\sum_{i=1}^j l_i \geq d_0$. Then the arm is in *normal form* if and only if L_1, \dots, L_j contains at most one bent joint, and for each $k, j \leq k \leq n, A_k$ is on the circle. Moreover, if L_1, \dots, L_j is not a straight line of links, the bend is at joint A_{j-1} . (See Fig. 4.2.) In any event, L_1, \dots, L_{j-1} lie on a radius.

LEMMA 4.1 (normal form). For any given configuration of an arm within a circle there is a sequence of $O(n)$ simple motions that moves the arm to normal form. Moreover, this sequence can be computed in $O(n)$ time.

Proof. The process consists of two stages. First, the tail will be straightened until A_n reaches the circle. Then, starting with A_{n-1} , the other joints will be moved one by one onto the circle.

Suppose L_j, L_{j+1}, \dots, L_n form a straight line segment. Move A_n toward the circle by rotating this segment about A_{j-1} until A_n reaches the circle or L_{j-1} is added to the



$A_0, A_1,$ and A_2 lie on a radius. A_3 is the first joint that can reach the circle. A_3 and its successors lie on the circle.

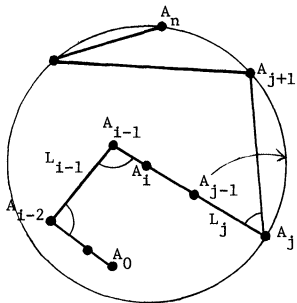
FIG. 4.2. An arm in normal form.

straight segment. In this latter case, rotate the extended straight segment about A_{j-2} . Eventually, A_n reaches the circle or the entire arm becomes a straight segment that can be rotated about A_0 to place A_n on the circle. (Recall that we are assuming that the arm is long enough to reach the circle.) This process requires at most $O(n)$ simple motions and can be computed in $O(n)$ time.

Now assume that A_n, A_{n-1}, \dots, A_j are on the circle, that A_{j-1} is not on the circle, and that the sum of the lengths of the first $j-1$ links exceed the distance from A_0 to the circle. Let $L_i, L_{i+1}, \dots, L_{j-1}$ be the maximal straight segment leading back from A_{j-1} .

If $A_{i-1} = A_0$, then leave the line of links L_1, \dots, L_{j-1} straight while rotating it about A_0 so that A_{j-1} moves closer to the circle. (If L_1, \dots, L_{j-1} lies on a diameter, rotate clockwise, say.) At the same time, adjust the angle at A_j and move A_j, \dots, A_n around the circle. Stop when A_{j-1} reaches the circle.

If A_{i-1} is not A_0 , then A_{i-1} is a bent joint. Keeping $L_i, L_{i+1}, \dots, L_{j-1}$ straight and the positions of A_j and A_{i-2} fixed, rotate L_j about A_j moving A_{j-1} away from A_{i-2} . Rotate clockwise, say, if there is a choice because A_{i-1} is completely folded. (See Fig. 4.3.) L_j is rotated until A_{j-1} hits the circle (in which case we have a new joint on the circle), or L_{i-1} is added to the straight segment L_i, \dots, L_{j-1} , or A_{i-1} hits the circle. If L_{i-1} is added to the straight segment, then the process of rotating L_j is continued with the straight segment replaced by a new one containing at least L_i, \dots, L_{j-1} and L_{i-1} . If A_{i-1} hits the circle, then A_{i-1} is held fixed while the angles at joints A_{i-1}, A_{j-1} and A_j are adjusted so as to push A_{j-1} to the circle while keeping A_j and its successors on the circle. In this way, one can force onto the circle as many joints as possible (i.e., A_j can be placed on the circle, where j is minimum such that the sum of the lengths of the first j links exceeds the distance from A_0 to the circle). Once these joints are on the circle, it is easy to position the links at the beginning of the arm as desired.



A_{j-1} moves toward the circle away from A_{i-2} . The locations of A_{i-2} and its predecessors and the locations of A_j and its successors remain fixed. Only the angles at $A_{i-2}, A_{i-1}, A_{j-1},$ and A_j are changing.

FIG. 4.3. Moving an arm to normal form.

This process requires $O(n)$ simple motions and once again, these motions can be computed in $O(n)$ time. Thus, a total of $O(n)$ simple motions is needed to put an arm into normal form, and $O(n)$ time is needed to compute the motions. \square

Reorientation of links. For any given position of an arm inside a circle, we define each link to have either “left” or “right” orientation. This is done by first observing that the straight line extension of a link L_i cuts the circle into two arcs. L_i is said to have *left orientation* if the arc on the left of the extension, viewed from A_{i-1} to A_i , is no longer than the arc on the right. *Right orientation* is defined in a similar manner. (See Fig. 4.4.) Note that a link that is on a diameter of the circle can be regarded as having either orientation and that a link must move to a diameter in order to change orientation.

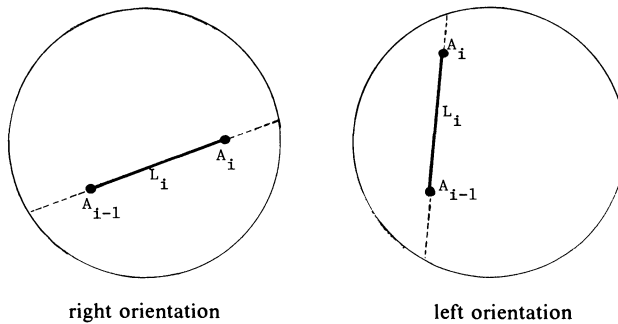


FIG. 4.4. Link orientations.

An obvious necessary condition for being able to move the arm from one configuration to another is that it be possible to reorient each link whose orientation differs in the two configurations. (It turns out that this condition is also sufficient.) We are about to show that determining whether a link can be reoriented is simply a matter of determining how far its endpoints can be moved from the circle.

For an arm with A_0 fixed within a circle C , let c_i and d_i denote the minimum and maximum distance that A_i can be moved from C by arbitrary motions of the arm within C . Of course, distance is measured along a radius of C , so $0 \leq c_i \leq d_i \leq d/2$, where d is the diameter of C .

Since A_0 is fixed, c_0 and d_0 are determined by the position of A_0 . The Normal Form Lemma 4.1 shows that each successive A_i can get closer to the circle by the amount l_i until the circle is reached. Thus

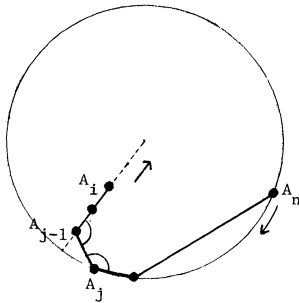
$$c_i = \max \{c_{i-1} - l_i, 0\}.$$

Computing the d_i 's is slightly more complicated. We begin by computing for each $i, 0 \leq i \leq n$, the maximum distance t_i that A_i could move from the circle if it were constrained only by the tail of the arm (i.e., if L_{i+1}, \dots, L_n were freed from L_1, \dots, L_i and L_1, \dots, L_i were discarded). This leaves a “tail” L_{i+1}, \dots, L_n that has no joints pinned down. Note, however, that the presence of a long link could prevent A_i from moving far off the circle. For example, if link L_{i+1} has length d , then $t_i = 0$! Then we compute d_i from t_i, c_{i-1} , and d_{i-1} .

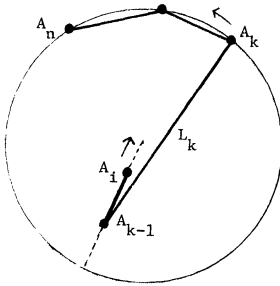
LEMMA 4.2. For any arm $L_1, \dots, L_i, \dots, L_n$ inside a circle of diameter d ,

$$t_i = \begin{cases} d/2 & \text{if no link beyond } A_i \text{ is longer than } d/2, \\ \min \{d/2, d - l_k + \sum_{i < j < k} l_j\} & \text{where } l_k \text{ is the length of the} \\ & \text{first link beyond } A_i \text{ longer than } d/2 \text{ otherwise.} \end{cases}$$

Proof. For any configuration of the tail, L_{i+1}, \dots, L_n , keep A_i fixed while moving the tail to a normal form. Let A_j be the first joint on the circle. If $j \geq i + 2$, the straight section of arm between A_i and A_{j-1} lies on a radius of the circle. (If $j = i$ or $i + 1$, this section is just the point A_i .) While changing only the angles at joints A_{j-1} and A_j , one can push this straight section along the radius toward the circle's center while A_j and its successors move around the circle. (See Fig. 4.5.) New links are added to the moving straight section until A_i reaches the center or the first long link L_k prevents further travel because it has folded against the straight section (or reached the diameter in the case $L_k = L_{i+1}$). \square



A_0, \dots, A_{i-1} have been removed. A_i, \dots, A_{j-1} move along the radius while A_j, \dots, A_n move around the circle. Only the angles at A_{j-1} and A_j are changing.



Joint A_{k-1} is about to fold completely, preventing further travel of A_i along the radius.

FIG. 4.5. Moving A_i distance t_i from the circle.

Now that we have calculated the t_i 's, it is easy to calculate the d_i 's. We only need to observe that for any given distance x between c_i and d_i , there is obviously some way to move A_i to a position that is distance x from the circle. (The distance of A_i from the circle is a continuous function that must take on all values from c_i to d_i as A_i moves from one extreme to the other.) For $i > 0$:

$$d_i = \begin{cases} \min \{t_i, d_{i-1} + l_i\} & \text{if } l_i < d/2 - d_{i-1}, \\ t_i & \text{if } d/2 - d_{i-1} \leq l_i \leq d/2 - c_{i-1}, \\ \min \{t_i, d - l_i - c_{i-1}\} & \text{if } l_i > d/2 - c_{i-1}. \end{cases}$$

The point of the next remarks and lemma, which we need before we can give an algorithm for reorienting the links of an arm, is that this can be done using a short sequence of simple motions.

Remark 4.1. Suppose that the tail L_{j+1}, \dots, L_n has been detached from the arm L_1, \dots, L_n . Then note that this tail can be moved from its initial position so that the distance between A_j and the circle monotonically increases or decreases. To see this, put the tail (regarded as an arm with initial point A_j fixed) into normal form. Then move the straight segment of links containing A_j along the radius on which it lies,

adding or deleting links from the segment as their endpoints get closer to or farther from the center of the circle. \square

Remark 4.2. Consider the arm as a whole, and suppose the tail beginning at A_j is in normal form. Then L_j can be rotated about A_{j-1} to push A_j closer to or farther from the circle while the angles at A_j and two other joints in the tail are adjusted to keep the tail constantly in normal form. In fact, Remark 4.1 shows that any rotation of L_j for which the distance between A_j and the circle is either an increasing or a decreasing function can be carried out in at most $n - j$ simple motions. \square

LEMMA 4.3. *Let A_j be a joint of an n -link arm positioned within a circle. For any x between c_j and d_j , there is a sequence of $O(n^2)$ simple motions that moves the arm from its original position to a position in which A_j is distance x from the circle.*

Proof. Compute the c_i for each predecessor A_i of A_j . Then, given x , compute the sequence of numbers defined by the following recursive formula:

$$x_i = x \quad \text{for } i = j,$$

$$x_{i-1} = \max \{c_{i-1}, x_i - l_i\} \quad \text{for } 2 \leq i \leq j.$$

(Note that $c_i \leq x_i \leq d_i$. This can be seen by working backwards from $i = j$ and observing that $x_i \leq d_{i-1} + l_i$ at each step. Thus $x_i - l_i \leq d_{i-1}$.) To position A_j distance x_j from the circle, first put the entire arm into normal form ($O(n)$ steps). Then, beginning with A_1 , move each A_i in turn to a position distance x_i from the circle. This is done by rotating L_i about A_{i-1} while keeping the tail in normal form. All together, at most $(n - 1) + (n - 2) + \dots + (n - j)$ additional simple motions are needed, so the entire repositioning sequence contains $O(n^2)$ motions. Note that this sequence can be computed in $O(n^2)$ time. \square

We are now ready to give the conditions under which links can be reoriented.

LEMMA 4.4. *A link L_i can be reoriented if and only if at least one of the following inequalities holds:*

- i) $d - l_i \leq d_{i-1} + d_i$;
- ii) $d_i \geq l_i + c_{i-1}$;
- iii) $d_{i-1} \geq l_i$.

Furthermore, if L_i can be reoriented, then this can be done with $O(n^2)$ simple motions that can be quickly computed.

Proof. As we noted at the beginning of this subsection, L_i must lie on a diameter in order to be reoriented. Hence, the above conditions are obviously necessary because i) holds when L_i is on a diameter and the center of the circle is between A_{i-1} and A_i , ii) holds when L_i lies on a radius with A_i closer to the center than A_{i-1} , and iii) holds when L_i lies on a radius with A_{i-1} closer to the center than A_i .

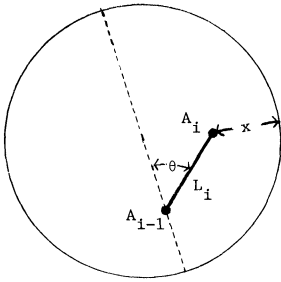
To prove that the conditions are also sufficient, first suppose that inequality i) or iii) holds. Using the method in the proof of Lemma 4.3, move A_{i-1} to a position distance d_{i-1} from the circle in $O(n^2)$ simple motions. After this has been done, hold A_{i-1} fixed, and rotate L_i about A_{i-1} to bring L_i to the diameter through A_{i-1} . By Remark 4.2 this takes at most $n - i$ simple motions, and these can be quickly computed.

If inequality ii) holds, move A_{i-1} distance c_{i-1} from the circle, and then rotate L_i to the diameter. \square

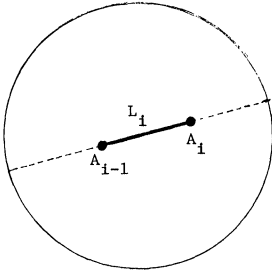
We need to make one more observation before we can show how to change configurations.

Remark 4.3. Suppose L_i is a link that can be reoriented. Then starting from any initial configuration of the arm, we can reorient L_i and return A_1, \dots, A_{i-1} to their starting positions without changing the new orientation of L_i , all with $O(n^2)$ motions.

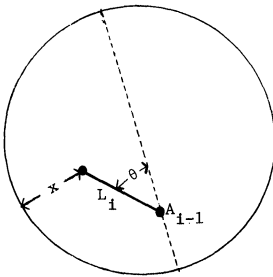
To see this, bring L_i to a diameter with $O(n^2)$ simple motions, and then “undo” these motions but with the orientation of L_i reversed. That is, keep the angle at A_{i-1} adjusted so that at corresponding moments before and after L_i reaches the diameter through A_i , L_i forms the same angle with this diameter but lies on the opposite side of it. This keeps A_i the same distance from the circle at corresponding times. (See Fig. 4.6.) To check that the tail can be moved in a compatible fashion, note that reversing the changes in the size of the angles in the tail indeed keeps A_i the same distance from the circle at corresponding times. Although the tail does not return to its original *configuration*, it does return to its original *shape*. \square



At time $t_0 - t$, L_i forms an angle θ with the diameter through A_{i-1} , and A_i is distance x from the circle.



At time t_0 , L_i reaches a diameter.



At time $t_0 + t$, A_{i-1} has returned to the position it occupied at time $t_0 - t$. L_i again forms angle θ with the diameter through A_{i-1} , but has changed orientation. The distance between A_i and the circle is again x .

FIG. 4.6. Reorientation of a link L_i with restoration of A_1, \dots, A_{i-1} .

An algorithm for changing configurations. Suppose we are given an initial configuration and a desired final configuration of an arm within a circle. Using the formulas of the preceding subsection, we can quickly compute the c_i 's, d_i 's, and t_i 's. Using Lemma 4.4, we can then quickly check whether each link with differing initial and final configuration can be brought to the diameter. If this necessary and sufficient condition holds, then the following motion algorithm shows that the arm can be moved to the desired final configuration with $O(n^3)$ simple motions.

ALGORITHM 4.1. *Algorithm for changing configuration.*

Step i) Move the arm to normal form ($O(n)$ simple motions);

Step ii) Once the predecessors of A_i are in their final positions, reorient L_i if necessary, restoring the predecessors of A_i to their final positions ($O(n^2)$ motions, by Remark 4.3). Then rotate L_i about A_{i-1} to put A_i in final position ($n - i$ simple motions, by Remark 4.2). Increment i , and repeat Step ii) until $i > n$.

Notice also that the decision problem of whether the desired final configuration can be attained can be answered in linear time on a machine that does real arithmetic (+, -, *, /2, min(,)) since it is necessary only to compute the c_i 's, d_i 's, and t_i 's, determine the links which must be reoriented, and check that the conditions of Lemma 4.4 hold for these links.

In the next section, we show how to reduce the *problem* of reaching a given point with A_n to a problem of changing configurations.

5. Reaching a point with an arm inside a circle. In this section, we will solve the problem of deciding whether an arm inside a circle can be moved from a given initial position to one which places A_n at some given point p . We will do this by showing that this problem can be reduced to the problem of changing configurations, which we solved in the last section.

Points on the circle reached by the A's. We want to compute a *feasible configuration* (i.e., one to which the arm can be moved from its initial configuration) that places A_n at a given point p (inside or on the circle). In order to find such a configuration, we first construct the set R_j of points *on* the circle that can be reached by A_j from the given initial position of the arm.

LEMMA 5.1. *Each R_j consists of at most two arcs of the circle.*

Proof. (Induction on j). Clearly, $R_0 = \{A_0\}$ if A_0 is on the circle. Otherwise, the Normal Form Lemma 4.1 shows that the first nonempty R_j is the one for which

$$l_1 + \cdots + l_{j-1} < c_0 = d_{j0} \leq l_1 + \cdots + l_j,$$

and that all subsequent R_j 's are nonempty. It is easy to see that the *first* nonempty R_j consists of at most two arcs.

Now consider a joint for which R_{j-1} is nonempty but consists of at most two arcs. If A_j is at some point in R_j , we can move A_{j-1} to the circle while moving A_j around the circle. (This can be done in the same way that an arm is put into normal form.) Of course, A_j stays in one arc of R_j during this process. Thus, each point in R_j belongs to an arc of R_j that contains a point reached by A_j with A_{j-1} in R_{j-1} . This number is at most the number of arcs generated by placing A_{j-1} in R_{j-1} and A_j on the circle.

Depending on the possible orientations for L_j , there are one or two possible locations for A_j for each location of A_{j-1} . These locations for A_j , when taken together, give rise to at most two arcs for each arc of R_{j-1} . (It need not be the case that A_{j-1} can move throughout its arc of R_{j-1} without leaving the circle.) We have already observed that the other points in R_j can be moved along the circle to points in these arcs. What we show next is that if L_j has two possible orientations, then the two corresponding arcs that an arc of R_{j-1} produces must overlap, so that each arc of R_{j-1} gives rise to just one arc of R_j .

Suppose that A_{j-1} and A_j are on the circle and that $d_{j-1} \geq l_j$. Then we can reorient L_j while moving A_j around the circle, keeping A_j in R_j . Our observation about counting arcs shows that each arc of R_{j-1} gives rise to only one arc in R_j . Thus in this case, R_j consists of at most two arcs.

Now suppose that A_{j-1} and A_j are on the circle and that $d_{j-1} < l_j$. Then we can move A_{j-1} from any point in R_{j-1} to any other point in R_{j-1} , which may require lifting A_{j-1} off the circle, without ever taking A_j off the circle or changing the orientation of L_j . Hence, all the points of R_j that are reached from R_{j-1} by L_j with left orientation are in the same arc of R_j . The same is true for L_j with right orientation, so again R_j consists of at most two arcs. \square

In our algorithm for reaching a point p , we will need to find for any given point in R_j a feasible configuration of the arm that positions A_j at that point. In the next section, we show how to compute this information quickly.

Determining the R 's. First we will show that each set R_j is a union of certain contributions from its predecessors, and then we will describe an algorithm for calculating the R_j 's and determining how to reach them.

The following lemma, the proof for which we omit, can easily be established using the ideas in the proof of the Normal Form Lemma 4.1.

LEMMA 5.2. *Suppose an arm is positioned inside a circle so that A_j is located at a point p_j on the circle. Then A_j can be kept fixed at p_j while the arm is moved to a position where one of the following conditions holds:*

- i) links L_1, \dots, L_j form either a straight line (with no folds) or an "elbow" whose only bend is at A_{j-1} ;
- ii) for some $i < j$, A_i is on the circle, and links L_{i+1}, \dots, L_j form either a straight line or an elbow whose only bend is at A_{j-1} .

Given a value for j , we need to find out for each R_j , $i < j$, which points of R_j can be reached from R_i by the straight lines and elbows of Lemma 5.2.

Suppose that p_i is a point in R_i and that $l_{i+1} + \dots + l_j \leq d$. If all the links between A_i and A_j can be given the same orientation, then p_i contributes a point to R_j by means of a straight line. (If both orientations are possible, then p_i contributes two points to R_j .) Contributions of this type from points in R_i form at most four arcs, two for each arc of R_i . These arcs amount to shifts of R_i around the circle.

Now consider the possibilities for joining a point p_i in R_i to a point p_j in R_j by an elbow whose last joint is the one which is bent. Certainly $l_{i+1} + \dots + l_{j-1}$ must be at most d . Since L_j and the straight line from A_i to A_{j-1} might have either orientation, there are four types of elbows to consider. Consider a particular feasible elbow, and note that it must place A_{j-1} somewhere on an arc of a circle of radius $l_{i+1} + \dots + l_{j-1}$ centered at p_i . Since the orientations of the links in the elbow are specified, this arc is bounded by the circle at one end and by the diameter through A_i at the other. The set of points that can then be reached by L_j in its specified orientation, with A_{j-1} on the arc, forms an arc on the circle. Hence, each feasible elbow type allows R_i to contribute a widened shift of itself to R_j .

The contributions of A_0 to R_j can be determined in a similar fashion.

It is now easy to give an $O(n^2)$ algorithm to do the following: compute the endpoints of the R_j 's, and build a table that allows one, given a p_j in R_j , to find in $O(n)$ time (where n is the number of links in the arm) a feasible configuration having A_j at p_j .

ALGORITHM 5.1. *Finding the R 's.* First, determine how the links can be oriented ($O(n)$ time). Next, compute the contributions from A_0 of straight lines and elbows whose last joint is the one that is bent. Record these contributions by listing the endpoints of the arcs together with the description of the lines or elbows that generated them ($O(n)$ time). At this stage, the first nonempty R_i has been completely determined, and so its endpoints (of which there are at most four) can be computed ($O(n)$ time).

Finally, for each R_i in turn, compute the contribution of R_i to its successors, and then compute the endpoints of R_{i+1} ($O(n)$ time per iteration).

In the next subsection, we use the information about the R_j 's to solve the problem of moving A_n to an arbitrary point inside the circle.

How to reach a point. If we want to place A_n at a point p on the circle, we merely compute R_n and test p for membership. If p is in R_n , we use the table generated by Algorithm 5.1 to determine a feasible arm configuration that has A_n at p . Then we can use Algorithm 4.1 to move the arm to this configuration.

Now suppose p is inside the circle. If the arm can be moved to a configuration in which A_n is at p and some other joint is on the circle, then p can be reached by a feasible configuration in which some A_i is on the circle and links L_{i+1}, \dots, L_n form either a straight line or an elbow with the bend at A_{i+1} . To see whether this happens, we compute the R_j 's and then look for an appropriate straight line or elbow reaching from p back to a nonempty R_j . If no such line or elbow can be found, we check to see whether p can be reached by a configuration that does not touch the circle.

LEMMA 5.3. *Suppose that an arm L_1, \dots, L_n can be moved to a configuration in which A_n is at a given point p inside the circle, but that no such feasible configuration can have any joint on the circle. Then the arm can be moved to a configuration in which A_n is at p and at most two joints are bent.*

Proof. Consider a feasible configuration with A_n at p . If it has more than two bends, proceed as follows. Let A_i, A_j , and A_k , where $0 < i < j < k < n$, denote the first three bent joints. Let A_m denote the fourth bent joint if one exists; otherwise, set $A_m = A_n$. Keeping A_k and its successors pinned down, rotate the line of links between A_0 and A_i about A_0 so that A_i moves away from A_m . (See Fig. 5.1.) Eventually, one of three events must occur:

- i) some joint straightens (in which case we can start over with a smaller number of bends);
- ii) A_i moves close enough to A_k to fold the joint A_j completely;
- iii) A_i reaches the line through A_0 and A_m .

Note that by hypothesis, no joint can hit the circle.

If ii) occurs, keep joint A_j folded, unpin A_k , and continue the rotation. Since A_i is moving away from A_m , the rotation can continue until joint A_k straightens or A_i reaches the line through A_0 and A_m .

Assume that A_i, A_0 , and A_m are collinear. Pin down A_0, \dots, A_i and A_m, \dots, A_n , and rotate the line of links between A_i and A_j about A_i so that A_j moves away from A_m . One of the joints A_i and A_k must straighten during this rotation. \square

There are $O(n^2)$ configurations of the type described in Lemma 5.3, and each one can be tested for feasibility in constant time. All together, then, we need $O(n^2)$ time to compute the R_j 's, $O(n)$ additional time to check for a feasible configuration with some joint on the circle, and if no such configuration exists, $O(n^2)$ time to check for feasible configurations with no joint on the circle. If a feasible configuration is found, we can then use Algorithm 4.1 to move A_n to p with $O(n^3)$ simple motions. Note that our method can be used to solve the problem of moving any arbitrary joint A_j to a specified point.

6. Covering the boundary of a reachable region. In this section, we consider the boundary of the set of points that can be reached by a joint of an arm enclosed in a circle. It turns out that this boundary can always be covered by a finite number of arcs of circles. In fact, the number of circles involved is never more than 148, a bound which we did not attempt to lower. For each joint, the entire process of computing

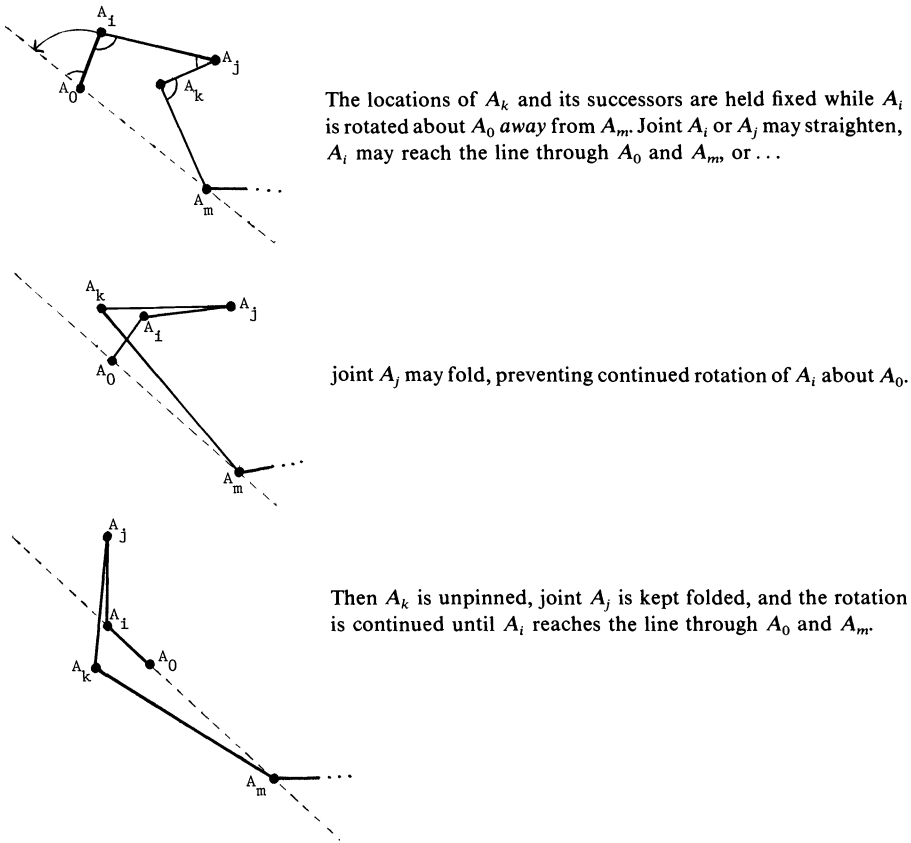


FIG. 5.1. Reaching p with at most two bent joints.

the centers and radii of the circles can be done in time proportional to a polynomial in the number of links in the arm.

Our proof that the boundary of a reachable region can be covered by at most 148 circles is technical and involves much case by case analysis. We will outline the proof here, referring the reader to Hopcroft, Joseph and Whitesides [2], where all the technical details appear.

We now summarize the main points of the outline before giving it in more detail.

There are certain circles, such as C itself, that are obvious candidates for inclusion in the set of covering circles and that will be called "basic". If a joint A_m is on the boundary of its region but does not lie on one of these "basic" circles (which we define later), then at least one of the predecessors of A_m must lie on C . In fact, the joints between A_0 and A_m that lie on C can be thought of as breaking the arm between A_0 and A_m into "segments". The intermediate segments consist of straight lines of links, but the initial and final segments may each have one joint that is completely folded. No joint is partially folded. Consideration of the special case in which the final segment lies on a diameter of C gives rise to some additional "supplementary" circles that are added to the set of covering circles.

If A_m lies on the boundary of its region S_m but does not lie on a basic or supplementary circle, then the portion of the arm between A_0 and A_m must lie in one of several possible configurations. In each of these configurations, the number of possible locations for the last joint A_j before A_m that lies on C is small. These possible

locations become centers for covering circles of radii determined by the final segment between A_j and A_m . The number of possible locations is small because certain inequalities in the link lengths must hold, and these inequalities can have only a small number of solutions.

Our final observation before giving the detailed outline is that we may assume A_0 is the only joint whose location is fixed. Of course by our definition of "arm", A_0 is the only joint that is fastened to the plane. However, it may be that other joints are effectively fixed for geometric reasons. For example, A_0 may be located on C , and the first link L_1 may have length equal to the diameter d of C so that the location of joint A_1 cannot change. However, it can be shown with the aid of the Normal Form Lemma 4.1 that there is a joint index j such that the location of A_i can change if, and only if, $j < i \leq n$. Furthermore, this index can be found quickly. This result takes care of regions consisting of single points and allows us to assume without loss of generality that A_0 is the only fixed joint.

We now give a detailed outline of the proof of the following theorem.

THEOREM 6.1. *For any joint A_m of an n -link arm enclosed in a circle, the boundary of the set S_m of points that A_m can reach can be covered by at most 148 circles. Descriptions of these circles can be computed in $p(n)$ steps, where p is a polynomial.*

As a notational matter, we denote a straight line of links between a joint x and a joint y by $[xy]$.

Basic circles. To define two of the four basic circles that we immediately put into the set of covering circles, recall that in § 4 we proved that the minimum and maximum distances c_m and d_m that a joint A_m can move off the circle can be computed in $p(n)$ steps, where p is a polynomial in the number of links. Call the two circles centered at the center O of circle C that have radii c_m and d_m *basic*.

To obtain the other two basic circles, note that summing the lengths of the links preceding A_m gives an upper bound for the maximum distance A_m can move from A_0 . If A_m is preceded by a link L_j that is so long that

$$l_j - \sum_{i=1, i \neq j}^m l_i > 0,$$

then this difference gives a positive lower bound for the minimum distance between A_0 and A_m ; otherwise, 0 is a bound. Call the circles centered at A_0 with these radii, which are easy to compute, *basic* also.

Facts about joints. Before continuing to build up a collection of circles covering the boundary points of S_m , we first need to observe some facts about joints. These are stated in Lemmas 6.1 through 6.3 below.

Consider a joint A_j that does not lie on the circle C . If L_j and L_{j+1} form a $0^\circ (= 360^\circ)$ or 180° angle, A_j is said to be a *straight joint* or a *fold*, respectively. If A_j does not lie on the circle and is open to any other angle, it is called an *elbow*. (It is important to note that the definition of an elbow requires that the joint not be on C .) The next lemma gives a simple but fundamental observation about elbows. The proof is a consequence of the fact that links in the tail beyond a given joint never constrain the motion of the joint along any path that stays within the minimum and maximum distances that the joint can move off the circle C . (Recall Remarks 4.1 and 4.2.)

LEMMA 6.1. *Suppose that no joint strictly between A_i and A_k lies on circle C but that some joint A_j between them is an elbow (A_i and A_k may or may not lie on C .) Then the location of A_i can be held fixed while A_k is moved to all those points in some open ball centered at A_k that do not violate the minimum and maximum distances that A_k can be located from the circle. (See Figs. 6.1a and b.)*

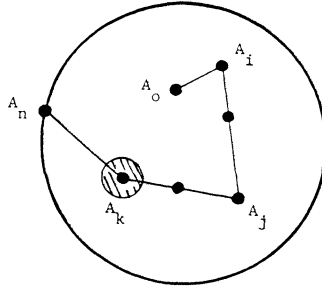


FIG. 6.1a. The elbow at A_j enables A_k to reach the points in the shaded area while the location of A_i remains fixed.

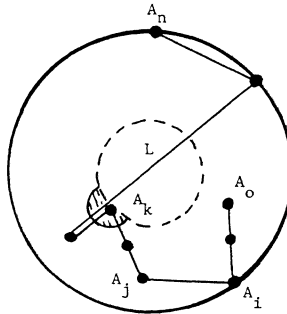


FIG. 6.1b. Link L is so long that A_k cannot reach any points inside the dashed circle.

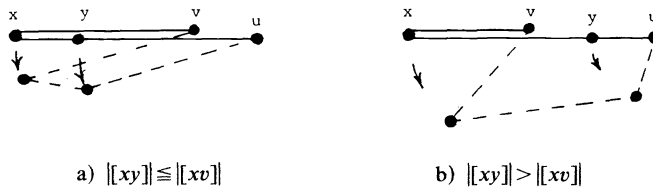
Another basic observation is that a fold can sometimes be turned into an elbow.

LEMMA 6.2. Suppose that u and v are two joints of an arm enclosed in a circle C and that all joints between u and v are straight with one exception, x , which is a folded joint not lying on C . If the lines of links $[xu]$ and $[xv]$ from x to u and x to v are not equal in length and if the longer contains at least two links, then an elbow can be created at x without changing the locations of u and v . If the lines $[xu]$ and $[xv]$ (possibly of equal length) each contain at least two links, then again, x can be turned into an elbow without moving u and v . (See Fig. 6.2.)

A final basic observation is that an elbow can be created from two folds that are joined by a straight line of links unless the line consists of a single “long” link.

LEMMA 6.3. Let u and v be joints of an arm lying inside a circle C . Suppose all joints strictly between u and v are straight with two exceptions, which are folds. Then the locations of u and v can be held fixed while the arm is moved to create an elbow between u and v unless the folds are joined by a single link that is at least as long as the sum of the lengths of all the other links between u and v .

Segments. If a configuration of the arm places A_m on the boundary of region S_m but not on one of the four basic circles, then it is not difficult to use Lemmas 6.1–6.3



a) $||[xy]|| \leq ||[xv]||$

b) $||[xy]|| > ||[xv]||$

FIG. 6.2. Creating an elbow at x .

to prove that some joint strictly between A_0 and A_m must lie on circle C . In particular, we can find some *last* joint A_j between A_0 and A_m that is on C . We will say that the links between A_j and A_m form the *final segment of the configuration before A_m* , or simply, the *final segment*. Similarly, we will say that the links between A_0 and the *first* joint beyond A_0 on C form the *initial segment* of the configuration. (Here, A_0 may or may not be on C .)

It is clear from Lemmas 6.1–6.3 that *the final segment is made up of either a straight line of one or more links from a joint on C to A_m or a single link from a joint on C to a fold that is followed by a straight line of one or more links to A_m* . In either case, the final segment lies along a line.

Recall that by Lemma 5.1, $S_j \cap C$ consists of at most two arcs of C . A routine analysis of several cases shows that if the final segment from A_j to A_m lies on a diameter of C , then R'_j , the arc of $S_j \cap C$ to which A_j belongs, consists of a single point. Using this fact, together with Lemmas 6.1–6.3, it is easy to establish the general form of a configuration that places A_m on the boundary of S_m but not on a basic circle.

LEMMA 6.4. *Suppose that an arm has been moved to a configuration that places A_m on the boundary of S_m but not on a basic circle. Let A_i be the first joint beyond A_0 on C , and let A_j be the last joint before A_m on C . Then all joints between A_0 and A_m that do not lie on C are straight, with the possible exceptions of A_{i-1} and A_{j+1} . If A_{i-1} and A_{j+1} are not straight, then they must be folds.*

This general form will help us to enumerate the remaining configurations that might have A_m on the boundary of its reachable region. Before we do this, we first observe that we can simplify the enumeration by assuming that the final segment does not lie on a diameter of C . In order to assume this, however, we must add some more circles to our collection.

Supplementary circles. If A_m lies on the boundary of S_m but not on a basic circle, and if the final segment from A_j to A_m lies on a diameter of C , then it can be shown that the initial segment consists of a single link L_1 that is connected directly to the final segment, which begins at $A_j = A_1$. The proof involves the analysis of several cases and uses Lemma 6.4 together with the fact cited previously that R'_j consists of a single point. The important consequence of this new fact is that if the final segment lies on a diameter of C , then A_m lies on one of at most four *supplementary* circles that we are about to describe. Note that in this situation, there are at most two possible locations for A_1 , corresponding to the two possible orientations for L_1 (see § 4 for the definition of orientation). Then, for a fixed position of A_1 , A_m lies on a circle centered at A_1 of radius either $\sum_{j=2}^m l_j$ or, when positive, $l_2 - \sum_{j=3}^m l_j$. This defines at most four circles, which we call *supplementary* and add to our set.

Enumerating configurations. From now on, we assume that A_m is neither on a basic circle nor on a supplementary circle so that we need only concern ourselves with situations in which *the final segment before A_m* does not lie on a diameter of C .

In order to enumerate the possible configurations of the arm between A_0 and A_m , it is useful to establish some forbidden subconfigurations. In [2] we listed six of these, two of which are shown in Fig. 6.3. These subconfigurations are forbidden because they can be moved to form elbows, which are excluded by Lemma 6.4, without changing the location of their endpoints. In Fig. 6.3a, the locations of u and x can be held fixed while $[uv]$ is rotated about u . This requires that v move closer to x , which can be accomplished by creating an elbow between v and x . Similarly in Fig. 6.3b, the locations of u and y can be held fixed while $[xy]$ is rotated about y . This requires that x move away from u , which can be accomplished by opening the joint at v while simultaneously rotating $[uv]$ about u .

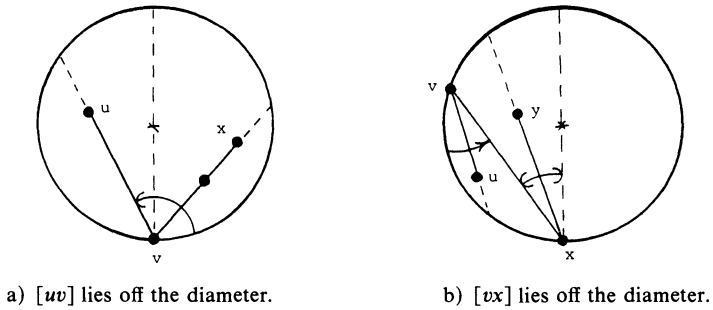


FIG. 6.3. Configurations that give rise to elbows. Arrows indicate the angular range for a line of links. A sharp tip indicates that the endpoint of the arc belongs to the range, and a round tip indicates that it does not. No order is implied by the letters at joints: u could come before or after v . There may be additional joints between the ones that appear in the figure. A dashed extension of a link indicates that its endpoint may lie on C .

By using the list of six forbidden configurations, we were able to list by careful and tedious analysis a set of ten possible configurations for the arm when the final segment contains more than one link. (Two typical ones are shown in Fig. 6.4.) Consequently the boundary of S_m can be covered by a collection of circles consisting of the basic circles (at most 4), the supplementary circles (at most 4), circles of radius l_m centered at the endpoints of R_{m-1} (at most 4) together with circles covering the ten configurations enumerated. In half of these, as in Fig. 6.4a, A_m lies on a circle of radius $l_{j+1} - |[A_{j+1}A_m]|$ centered at the end point of an arc of R_j , where A_j is the last joint on C between A_0 and A_m . In the other configurations, A_m lies on a circle of radius $|[A_jA_m]|$, where A_j has the same definition. Thus each possibility for j in each of the configurations gives rise to at most four new circles to add to the collection because R_j has at most four endpoints. Therefore, it suffices to show that the total number of possibilities for A_j is small, and that the possibilities can be determined in polynomial time. This can be done one configuration at a time.

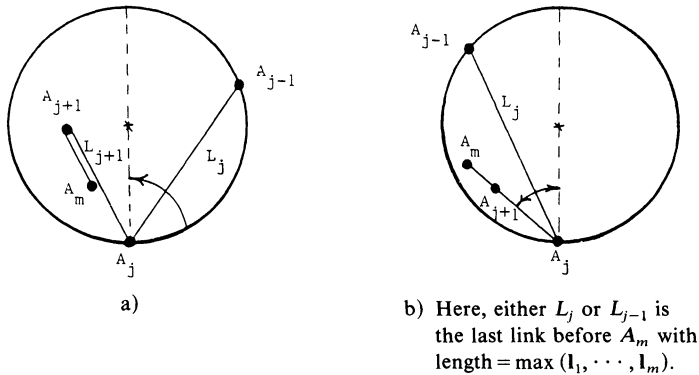


FIG. 6.4. Two of the ten possible configurations.

The basic idea for handling each configuration is this. Show that for a fixed m , there are only a constant number (independent of the arm) of possibilities for A_j , the last joint before A_m on C . Then a constant number of circles (at most 8 for the first choice of A_j) can be added to the basic and supplementary circles to form a collection that covers the boundary of S_m . This is because A_m must lie on a circle of radius either $\sum_{k=j+1}^m l_k$ or $l_{j+1} - \sum_{k=j+2}^m l_k$ about A_j , and A_j must lie at one of the endpoints of R_j , of

which there are at most four. The possibilities for A_j will be determined by inequalities involving the link lengths that can only be satisfied in a few ways.

Consider, as a simple example, the configuration in Fig. 6.4b. There are only two choices for A_j . It could be the higher indexed endpoint of the highest indexed link of longest length, or it could be the next joint after that.

As for the configuration in Fig. 6.4a, it can be shown that unless l_j or $l_{j+1} > r$, the radius of C , then the entire arm could be moved so that the configuration between A_{j-1} and A_{j+1} would go to its mirror image with respect to the initial line determined by A_{j-1} and A_{j+1} while the configuration of the rest of the arm would be restored. Since this would create elbows, it must be the case that $l_j + l_{j+1} > r$. Of course, diameter $d > l_{j+1} > \lfloor [A_{j+1}A_m] \rfloor$. If there are solutions to these inequalities, let z be the largest feasible choice for index $j+1$. Then note that there can be at most three feasible choices for $j+1$ that are smaller than z , giving a total of four choices for A_j .

The idea of moving a subconfiguration to its mirror image to show that certain inequalities must hold is used to handle several of the configurations.

Summing over all ten possible configurations listed in [2], the total number of choices for A_j is at most 34. Since R_j may have as many as four endpoints, this generates at most 136 circles. There were at most 12 circles initially, so the total number of circles needed is at most 148. This completes the outline of the proof of Theorem 6.1.

The bound of 148 is probably very generous. The important point, though, is that the boundary does not depend on the arm.

REFERENCES

- [1] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.
- [2] J. E. HOPCROFT, D. A. JOSEPH AND S. H. WHITESIDES, *Determining points of a circular region reachable by joints of a robot arm*, Computer Science Department Technical Report TR82-516, Cornell Univ., Ithaca, NY, October 1982.
- [3] T. LOZANO-PEREZ AND M. A. WESLEY, *An algorithm for planning collision-free paths among polyhedral obstacles*, Comm. ACM, 22 (1979), pp. 560-570.
- [4] T. LOZANO-PEREZ, *Automatic planning of manipulation transfer movements*, IEEE Trans. on Sys., Man. and Cyber., SMC 11 (1981), pp. 681-698.
- [5] J. REIF, *Complexity of the mover's problem and generalization*, Proc. 20th IEEE Symposium on the Foundations of Computer Science, 1979, pp. 421-427.
- [6] J. T. SCHWARTZ AND M. SHARIR, *On the "piano movers" problem I. The case of a two-dimensional rigid polygonal body moving amidst polygonal barriers*, TR39, Dept. Computer Science, New York Univ., New York, October 1981.
- [7] ———, *On the "piano movers" problem II. General techniques for computing topological properties of real algebraic manifolds*, TR41, Dept. Computer Science, New York Univ., New York, February 1982.

SOME RESULTS ON THE REPRESENTATIVE INSTANCE IN RELATIONAL DATABASES*

MINORU ITO†, MOTOAKI IWASAKI† AND TADAO KASAMI†

Abstract. Recently, the representative instance has been proposed as a generalized concept of the pure universal relation. Let $\mathbf{R} = \{\langle R_1, F_1 \rangle, \dots, \langle R_m, F_m \rangle\}$ be a database scheme, where each R_i is a set of attributes and F_i is a set of functional dependencies over R_i . \mathbf{R} is said to be consistent if the representative instance of every database $I = \{r_1, \dots, r_n\}$ of \mathbf{R} satisfies $F (= F_1 \cup \dots \cup F_m)$, that is, if whenever each relation r_i satisfies its own functional dependencies F_i , the representative instance satisfies all the functional dependencies F . In this paper, we present the following two results, which are generalizations of the previous results by [Sag2].

(1) It can be determined in $O(n|F|\|F\|)$ time whether \mathbf{R} is consistent, where $|F|$ is the number of functional dependencies in F and $\|F\|$ is the size of the description of F . (A polynomial time algorithm for determining whether \mathbf{R} is consistent is presented, independently, in [GY].)

(2) Suppose that \mathbf{R} is consistent. Given a subset V of $R_1 \cup \dots \cup R_m$, we can construct in $O(n|F|\|F\|)$ time a relational expression such that (a) its value is the total projection of the representative instance onto V for every database of \mathbf{R} , (b) it consists of projection, extension join, and union, and (c) it contains neither a redundant union nor a redundant join.

Key words. consistency, functional dependency, representative instance, total projection

1. Introduction. In the design theory of relational databases, a database scheme is defined as an ordered set $\mathbf{R} = \{\langle R_1, F_1 \rangle, \dots, \langle R_m, F_m \rangle\}$ of relation schemes, where each R_i is a set of attributes and F_i is a set of constraints over R_i . An ordered set $I = \{r_1, \dots, r_n\}$ of relations is called a database of \mathbf{R} if each r_i is a relation over R_i that satisfies F_i . In this paper, functional dependencies (FDs) [Arm], [Cod1], are treated as constraints. It has been often assumed in many papers that for a database $I = \{r_1, \dots, r_n\}$ of \mathbf{R} , there is a single relation r over the set $R_1 \cup \dots \cup R_m$ of all the attributes, called the *pure universal relation*, such that (1) r satisfies all the FDs in $F_1 \cup \dots \cup F_m$ and (2) each r_i coincides with the projection of r onto R_i . However, the pure universal relation assumption is controversial and there are some criticisms (e.g., [Ken]). Recently, the *representative instance* is proposed as a generalized concept of the pure universal relation [Hon1], [Sag1], [Sag2], [Vas]. The representative instance is based on an assumption that for a database $I = \{r_1, \dots, r_n\}$ of \mathbf{R} , there is a relation r over $R_1 \cup \dots \cup R_m$, called the *weak universal relation*, such that (1) r satisfies $F_1 \cup \dots \cup F_m$ and (2) each r_i is contained in the projection of r onto R_i . As pointed out in [MUV], the representative instance is a suitable model of the data as stored in one relation under the weak universal relation assumption. A database scheme $\mathbf{R} = \{\langle R_1, F_1 \rangle, \dots, \langle R_m, F_m \rangle\}$ is said to be *consistent* if the representative instance of every database $I = \{r_1, \dots, r_n\}$ of \mathbf{R} always satisfies $F_1 \cup \dots \cup F_m$, that is, if whenever each r_i satisfies F_i , the representative instance of I satisfies $F_1 \cup \dots \cup F_m$. (The notion of "consistency" is equivalent to the notion that "local consistency implies global consistency" in [Sag2] and the notion that " \mathbf{R} is independent with respect to $F_1 \cup \dots \cup F_m$ " in [GY].) In this paper, we consider the following two problems:

- (1) Determine whether \mathbf{R} is consistent.
- (2) Given a subset V of $R_1 \cup \dots \cup R_m$ and a database I of \mathbf{R} , how can we compute efficiently the *total projection* of the representative instance onto V ?

* Received by the editors July 14, 1982, and in revised form February 2, 1984.

† Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University, Toyonaka, Osaka 560, Japan.

The computation of the total projection is important for evaluating a query that refers to the set V with respect to the representative instance [Sag1], [Sag2], [MUV].

Sagiv [Sag2] presented some results on these problems. As for the problem (1), he presented a necessary and sufficient condition for \mathbf{R} to be consistent, called the uniqueness condition, under the restriction that each $\langle R_i, F_i \rangle$ is in Boyce-Codd normal form (BCNF), that is, the left-hand side of every FD in F_i is the key of $\langle R_i, F_i \rangle$. As for the problem (2), he presented a quadratic algorithm for constructing a relational expression whose value is the total projection of the representative instance onto V for every database I of \mathbf{R} , provided that \mathbf{R} satisfies the uniqueness condition. The expression consists of projection, extension join [Hon2], and union. Thus its value for a database of \mathbf{R} can be computed efficiently. Finally, he presented a quadratic algorithm for minimizing the number of unions and the number of joins of the expression. However, the following negative results on BCNF are known [BB].

(a) There is a universal relation scheme that cannot be transformed into any BCNF database scheme. Furthermore, it is NP-hard to determine whether a given universal relation scheme can be transformed into a BCNF database scheme.

(b) It is NP-complete to determine whether a given relation scheme $\langle R_i, F_i \rangle$ is not in BCNF (that is, whether there is a subset X of R_i such that F_i implies a nontrivial FD $X \rightarrow A$ but does not imply $X \rightarrow R_i$).

As for the problem (1), Graham et al. [GY] and, independently, the authors [IIK] presented polynomial time algorithms for determining whether \mathbf{R} is consistent with no restriction on \mathbf{R} . Furthermore, Graham et al. [GY] extended this result to the case where the join dependency $*[R_1, \dots, R_n]$ also exists. The algorithm of [GY] requires repeated tableau computations. The basic idea of our algorithm of [IIK] is essentially the same as that of theirs, but our algorithm is simpler and easier to implement, since no tableau computation is needed.

In this paper, we extend Sagiv's results to any database scheme as follows. All the results were presented in [IIK], but the proofs are simplified in this paper.

(i) Let $\mathbf{R} = \{\langle R_1, F_1 \rangle, \dots, \langle R_m, F_m \rangle\}$ be a database scheme. It can be determined in $O(n|F| \|F\|)$ time whether \mathbf{R} is consistent, where $F = F_1 \cup \dots \cup F_m$, $|F|$ is the number of FDs in F , and $\|F\|$ is the size of the description of F .

(ii) We can construct in $O(n|F| \|F\|)$ time a relational expression whose value is the total projection of the representative instance onto V for every database of \mathbf{R} , provided that \mathbf{R} is consistent. The expression consists of projection, extension join, and union. The expression can be transformed in $O(n|F| \|F\|)$ time into a simplified relational expression in the sense that it contains neither a redundant union nor a redundant join.

2. Definitions. A relation r over a set $R = \{A_1, \dots, A_v\}$ of attributes is a finite set of tuples that are members of the Cartesian product $\text{dom}(A_1) \times \dots \times \text{dom}(A_v)$, where $\text{dom}(A_i)$ is the domain of A_i . A relation can be viewed as a table whose rows are tuples, and whose columns are labeled by attributes. Let μ be a tuple in r . For an attribute A in R , $\mu[A]$ denotes the value of μ in A column. For a subset X of R , $\mu[X]$ denotes the values of μ in X columns. We use A, B, C, \dots for attributes, and \dots, X, Y, Z for sets of attributes. We often write A for the singleton set $\{A\}$, and XY for the union $X \cup Y$.

A functional dependency (FD) over R is a statement $X \rightarrow Y$, where X and Y are subsets of R [Arm], [Cod1]. A relation r is said to satisfy $X \rightarrow Y$ if for all tuples μ and ν in r , $\mu[X] = \nu[X]$ implies $\mu[Y] = \nu[Y]$. A set F of FDs is said to imply an FD f if whenever a relation satisfies F , it also satisfies the FD f . For a set X of attributes,

we define closure $(X, F) = \{A \mid F \text{ implies } X \rightarrow A\}$. We can compute $\text{closure}(X, F)$ in $O(\|F\|)$ time [BB].

In the following we often consider a relation with *variables*. That is, a tuple in a relation may contain variables in some columns. For two tuples μ and ν , $\mu[A] = \nu[A]$ if and only if μ and ν have either the same constant or the same variable in A column. We say that μ and ν *agree* in X columns if $\mu[X] = \nu[X]$. Let r be a relation that may contain variables and let F be a set of FDs. The *chase* of r under F is a relation obtained by applying FD-rules for F , which are defined below, to r until no rule can be applied anymore [ABU], [MMS]. An application sequence of FD-rules for F to r is called a *chase process* of r under F .

FD-rules. An FD $X \rightarrow Y$ in F has an associated rule as follows. Suppose that there are two tuples μ and ν which agree in X columns. FD-rule for $X \rightarrow Y$ executes the following for each attribute A in $Y - X$.

(1) If μ (or ν) has a variable v in A column and ν (or μ) has a constant c in that column, then replace all occurrences of the variable v in A column with the constant c .

(2) If μ and ν have different variables v_1 and v_2 in A column, then replace all occurrences of v_1 in A column with v_2 .

If μ and ν have different constants in A column, then μ and ν are said to *conflict* (for $X \rightarrow Y$). In this case, the chase of r under F does not satisfy F . By FD-rule for $X \rightarrow Y$, μ and ν will be equated in Y columns unless they conflict. The chase of r under F satisfies F if and only if no conflict occurs by any chase process of r under F . If the chase satisfies F , then it is unique up to renaming of variables [MMS].

A *relation scheme* is a pair $\langle R, F \rangle$ of a set R of attributes and a set F of FDs over R . A *database scheme* over a set U of attributes is an ordered set $\mathbf{R} = \{\langle R_1, F_1 \rangle, \dots, \langle R_n, F_n \rangle\}$ of relation schemes such that $U = R_1 \cdots R_n$. An ordered set $I = \{r_1, \dots, r_n\}$ of relations is called a *database* of \mathbf{R} if each r_i is a relation over R_i that satisfies F_i . We assume that no database of \mathbf{R} contains any variable. Let $I = \{r_1, \dots, r_n\}$ be a database of \mathbf{R} . Each r_i can be viewed as a relation over U by adding columns for the attributes in $U - R_i$ that contain distinct variables, as defined below. For a tuple μ in r_i , let $\text{aug}_U(\mu)$ be a tuple over U that agrees with μ in R_i columns and has distinct variables for the attributes in $U - R_i$. We define $\text{aug}_U(r_i) = \{\text{aug}_U(\mu) \mid \mu \text{ is in } r_i\}$, and $\text{aug}_U(I) = \text{aug}_U(r_1) \cup \dots \cup \text{aug}_U(r_n)$. We assume that each variable occurs once in one tuple in $\text{aug}_U(I)$. ($\text{aug}_U(r_i)$ is called the *augmentation* of r_i in [Sag1].) We denote $F = F_1 \cup \dots \cup F_n$. The *representative instance* of I , denoted $\text{rep}(I)$, is defined as the chase of $\text{aug}_U(I)$ under F [Hon1], [Sag1], [Vas]. A database scheme \mathbf{R} is said to be *consistent* if for every database I of \mathbf{R} , $\text{rep}(I)$ satisfies F . For simplicity, we assume the following.

Assumption 1. For every FD $X \rightarrow Y$ in F_i with $1 \leq i \leq n$,

- (a) $Y = \text{closure}(X, F_i) - X$, and
- (b) $X \rightarrow Y$ is not implied by $F_i - \{X \rightarrow Y\}$.

If F_i does not satisfy Assumption 1, then it can be transformed into a set satisfying the assumption in $O(\|F_i\| \|F_i\|)$ time [BB].

Assumption 2. F_1, \dots, F_n are pairwise disjoint.

If Assumption 2 does not hold, then we can see that \mathbf{R} is not consistent by Algorithm 1, which will be presented in § 3.2. In fact, suppose that F_i and F_j contain the same FD $X \rightarrow Y$. Consider a database $I = \{r_1, \dots, r_n\}$ of \mathbf{R} such that (1) r_i consists of a single tuple that has a constant c in all the columns, (2) r_j consists of a single tuple that has c exactly in X columns (and another constants in $R_j - X$ columns), and (3) any other relation is empty. (Note that if a relation is empty or consists of a single

tuple, then it satisfies any FD trivially.) A conflict for $X \rightarrow Y$ occurs in $\text{aug}_U(I)$, and thus \mathbf{R} is not consistent. Assumption 2 is used in order to uniquely identify the set F_i that contains $X \rightarrow Y$ for each $X \rightarrow Y$ in F .

3. Testing consistency of a database scheme. In this section, we present an algorithm for determining whether a given database scheme is consistent. In § 3.1, we present three lemmas that are useful for developing the algorithm. In § 3.2, we present the algorithm, and estimate its time complexity.

3.1. Conditions for consistency of a database scheme. Let $\mathbf{R} = \{\langle R_1, F_1 \rangle, \dots, \langle R_n, F_n \rangle\}$ be a database scheme over U and let $I = \{r_1, \dots, r_n\}$ be a database of \mathbf{R} . Consider a chase process of $\text{aug}_U(I)$ under F . If a tuple μ in $\text{aug}_U(I)$ is transformed into a tuple μ' by a number of applications of FD-rules for F , then μ is said to be *expanded* to μ' , which is called an *expansion* of μ . An application of FD-rule for $X \rightarrow Y$ in F_i to μ and ν that agree in X columns is said to be *restricted* if either μ or ν is an expansion of a tuple in $\text{aug}_U(r_i)$. If μ is the expansion, then ν is equated to μ in Y columns by the restricted application unless μ and ν conflict, and μ remains unchanged. Let ν' be the resulting tuple. Then ν' agrees with ν in $U - Y$ columns and agrees with μ in XY columns. We denote the restricted application by

$$\nu \xrightarrow[\mu]{X \rightarrow Y} \nu' \quad \left(\text{or simply } \nu \xrightarrow{X \rightarrow Y} \nu' \right).$$

If μ and ν conflict for $X \rightarrow Y$ in F_i (that is, if μ and ν agree in X columns but have different constants in a column in Y) and if either μ or ν is an expansion of a tuple in $\text{aug}_U(r_i)$, then the conflict is said to be *restricted*.

Example 1. Let

$$\begin{aligned} \mathbf{R} = \{ & \langle ABC, \{C \rightarrow A\} \rangle, \\ & \langle BCDE, \{B \rightarrow D, C \rightarrow DE\} \rangle, \\ & \langle ABE, \{AB \rightarrow E, E \rightarrow A\} \rangle, \\ & \langle BF, \{B \rightarrow F\} \rangle \}. \end{aligned}$$

Let $I = \{\{111\}, \{1221, 2131\}, \{112\}, \{11\}\}$. Then $\text{aug}_U(I)$ is the following relation, where $U = ABCDEF$, and v_1, \dots, v_{14} are distinct variables.

| A | B | C | D | E | F |
|----------|---|----------|----------|----------|----------|
| 1 | 1 | 1 | v_1 | v_2 | v_3 |
| v_4 | 1 | 2 | 2 | 1 | v_5 |
| v_6 | 2 | 1 | 3 | 1 | v_7 |
| 1 | 1 | v_8 | v_9 | 2 | v_{10} |
| v_{11} | 1 | v_{12} | v_{13} | v_{14} | 1 |

FD-rule for $C \rightarrow DE$ is applied to the first and third tuples to replace v_1 and v_2 with 3 and 1, respectively. This application is restricted. By applying FD-rule for $E \rightarrow A$ to the first and second tuples, v_4 is replaced with 1. This application is not restricted. FD-rule for $B \rightarrow D$ is applied to the fourth and fifth tuples to replace v_9 with v_{13} . This application is not restricted. By applying FD-rule for $B \rightarrow D$ to the second and fourth tuples, all occurrences of v_{13} are replaced with 2. This application is restricted. The result is the following relation.

| A | B | C | D | E | F |
|----------|---|----------|---|----------|----------|
| 1 | 1 | 1 | 3 | 1 | v_3 |
| 1 | 1 | 2 | 2 | 1 | v_5 |
| v_6 | 2 | 1 | 3 | 1 | v_7 |
| 1 | 1 | v_8 | 2 | 2 | v_{10} |
| v_{11} | 1 | v_{12} | 2 | v_{14} | 1 |

In this relation, the second and fourth tuples conflict for $AB \rightarrow E$. This conflict is restricted. Furthermore, the first and fifth tuples conflict for $B \rightarrow D$. This conflict is not restricted.

We have the following lemma, whose proof is given in the appendix.

LEMMA 1. *If \mathbf{R} is not consistent, then there is a database I of \mathbf{R} such that a restricted conflict occurs by a number of restricted applications of FD-rules for F to $\text{aug}_U(I)$.*

For a relation scheme $\langle R_i, F_i \rangle$, a sequence $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ of FDs in $F - F_i$ is called a *derivation* of a subset V of U from R_i if $X_k \subseteq R_i Y_1 \dots Y_{k-1}$ for $1 \leq k \leq m$ and $V \subseteq R_i Y_1 \dots Y_m$. If $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ is a derivation of V from R_i , then $R_i \rightarrow Y_1 \dots Y_m$ is derived from the sequence by an inference rule, called the *additivity rule*: “if $S \rightarrow T, Z \rightarrow W$ and $Z \subseteq ST$, then $S \rightarrow TW$ ”. Since $R_i \rightarrow Y_1 \dots Y_m$ implies $R_i \rightarrow V$, we have $V \subseteq \text{closure}(R_i, F)$. In this section, we consider the case where V is a singleton set $\{A\}$, and Y_m contains A . Such a derivation is called a derivation of A from R_i .

For an FD $X \rightarrow Y$ in F_j , we define $\text{cover}(X \rightarrow Y) = \{Z \rightarrow W \mid Z \rightarrow W \text{ is in } F_j \text{ and } ZW \subseteq XY\}$, and $\text{proper-cover}(X \rightarrow Y) = \{Z \rightarrow W \mid Z \rightarrow W \text{ is in } F_j \text{ and } ZW \subsetneq XY\}$. Let $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ be a derivation of A from R_i . For $1 \leq k \leq m$, let $X_k \rightarrow Y_k$ be in F_{j_k} , and let H_k be the intersection of F_{j_k} and $\{X_1 \rightarrow Y_1, \dots, X_{k-1} \rightarrow Y_{k-1}\}$. We define $\text{cover}(H_k) = \bigcup_{Z \rightarrow W \text{ in } H_k} \text{cover}(Z \rightarrow W)$. The derivation $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ is said to *extend minimally* if for $1 \leq k \leq m$, (1) $\text{cover}(H_k)$ contains all the FDs $X \rightarrow Y$ in $\text{proper-cover}(X_k \rightarrow Y_k)$ such that $X \subseteq R_i Y_1 \dots Y_{k-1}$, and (2) $\text{cover}(H_k)$ does not contain $X_k \rightarrow Y_k$. We say that $X \rightarrow Y$ is a *minimal* FD in G such that $X \subseteq S$ if there is no FD $Z \rightarrow W$ in G such that $Z \subseteq S$ and $ZW \subsetneq XY$. Then it follows that the sequence $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ extends minimally if every $X_k \rightarrow Y_k$ is a minimal FD in $F_{j_k} - \text{cover}(H_k)$ such that $X_k \subseteq R_i Y_1 \dots Y_{k-1}$. Note that if a sequence $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ extends minimally, then so does its subsequence $X_1 \rightarrow Y_1, \dots, X_k \rightarrow Y_k$. The last FD $X_m \rightarrow Y_m$ is said to be *irreducible* (with respect to the derivation $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ of A from R_i) if $X_m \rightarrow A$ is not implied by $\text{cover}(H_m)$.

Example 2. Let

$$\begin{aligned} \mathbf{R} = \{ & \langle ABC, \{C \rightarrow A\} \rangle, \\ & \langle ABDG, \{AB \rightarrow DG\} \rangle, \\ & \langle BCEFK, \{E \rightarrow K, BC \rightarrow EFK\} \rangle, \\ & \langle DEFGHIJ, \{E \rightarrow G, D \rightarrow H, GH \rightarrow I, DEF \rightarrow GHIJ\} \rangle. \end{aligned}$$

Consider a sequence $AB \rightarrow DG, D \rightarrow H, BC \rightarrow EFK, E \rightarrow G, GH \rightarrow I, DEF \rightarrow GHIJ$. It is a derivation of I from ABC such that $DEF \rightarrow GHIJ$ is not irreducible, since $DEF \rightarrow I$ is implied by $\text{cover}(H_6) = \{E \rightarrow G, D \rightarrow H, GH \rightarrow I\}$. Consider a sequence $BC \rightarrow EFK, E \rightarrow G, AB \rightarrow DG, D \rightarrow H, E \rightarrow K, DEF \rightarrow GHIJ$. It is a derivation of J from ABC such that $DEF \rightarrow GHIJ$ is irreducible, since $DEF \rightarrow J$ is not implied by $\text{cover}(H_6) = \{E \rightarrow G, D \rightarrow H\}$. However, the derivation does not extend minimally, since $E \rightarrow K$ is in $\text{cover}(H_5) = \{BC \rightarrow EFK, E \rightarrow K\}$. Next, consider a derivation $BC \rightarrow EFK, E \rightarrow G,$

$AB \rightarrow DG, D \rightarrow H, DEF \rightarrow GHIJ$ of J from ABC . Then $DEF \rightarrow GHIJ$ is still irreducible, but the derivation does not extend minimally, since $DEF \rightarrow GHIJ$ is not minimal in F_4 -cover $(H_s) = \{GH \rightarrow I, DEF \rightarrow GHIJ\}$. Finally, consider a derivation $BC \rightarrow EFK, E \rightarrow G, AB \rightarrow DG, D \rightarrow H, GH \rightarrow I, DEF \rightarrow GHIJ$ of J from ABC . Then $DEF \rightarrow GHIJ$ is irreducible, and the derivation extends minimally.

LEMMA 2. *If \mathbf{R} is not consistent, then for a relation scheme $\langle R_i, F_i \rangle$, there is a minimally extending derivation $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ of an attribute A from R_i such that (1) $R_i Y_1 \dots Y_{m-1}$ contains A and (2) $X_m \rightarrow Y_m$ is irreducible.*

Proof. First we prove the following claim.

Claim 1. For an FD $X \rightarrow Y$ in F_j , if $F_j - \{X \rightarrow Y\}$ implies $X \rightarrow V$, then so does proper-cover $(X \rightarrow Y)$.

If $F_j - \{X \rightarrow Y\}$ implies $X \rightarrow V$, then there is a subset $H = \{Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s\}$ of $F_j - \{X \rightarrow Y\}$ such that $V \subseteq XW_1 \dots W_s$ and $Z_t \subseteq XW_1 \dots W_{t-1}$ for $1 \leq t \leq s$ [BB]. Since H implies $X \rightarrow W_1 \dots W_s$ by the additivity rule, we have $XW_1 \dots W_s \subseteq XY$ by Assumption 1(a). Since $X \rightarrow Y$ is not in H , we have $XW_1 \dots W_s \subsetneq XY$ by Assumption 1(b). Thus Claim 1 holds.

Suppose that \mathbf{R} is not consistent. There is a database $I = \{r_1, \dots, r_n\}$ of \mathbf{R} that satisfies the condition of Lemma 1. Suppose that an expansion ν of a tuple ν_1 in $\text{aug}_U(r_i)$ restrictedly conflicts with an expansion μ of a tuple μ_m in $\text{aug}_U(r_{j_m})$ for an FD $X_m \rightarrow Y_m$ in F_{j_m} . Then ν_1 can be expanded to ν by a number of restricted applications of FD-rules for F without changing any other tuple in $\text{aug}_U(I)$, and ν restrictedly conflicts with μ_m for $X_m \rightarrow Y_m$. Thus there is a chase process

$$\nu_1 \xrightarrow[\mu_1]{X_1 \rightarrow Y_1} \nu_2 \xrightarrow[\mu_2]{X_2 \rightarrow Y_2} \dots \xrightarrow[\mu_{m-1}]{X_{m-1} \rightarrow Y_{m-1}} \nu_m$$

of $\text{aug}_U(I)$ under F such that μ_m and ν_m agree in X_m columns but have different constants in A column for an attribute A in Y_m . For $1 \leq k \leq m$, $X_k \subseteq R_i Y_1 \dots Y_{k-1}$, and ν_k has constants exactly in $R_i Y_1 \dots Y_{k-1}$ columns. Thus the sequence $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ is a derivation of A from R_i such that $R_i Y_1 \dots Y_{m-1}$ contains A . Let $X_k \rightarrow Y_k$ be in F_{j_k} , and let H_k be the intersection of F_{j_k} and $\{X_1 \rightarrow Y_1, \dots, X_{k-1} \rightarrow Y_{k-1}\}$.

Claim 2. It can be assumed without loss of generality that (a) every $X_k \rightarrow Y_k$ is in F_{j_k} -cover (H_k) and (b) μ_m and ν_m satisfy proper-cover $(X_m \rightarrow Y_m)$, if each variable is considered as a constant.

Suppose that $X_k \rightarrow Y_k$ is in cover (H_k) . There is an FD $X_l \rightarrow Y_l$ in F_{j_k} such that $l < k$ and $X_k Y_k \subseteq X_l Y_l$. Since μ_k and ν_k agree in $X_k Y_k$ columns by

$$\nu_l \xrightarrow[\mu_l]{X_l \rightarrow Y_l} \nu_{l+1},$$

we have

$$\nu_k \xrightarrow[\mu_k]{X_k \rightarrow Y_k} \nu_k.$$

Thus $X_k \rightarrow Y_k$ can be deleted from the derivation, and Claim 2(a) has been proved. Suppose that μ_m and ν_m do not satisfy proper-cover $(X_m \rightarrow Y_m)$. There is an FD $Z \rightarrow W$ in proper-cover $(X_m \rightarrow Y_m)$ such that μ_m and ν_m satisfy proper-cover $(Z \rightarrow W)$ but do not satisfy $Z \rightarrow W$. Then either (1) ν_m restrictedly conflicts with μ_m for $Z \rightarrow W$ or (2) we have

$$\nu_m \xrightarrow[\mu_m]{Z \rightarrow W} \nu',$$

and ν' restrictedly conflicts with μ_m for $X_m \rightarrow Y_m$. In the former case, if we consider the restricted conflict for $Z \rightarrow W$ instead of the one for $X_m \rightarrow Y_m$, then μ_m and ν_m satisfy proper-cover ($Z \rightarrow W$), and Claim 2(b) holds. In the latter case, if μ_m and ν' satisfy proper-cover ($X_m \rightarrow Y_m$), then Claim 2(b) holds by considering a chase process

$$\nu_1 \xrightarrow[\mu_1]{X_1 \rightarrow Y_1} \cdots \xrightarrow[\mu_{m-1}]{X_{m-1} \rightarrow Y_{m-1}} \nu_m \xrightarrow[\mu_m]{Z \rightarrow W} \nu'$$

instead of the original chase process. If μ_m and ν' do not satisfy proper-cover ($X_m \rightarrow Y_m$), then Claim 2(b) will hold by repeating the process above.

Claim 3. $F_{j_m} - \{X_m \rightarrow Y_m\}$ does not imply $X_m \rightarrow A$.

Since μ_m and ν_m agree in X_m columns, and satisfy proper-cover ($X_m \rightarrow Y_m$) by Claim 2(b) above, they agree in closure (X_m , proper-cover ($X_m \rightarrow Y_m$)) columns. Since μ_m and ν_m have different constants in A column, A is not in closure (X_m , proper-cover ($X_m \rightarrow Y_m$)), that is, proper-cover ($X_m \rightarrow Y_m$) does not imply $X_m \rightarrow A$. Thus Claim 3 follows from Claim 1 above.

Now, we prove Lemma 2. Considering Claim 2(a), we can transform the derivation $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ of A from R_i into a minimally extending derivation $P_1 \rightarrow Q_1, \dots, P_l \rightarrow Q_l$, $X_m \rightarrow Y_m$ of A from R_i by inserting some of the FDs in $\bigcup_{1 \leq k \leq m}$ proper-cover ($X_k \rightarrow Y_k$). Since $R_i Y_1 \cdots Y_{m-1}$ contains A , so does $R_i Q_1 \cdots Q_l$. Let H' be the intersection of F_{j_m} and $\{P_1 \rightarrow Q_1, \dots, P_l \rightarrow Q_l\}$. Since $X_m \rightarrow Y_m$ is not in cover (H_m) by Claim 2(a), it is not in cover (H'). Thus cover (H') does not imply $X_m \rightarrow A$ by Claim 3. By the discussions above, the sequence $P_1 \rightarrow Q_1, \dots, P_l \rightarrow Q_l$, $X_m \rightarrow Y_m$ is a minimally extending derivation of A from R_i such that $A \in R_i Q_1 \cdots Q_l$ and $X_m \rightarrow Y_m$ is irreducible. Consequently, Lemma 2 holds. \square

Conversely, we have the following lemma.

LEMMA 3. *If there is a derivation $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ of an attribute A from R_i such that (1) $R_i Y_1 \cdots Y_{m-1}$ contains A and (2) $X_m \rightarrow Y_m$ is irreducible, then \mathbf{R} is not consistent.*

Proof. First we prove the following claim.

Claim 1. If $X \rightarrow Y$ and $Z \rightarrow W$ are in F_j and $Z \subseteq XY$, then $Z \rightarrow W$ is in cover ($X \rightarrow Y$).

Since F_j implies $X \rightarrow YW$ by the additivity rule, we have $W \subseteq XY$ by Assumption 1(a). Thus the claim holds.

Let $X_m \rightarrow Y_m$ be in F_j and let H_m be the intersection of F_j and $\{X_1 \rightarrow Y_1, \dots, X_{m-1} \rightarrow Y_{m-1}\}$. We denote $H_m = \{Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s\}$. In order to prove Lemma 3, we show that there is a database I of \mathbf{R} such that a restricted conflict for $X_m \rightarrow Y_m$ occurs by expanding one tuple by $m-1$ restricted applications of FD-rules for $X_1 \rightarrow Y_1, \dots, X_{m-1} \rightarrow Y_{m-1}$ in that order. We define $I = \{r_1, \dots, r_n\}$ as follows.

(1) Every r_k except r_j consists of a single tuple that has a constant c in all the columns.

(2) $r_j = \{\mu_1, \dots, \mu_s, \mu\}$, where each tuple μ_k for $1 \leq k \leq s$ has the constant c exactly in $Z_k W_k$ columns and distinct constants (that do not appear in any other tuple) in all other columns, and μ has the constant c exactly in closure (X_m , cover (H_m)) columns and distinct constants in all other columns.

We claim that I is a database of \mathbf{R} . It suffices to show that r_j satisfies F_j . Let $X \rightarrow Y$ be an FD in F_j and let μ_k and ν be tuples in r_j that agree in X columns, where ν is one of $\mu_1, \dots, \mu_{k-1}, \mu_{k+1}, \dots, \mu_s, \mu$. Since $\mu_k[X] = \nu[X]$ implies that μ_k and ν have c in X columns, we have $X \subseteq Z_k W_k$, which implies $Y \subseteq Z_k W_k$ by Claim 1 above. Thus μ_k has c in Y columns. If $\nu = \mu_t$, then μ_t has c in Y columns by the same reason, and thus μ_k and μ_t satisfy $X \rightarrow Y$. Let $\nu = \mu$. Since (1) $X \subseteq$ closure (X_m , cover (H_m))

and (2) cover (H_m) contains $X \rightarrow Y$ by the fact that $X \subseteq Z_k W_k$ and Claim 1, cover (H_m) implies $X_m \rightarrow \text{closure}(X_m, \text{cover}(H_m)) \cup Y$ by the additivity rule, that is, $Y \subseteq \text{closure}(X_m, \text{cover}(H_m))$. Thus μ has c in Y columns, and μ_k and μ satisfy $X \rightarrow Y$. The claim has been proved.

Let τ_1 be in $\text{aug}_U(r_i)$. We claim that there is a chase process

$$\tau_1 \xrightarrow[\nu_1]{X_1 \rightarrow Y_1} \tau_2 \xrightarrow[\nu_2]{X_2 \rightarrow Y_2} \dots \xrightarrow[\nu_{m-1}]{X_{m-1} \rightarrow Y_{m-1}} \tau_m$$

such that each τ_k has c exactly in $R_i Y_1 \dots Y_{k-1}$ columns. Since $i \neq j$, initially τ_1 has c exactly in R_i columns. Suppose that τ_k has c exactly in $R_i Y_1 \dots Y_{k-1}$ columns. Let $X_k \rightarrow Y_k$ be in F_{j_k} . If $j_k = j$, then as the tuple ν_k , we can choose a tuple in $\text{aug}_U(r_j)$ that has c exactly in $X_k Y_k$ columns. If $j_k \neq j$, then ν_k has c exactly in R_{j_k} columns. Thus by

$$\tau_k \xrightarrow[\nu_k]{X_k \rightarrow Y_k} \tau_{k+1},$$

tuple τ_{k+1} has c exactly in $R_i Y_1 \dots Y_k$ columns. The claim has been proved. Since $X_m \subseteq R_i Y_1 \dots Y_{m-1}$, τ_m agrees with $\text{aug}_U(\mu)$ in X_m columns. Since cover (H_m) does not imply $X_m \rightarrow A$ by the irreducibility of $X_m \rightarrow Y_m$, $\text{aug}_U(\mu)$ does not have c in A column. However, since $A \in R_i Y_1 \dots Y_{m-1}$, τ_m has c in A column. Thus τ_m restrictedly conflicts with $\text{aug}_U(\mu)$ for $X_m \rightarrow Y_m$. \square

3.2. The method.

ALGORITHM 1.

input: A database scheme $\mathbf{R} = \{\langle R_1, F_1 \rangle, \dots, \langle R_n, F_n \rangle\}$ over U .

method: If there is a number i such that the following procedure EXAM (R_i) returns "no", then \mathbf{R} is not consistent. Otherwise, \mathbf{R} is consistent.

procedure EXAM (R_i)

begin

(1) Let $S = R_i$ (that is, assign R_i to S). For $1 \leq j \leq n$, let $G_j = \emptyset$ (that is, let G_j be empty).

(2) **while** there is a number $j (\neq i)$ such that $F_j - G_j$ contains an FD $X \rightarrow Y$ with $X \subseteq S$

do begin

(2-i) Select a minimal FD $X \rightarrow Y$ in $F_j - G_j$ such that $X \subseteq S$.

(2-ii) If $S \cap Y$ -closure (X, G_j) $\neq \emptyset$, then return "no".

(2-iii) Otherwise, let $S = S \cup Y$ and $G_j = G_j \cup \text{cover}(X \rightarrow Y)$.

end while

(3) return "yes".

end EXAM

Example 3. Let

$$\begin{aligned} \mathbf{R} = \{ & \langle ABC, \{BC \rightarrow A\} \rangle, \\ & \langle ABDE, \{A \rightarrow D, AB \rightarrow DE\} \rangle, \\ & \langle BCF, \{BC \rightarrow F, F \rightarrow C\} \rangle, \\ & \langle DEFGH, \{D \rightarrow H, DE \rightarrow GH, F \rightarrow G, H \rightarrow D\} \rangle \}. \end{aligned}$$

We execute EXAM (ABC). Initially, $S = ABC$ and $G_1 = G_2 = G_3 = G_4 = \emptyset$. Either $A \rightarrow D$ or $BC \rightarrow F$ can be selected in step (2-i). Select $BC \rightarrow F$. Then the condition of step (2-ii) does not hold, and thus we have $S = ABCF$ and $G_3 = \{BC \rightarrow F, F \rightarrow C\}$ in step (2-iii). Since $F \subseteq S$, $F \rightarrow G$ as well as $A \rightarrow D$ can be selected in step (2-i). Select

$A \rightarrow D$. It does not satisfy the condition of step (2-ii). We have $S = ABCDF$ and $G_2 = \{A \rightarrow D\}$ in step (2-iii). Since $D \subseteq S$ and $AB \subseteq S$, one of $F \rightarrow G$, $D \rightarrow H$, and $AB \rightarrow DE$ can be selected in step (2-i). Select $D \rightarrow H$. It does not satisfy the condition of step (2-ii). We have $S = ABCDFH$ and $G_4 = \{D \rightarrow H, H \rightarrow D\}$. Either $F \rightarrow G$ or $AB \rightarrow DE$ can be selected in step (2-i). Select $AB \rightarrow DE$. Though $S \cap DE = D \neq \emptyset$, $AB \rightarrow DE$ does not satisfy the condition of step (2-ii), since $\text{closure}(AB, G_2) = \text{closure}(AB, \{A \rightarrow D\}) = ABD$. Thus we have $S = ABCDEFH$ and $G_2 = \{A \rightarrow D, AB \rightarrow DE\}$. Either $F \rightarrow G$ or $DE \rightarrow GH$ can be selected in step (2-i). Select $F \rightarrow G$. It does not satisfy the condition of step (2-ii). We have $S = ABCDEFGH$ and $G_4 = \{D \rightarrow H, H \rightarrow D, F \rightarrow G\}$. Finally, we select $DE \rightarrow GH$ in step (2-i). Since $S \cap GH = GH$ and $\text{closure}(DE, G_4) = \text{closure}(DE, \{D \rightarrow H, H \rightarrow D, F \rightarrow G\}) = DEH$, we have $S \cap GH - \text{closure}(DE, G_4) = G$. Thus, $\text{EXAM}(ABC)$ returns “no” in step (2-ii). We conclude that \mathbf{R} is not consistent. In fact, consider a database $I = \{\{111\}, \{1111\}, \{111\}, \{11211, 22122\}\}$ of \mathbf{R} . We shall see that $\text{rep}(I)$ does not satisfy F .

Example 4. We execute $\text{EXAM}(ABDE)$ for the database scheme \mathbf{R} of Example 3. Initially, $S = ABDE$ and $G_1 = G_2 = G_3 = G_4 = \emptyset$. Only $D \rightarrow H$ can be selected in step (2-i). It does not satisfy the condition of step (2-ii), and thus we have $S = ABDEH$ and $G_4 = \{D \rightarrow H, H \rightarrow D\}$ in step (2-iii). Then $DE \rightarrow GH$ can be selected in step (2-i). It does not satisfy the condition of step (2-ii): We have $S = ABDEGH$ and $G_4 = \{D \rightarrow H, H \rightarrow G, DE \rightarrow GH\}$. No FD can be selected in step (2-i) anymore, and the loop of step (2) terminates. Thus $\text{EXAM}(ABDE)$ returns “yes” in step (3).

We prove the correctness of Algorithm 1. We denote the values of S, G_1, \dots, G_n at the k th execution of step (2-i) by $S^{(k)}, G_1^{(k)}, \dots, G_n^{(k)}$, respectively. We denote the FD selected at the k th execution of step (2-i) by $X^{(k)} \rightarrow Y^{(k)}$. Then we have $S^{(k)} = R_i Y^{(1)} \dots Y^{(k-1)}$ by step (2-iii). Since $X^{(k)} \subseteq S^{(k)}$, the sequence $X^{(1)} \rightarrow Y^{(1)}, \dots, X^{(k)} \rightarrow Y^{(k)}$ is a derivation of each attribute in $Y^{(k)}$ from R_i . Let $X^{(k)} \rightarrow Y^{(k)}$ be in F_{j_k} and let H_k be the intersection of F_{j_k} and $\{X^{(1)} \rightarrow Y^{(1)}, \dots, X^{(k-1)} \rightarrow Y^{(k-1)}\}$. Then $G_{j_k}^{(k)} = \text{cover}(H_k)$ by step (2-iii). Since $X^{(k)} \rightarrow Y^{(k)}$ is a minimal FD in $F_{j_k} - G_{j_k}^{(k)}$ such that $X^{(k)} \subseteq R_i Y^{(1)} \dots Y^{(k-1)}$, the sequence $X^{(1)} \rightarrow Y^{(1)}, \dots, X^{(k)} \rightarrow Y^{(k)}$ extends minimally.

Suppose that $\text{EXAM}(R_i)$ returns “no” at the k th execution of step (2-ii), that is, $S^{(k)} \cap Y^{(k)} - \text{closure}(X^{(k)}, G_{j_k}^{(k)}) \neq \emptyset$. Let A be in $S^{(k)} \cap Y^{(k)} - \text{closure}(X^{(k)}, G_{j_k}^{(k)})$. Since $\text{closure}(X^{(k)}, G_{j_k}^{(k)})$ does not contain A and $G_{j_k}^{(k)} = \text{cover}(H_k)$, $X^{(k)} \rightarrow Y^{(k)}$ is irreducible with respect to the derivation $X^{(1)} \rightarrow Y^{(1)}, \dots, X^{(k)} \rightarrow Y^{(k)}$ of A from R_i . Furthermore, since $A \in S^{(k)} = R_i Y^{(1)} \dots Y^{(k-1)}$, \mathbf{R} is not consistent by Lemma 3.

In order to prove the converse, we show two lemmas in advance.

LEMMA 4. *Let $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ be a minimally extending derivation of an attribute A from R_i such that $X_m \rightarrow Y_m$ in F_j is irreducible. For a subset G of F_j , if G does not contain $X_m \rightarrow Y_m$, then $\text{closure}(X_m, G)$ does not contain A .*

Proof. It suffices to show that $F_j - \{X_m \rightarrow Y_m\}$ does not imply $X_m \rightarrow A$. Let H_m be the intersection of F_j and $\{X_1 \rightarrow Y_1, \dots, X_{m-1} \rightarrow Y_{m-1}\}$. Since the derivation $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ extends minimally, it holds that $\text{closure}(X_m, \text{proper-cover}(X_m \rightarrow Y_m)) \subseteq \text{closure}(X_m, \text{cover}(H_m))$. (If it does not hold, then there is an FD $Z \rightarrow W$ in $F_j - \text{cover}(H_m)$ such that $ZW \subsetneq X_m Y_m$ and $Z \subseteq R_i Y_1 \dots Y_{m-1}$.) Since $X_m \rightarrow Y_m$ is irreducible, A is not in $\text{closure}(X_m, \text{cover}(H_m))$. Thus A is not in $\text{closure}(X_m, \text{proper-cover}(X_m \rightarrow Y_m))$, that is, $\text{proper-cover}(X_m \rightarrow Y_m)$ does not imply $X_m \rightarrow A$. Hence, $F_j - \{X_m \rightarrow Y_m\}$ does not imply $X_m \rightarrow A$ by Claim 1 in the proof of Lemma 2. \square

LEMMA 5. *Let $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ be a minimally extending derivation of an attribute from R_i . If $X_m \rightarrow Y_m$ is not selected in step (2-i) during the execution of $\text{EXAM}(R_i)$, then $\text{EXAM}(R_i)$ returns “no”.*

Proof. Suppose that EXAM (R_i) returns “yes”. We denote the final values of S, G_1, \dots, G_n by S', G'_1, \dots, G'_n , respectively. Then $S' = \text{closure}(R_i, F)$, since for every $X \rightarrow Y$ in F , if $X \subseteq S'$, then $Y \subseteq S'$ by the terminating condition of the loop of step (2). Since $R_i Y_1 \dots Y_m \subseteq \text{closure}(R_i, F)$, it holds that $X_k \subseteq S'$ for $1 \leq k \leq m$, and thus $X_k \rightarrow Y_k$ is in $G'_1 \cup \dots \cup G'_n$ by the terminating condition of the loop of step (2). Suppose that $X_m \rightarrow Y_m$ is not selected in step (2-i). Let $X_m \rightarrow Y_m$ be in F_j . There is an FD $X^{(k)} \rightarrow Y^{(k)}$ in F_j such that $X_m Y_m \subseteq X^{(k)} Y^{(k)}$, and that $X_m \rightarrow Y_m$ is added to G_j at the k th execution of step (2-iii). Then it holds that (1) $X^{(k)} \subseteq S^{(k)}$, (2) $X^{(k)} \rightarrow Y^{(k)}$ is a minimal FD in $F_j - G_j^{(k)}$ such that $X^{(k)} \subseteq R_i Y^{(1)} \dots Y^{(k-1)}$, and (3) $X_m \rightarrow Y_m$ is in $F_j - G_j^{(k)}$. We prove Lemma 5 by induction on the number m .

Basis. Consider the case where $m = 1$. Then $X_m \subseteq R_i$ implies $X_m \subseteq S^{(k)}$, and thus $X^{(k)} Y^{(k)} \subseteq X_m Y_m$ by the minimality of $X^{(k)} \rightarrow Y^{(k)}$. Since $X_m Y_m \subseteq X^{(k)} Y^{(k)}$, we have $X^{(k)} Y^{(k)} = X_m Y_m$. We claim that there is an attribute A in X_m such that $F_j - \{X^{(k)} \rightarrow Y^{(k)}\}$ does not imply $X^{(k)} \rightarrow A$. If there is no such attribute A , then $F_j - \{X^{(k)} \rightarrow Y^{(k)}\}$ implies $X^{(k)} \rightarrow X_m$. Since $F_j - \{X^{(k)} \rightarrow Y^{(k)}\}$ contains $X_m \rightarrow Y_m$, it implies $X^{(k)} \rightarrow X_m Y_m (= X^{(k)} Y^{(k)})$ by the additivity rule. This, however, contradicts Assumption 1(b). The claim has been proved. Note that A is in $Y^{(k)}$. Since $G_j^{(k)}$ does not contain $X^{(k)} \rightarrow Y^{(k)}$, closure $(X^{(k)}, G_j^{(k)})$ does not contain A by the claim. Furthermore, since $A \in X_m \subseteq S^{(k)}$, we have $S^{(k)} \cap Y^{(k)} - \text{closure}(X^{(k)}, G_j^{(k)}) \neq \emptyset$. Thus EXAM (R_i) returns “no” at the k th execution of step (2-ii).

Induction. If $X_m \subseteq S^{(k)}$, then EXAM (R_i) returns “no” by the same reason above. Suppose that $X_m - S^{(k)} \neq \emptyset$. Since $X_m \subseteq R_i Y_1 \dots Y_{m-1}$, there is an FD $X_p \rightarrow Y_p$ such that $p < m$ and Y_p contains an attribute A in $X_m - S^{(k)}$. Since $X_m Y_m \subseteq X^{(k)} Y^{(k)}$ and the derivation $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ extends minimally, $X_p \rightarrow Y_p$ is different from $X^{(k)} \rightarrow Y^{(k)}$. Let $X_p \rightarrow Y_p$ be the first FD in the derivation $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ such that Y_p contains A . Then $X_p \rightarrow Y_p$ is irreducible with respect to the minimally extending derivation $X_1 \rightarrow Y_1, \dots, X_p \rightarrow Y_p$ of A from R_i , since none of Y_1, \dots, Y_{p-1} contains A . By the induction hypothesis, $X_p \rightarrow Y_p$ is selected in step (2-i). Suppose that $X_p \rightarrow Y_p$ is selected at the l th execution of step (2-i). Since (1) $A \notin S^{(k)}$ but $A \in Y_p \subseteq S^{(l+1)}$ and (2) $X^{(k)} \rightarrow Y^{(k)}$ and $X_p \rightarrow Y_p$ are different, we have $k < l$, and thus $S^{(k+1)} \subseteq S^{(l)}$. Since $A \in X_m \subseteq X^{(k)} Y^{(k)} \subseteq S^{(k+1)}$, $S^{(l)}$ contains A . Let $X_p \rightarrow Y_p$ be in F_q . Since (1) $G_q^{(l)}$ does not contain $X_p \rightarrow Y_p$ and (2) $X_p \rightarrow Y_p$ is irreducible, closure $(X_p, G_q^{(l)})$ does not contain A by Lemma 4. Thus we have $S^{(l)} \cap Y_p - \text{closure}(X_p, G_q^{(l)}) \neq \emptyset$. EXAM (R_i) returns “no” at the l th execution of step (2-ii). \square

Suppose that R is not consistent. By Lemma 2, there is a minimally extending derivation $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ of A from R_i such that (1) $A \in R_i Y_1 \dots Y_{m-1}$ and (2) $X_m \rightarrow Y_m$ is irreducible. By Lemma 5, it suffices to consider the case where $X_m \rightarrow Y_m$ is selected in step (2-i). Suppose that $X_m \rightarrow Y_m$ is selected at the k th execution of step (2-i). There are two cases to be considered.

Case 1. Suppose that A is not in R_i . There is an FD $X_p \rightarrow Y_p$ such that $p < m$ and Y_p contains A . Let $X_p \rightarrow Y_p$ be the first FD such that Y_p contains A . Then $X_p \rightarrow Y_p$ is irreducible with respect to the minimally extending derivation $X_1 \rightarrow Y_1, \dots, X_p \rightarrow Y_p$ of A from R_i , since none of Y_1, \dots, Y_{p-1} contains A . By Lemma 5, it suffices to consider the case where $X_p \rightarrow Y_p$ is selected in step (2-i). Suppose that $X_p \rightarrow Y_p$ is selected at the l th execution of step (2-i). Suppose that $l < k$. Let $X_m \rightarrow Y_m$ be in F_j . Since $X_m \rightarrow Y_m$ is irreducible and not in $G_j^{(k)}$, closure $(X_m, G_j^{(k)})$ does not contain A by Lemma 4. Since $A \in Y_p \subseteq S^{(l+1)} \subseteq S^{(k)}$, we have $S^{(k)} \cap Y_m - \text{closure}(X_m, G_j^{(k)}) \neq \emptyset$. Thus EXAM (R_i) returns “no” at the k th execution of step (2-ii). The same argument applies also to the case where $k < l$.

Case 2. Suppose that A is in R_i . Since $A \in R_i \subseteq S^{(k)}$, we have $S^{(k)} \cap Y_m - \text{closure}(X_m, G_j^{(k)}) \neq \emptyset$ by the same reason above. Thus $\text{EXAM}(R_i)$ returns “no” at the k th execution of step (2-ii). This completes the correctness proof of Algorithm 1.

We estimate the time complexity of Algorithm 1. We assume that as the input of Algorithm 1, each attribute in U is represented as an integer, and each given set of attributes (e.g., R_1, \dots, R_n and X, Y for $X \rightarrow Y$ in F) is represented as an increasing sequence of integers. Before executing the procedure $\text{EXAM}(R_i)$, we execute the following (a), (b), and (c). (These can be executed in $O(\|F\| \|F\|)$ time.)

(a) For each $X \rightarrow Y$ in F_j with $1 \leq j \leq n$, list all the FDs $Z \rightarrow W$ in F_j such that $ZW \subseteq XY$, that is, cover $(X \rightarrow Y)$.

(b) For each A in U , list all the FDs $X \rightarrow Y$ in F with $A \in X$.

(c) For each $X \rightarrow Y$ in F , we introduce a variable count $(X \rightarrow Y)$, and let the initial value of count $(X \rightarrow Y)$ be $|R_i - X|$. We use count $(X \rightarrow Y)$ for examining whether S contains X .

When executing $\text{EXAM}(R_i)$, the loop of step (2) is most expensive. However, the loop is repeated at most $|F|$ times. Consider how to select an FD $X \rightarrow Y$ in step (2-i). For each execution of the loop, if an attribute A is added to S , then we decrease the value of count $(X \rightarrow Y)$ by one for each FD $X \rightarrow Y$ with $A \in X$. Note that such an FD has been listed in step (b) above. If count $(X \rightarrow Y) = 0$, then we have $X \subseteq S$. Since for each A in U , and for each $X \rightarrow Y$ in F with $A \in X$, the value of count $(X \rightarrow Y)$ is decreased by one at most once, this process can be executed in $O(\|F\|)$ time as a whole. Since we have listed cover $(X \rightarrow Y)$ for each $X \rightarrow Y$ in F_j in step (a), we can test in $O(|F_j|) \leq O(\|F\|)$ time whether $X \rightarrow Y$ is a minimal FD in $F_j - G_j$ such that $X \subseteq S$. This process can be executed in $O(\|F\|^2)$ time as a whole. Next, we can examine the condition of step (2-ii) in $O(\|F_j\|)$ time, since $\text{closure}(X, G_j)$ can be computed in $O(\|G_j\|) \leq O(\|F_j\|)$ time [BB]. Note that we examine the condition of step (2-ii) for each FD at most once. By the discussions above $\text{EXAM}(R_i)$ can be executed in $O(\|F\| \|F\|)$ time. Thus we have the following theorem.

THEOREM 1. *Let $\mathbf{R} = \{\langle R_1, F_1 \rangle, \dots, \langle R_n, F_n \rangle\}$ be a database scheme. It can be determined in $O(n\|F\| \|F\|)$ time whether \mathbf{R} is consistent, where $F = F_1 \cup \dots \cup F_n$.*

By Algorithm 1, we can determine whether a given database scheme is consistent. However, given a universal relation scheme $\langle U, F \rangle$, we do not know how to design a database scheme $\mathbf{R} = \{\langle R_1, F_1 \rangle, \dots, \langle R_n, F_n \rangle\}$ over U that is consistent, preserves F (that is, the set of FDs implied by F coincides with the one implied by $F_1 \cup \dots \cup F_n$), and has some “desired” property. We note that there is a universal relation scheme which cannot be transformed into any second normal form (2NF) database scheme [Cod2] without violating the consistency. For example, consider $\langle ABCDE, \{A \rightarrow DE, AB \rightarrow CDE, B \rightarrow E, D \rightarrow E\} \rangle$ as a universal relation scheme. It is not in 2NF, since AB is the key of the scheme but there is an FD $A \rightarrow DE$. It can be transformed into a 2NF database scheme $\mathbf{R} = \{\langle ABC, \{AB \rightarrow C\} \rangle, \langle ADE, \{A \rightarrow DE, D \rightarrow E\} \rangle, \langle BE, \{B \rightarrow E\} \rangle\}$. However, \mathbf{R} is not consistent. Note that the relation scheme $\langle ADE, \{A \rightarrow DE, D \rightarrow E\} \rangle$ is not in third normal form [Cod2].

4. Computing the total projection of the representative instance. Let r be a relation over R and let V be a subset of R . The *projection* of r onto V is defined by $r[V] = \{\mu[V] \mid \mu \text{ is in } r\}$. The *total projection* of r onto V is defined by $r[V\text{-total}] = \{\mu[V] \mid \mu \text{ is in } r, \text{ and has no variable in } V \text{ columns}\}$. Let r_1 and r_2 be relations over R_1 and R_2 , respectively. The *join* of r_1 and r_2 is a relation over $R_1 R_2$ defined by $r_1 \bowtie r_2 = \{\mu \mid \mu[R_1] \text{ is in } r_1, \text{ and } \mu[R_2] \text{ is in } r_2\}$. If r_2 satisfies $R_2 \cap R_1 \rightarrow R_2 - R_1$, then $r_1 \bowtie r_2$ is called

the *extension join*. Unlike usual joins, extension joins can be computed efficiently [Hon2].

Let $\mathbf{R} = \{\langle R_1, F_1 \rangle, \dots, \langle R_n, F_n \rangle\}$ be a database scheme over U . A *relational expression* over \mathbf{R} consists of R_1, \dots, R_n as operands and projection, join and union as operators. Formally a relational expression over \mathbf{R} is defined as follows.

(1) Each R_i is a relational expression over \mathbf{R} by itself.

(2) If E_1 and E_2 are relational expressions over \mathbf{R} , then so are $(E_1)[V]$, $(E_1) \bowtie (E_2)$, and $(E_1) \cup (E_2)$.

The value of a relational expression E over \mathbf{R} for a database $I = \{r_1, \dots, r_n\}$ of \mathbf{R} , denoted $E(I)$, is computed by assigning r_1, \dots, r_n to R_1, \dots, R_n , respectively, and applying the operators according to the definitions. For simplicity, we omit the parentheses of E in a usual manner.

In this section, we present an algorithm for constructing a relational expression E over \mathbf{R} whose value is the total projection of the representative instance onto V , that is, $E(I) = \text{rep}(I)[V\text{-total}]$ for every database I of \mathbf{R} , provided that \mathbf{R} is consistent. The expression E is of the form $\cup_i E_i[V]$, where each E_i is a sequence of extension joins, and thus $\text{rep}(I)[V\text{-total}]$ can be computed efficiently. In § 4.1, we present three lemmas that are useful for developing the algorithm. In § 4.2, we present the algorithm, and estimate its time complexity. In § 4.3, we discuss the simplification of E .

4.1. Conditions for computing the total projection. Let $Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s$ be a derivation of a subset V of U from R_i and let $H = \{Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s\}$. The derivation is said to be *minimal* if there is no FD $Z_t \rightarrow W_t$ such that $R_i \rightarrow V$ is implied by $(H - \{Z_t \rightarrow W_t\}) \cup \text{proper-cover}(Z_t \rightarrow W_t)$. Note that H implies $R_i \rightarrow V$. The following lemma implies that if \mathbf{R} is consistent, then a minimal derivation $Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s$ of V from R_i is really minimum in the sense that $R_i W_1 \dots W_s \subseteq R_i Y_1 \dots Y_m$ for every derivation $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ of V from R_i .

LEMMA 6. *Let $Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s$ be a minimal derivation of a subset V of U from R_i and let $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ be a derivation of V from R_i . If \mathbf{R} is consistent, then every $Z_t \rightarrow W_t$ is in $\text{cover}(X_k \rightarrow Y_k)$ for some $X_k \rightarrow Y_k$.*

Proof. First we prove the following claim.

Claim. $Z_s \rightarrow W_s$ is in $\text{cover}(X_k \rightarrow Y_k)$ for some $X_k \rightarrow Y_k$.

Suppose that there is no such FD $X_k \rightarrow Y_k$. We will show that \mathbf{R} is not consistent. Let $Z_s \rightarrow W_s$ be in F_j . There is an attribute A in $V \cap W_s$ such that $F_j - \{Z_s \rightarrow W_s\}$ does not imply $Z_s \rightarrow A$, as shown below.

Suppose that there is no such attribute A . Then $F_j - \{Z_s \rightarrow W_s\}$ implies $Z_s \rightarrow V \cap W_s$. By Claim 1 in the proof of Lemma 2, $\text{proper-cover}(Z_s \rightarrow W_s)$ implies $Z_s \rightarrow V \cap W_s$. Since subsequence $Z_1 \rightarrow W_1, \dots, Z_{s-1} \rightarrow W_{s-1}$ is a derivation of $Z_s(V - W_s)$ from R_i , $R_i \rightarrow V$ is implied by $\{Z_1 \rightarrow W_1, \dots, Z_{s-1} \rightarrow W_{s-1}\} \cup \text{proper-cover}(Z_s \rightarrow W_s)$. This, however, contradicts the minimality of the derivation $Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s$.

Consider a sequence $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m, Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s$. This is a derivation of A from R_i . Let H_{m+s} be the intersection of F_j and $\{X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m, Z_1 \rightarrow W_1, \dots, Z_{s-1} \rightarrow W_{s-1}\}$. Since the derivation $Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s$ is minimal, no $\text{cover}(Z_t \rightarrow W_t)$ with $t < s$ contains $Z_s \rightarrow W_s$. (If $\text{cover}(Z_t \rightarrow W_t)$ contains $Z_s \rightarrow W_s$, then subsequence $Z_1 \rightarrow W_1, \dots, Z_{s-1} \rightarrow W_{s-1}$ is a derivation of V from R_i .) Furthermore, since no $\text{cover}(X_k \rightarrow Y_k)$ contains $Z_s \rightarrow W_s$ by the assumption, $\text{cover}(H_{m+s})$ does not contain $Z_s \rightarrow W_s$. Since $F_j - \{Z_s \rightarrow W_s\}$ does not imply $Z_s \rightarrow A$, $\text{cover}(H_{m+s})$ does not imply $Z_s \rightarrow A$, that is, $Z_s \rightarrow W_s$ is irreducible. Since $A \in V \subseteq R_i Y_1 \dots Y_m$, \mathbf{R} is not consistent by Lemma 3. The claim has been proved.

The minimality of the derivation $Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s$ of V from R_i implies that subsequence $Z_1 \rightarrow W_1, \dots, Z_{s-1} \rightarrow W_{s-1}$ is a minimal derivation of $Z_s(V - W_s)$ from R_i . Since $V \subseteq R_i Y_1 \dots Y_m$ and $Z_s \rightarrow W_s$ is in cover $(X_k \rightarrow Y_k)$ for some $X_k \rightarrow Y_k$ by the claim above, sequence $X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m$ is a derivation of $Z_s(V - W_s)$ from R_i . By the claim, $Z_{s-1} \rightarrow W_{s-1}$ is in cover $(X_l \rightarrow Y_l)$ for some $X_l \rightarrow Y_l$. The same argument applies also to every $Z_t \rightarrow W_t$. Thus Lemma 6 follows. \square

LEMMA 7. *Let $Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s$ be a minimal derivation of a subset V of U from R_i and let $I = \{r_1, \dots, r_n\}$ be a database of \mathbf{R} . Suppose that \mathbf{R} is consistent. For a tuple μ_0 in $\text{aug}_U(r_i)$, if there is a chase process*

$$\mu_0 \xrightarrow[\nu_1]{X_1 \rightarrow Y_1} \dots \xrightarrow[\nu_m]{X_m \rightarrow Y_m} \mu_m$$

such that μ_m has constants in V columns, then there is a chase process

$$\mu_0 \xrightarrow{Z_1 \rightarrow W_1} \dots \xrightarrow{Z_s \rightarrow W_s} \mu'_s$$

such that μ'_s agrees with μ_m in V columns.

Proof. We show that there is a chase process

$$\mu_0 \xrightarrow{Z_1 \rightarrow W_1} \dots \xrightarrow{Z_t \rightarrow W_t} \mu'_t$$

such that μ'_t agrees with μ_m in $R_i W_1 \dots W_t$ columns by induction on the number t . This implies Lemma 7, since $V \subseteq R_i W_1 \dots W_s$.

Basis. If $t = 0$, then it holds trivially.

Induction. Suppose that there is a chase process

$$\mu_0 \xrightarrow{Z_1 \rightarrow W_1} \dots \xrightarrow{Z_{t-1} \rightarrow W_{t-1}} \mu'_{t-1}$$

such that μ'_{t-1} agrees with μ_m in $R_i W_1 \dots W_{t-1}$ columns. Since $Z_t \subseteq R_i W_1 \dots W_{t-1}$, μ'_{t-1} and μ_m have the same constants in Z_t columns. Since (1) $Z_t W_t \subseteq X_k Y_k$ for some $X_k \rightarrow Y_k$ by Lemma 6 and (2) μ_m and ν_k have the same constants in $X_k Y_k$ columns by

$$\mu_{k-1} \xrightarrow[\nu_k]{X_k \rightarrow Y_k} \mu_k,$$

tuples μ'_{t-1} and ν_k have the same constants in Z_t columns. Thus we have

$$\mu'_{t-1} \xrightarrow[\nu_k]{Z_t \rightarrow W_t} \mu'_t.$$

Clearly, μ'_t agrees with μ_m in $R_i W_1 \dots W_t$ columns. \square

For a database I of \mathbf{R} , we define that $\text{aug}_U(I)^*$ is a relation obtained by restrictedly applying FD-rules for F to $\text{aug}_U(I)$ until no variable can be replaced with any constant. We have the following lemma, whose proof is given in the appendix.

LEMMA 8. *Let I be a database of \mathbf{R} and suppose that no conflict occurs in $\text{aug}_U(I)^*$. If there is a chase process of $\text{aug}_U(I)^*$ under F such that a variable is replaced with a constant, then \mathbf{R} is not consistent.*

Lemma 8 implies that if \mathbf{R} is consistent, then $\text{rep}(I)[V\text{-total}] = \text{aug}_U(I)^*[V\text{-total}]$ for every subset V of U .

4.2. The method. Suppose that \mathbf{R} is consistent. Let $I = \{r_1, \dots, r_n\}$ be a database of \mathbf{R} and let V be a subset of U . Let $\mu[V]$ be a tuple in $\text{rep}(I)[V\text{-total}]$, where μ is

an expansion of a tuple μ_0 in $\text{aug}_U(r_i)$. By Lemma 8, μ_0 can be expanded to a tuple that agrees with μ in V columns by a number of restricted applications of FD-rules for F without changing any other tuple in $\text{aug}_U(I)$. That is, there is a chase process

$$\mu_0 \xrightarrow{X_1 \rightarrow Y_1} \cdots \xrightarrow{X_m \rightarrow Y_m} \mu_m$$

such that μ_m agrees with μ in V columns. Let $Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s$ be a minimal derivation of V from R_i . By Lemma 7, there is a chase process

$$\mu_0 \xrightarrow{Z_1 \rightarrow W_1} \cdots \xrightarrow{Z_s \rightarrow W_s} \mu'_s$$

such that μ'_s agrees with μ_m in V columns. Let $Z_t \rightarrow W_t$ be in F_{j_t} . Then μ'_s is in $r_i \bowtie r_{j_1}[Z_1 W_1] \bowtie \cdots \bowtie r_{j_s}[Z_s W_s]$. Thus $\mu[V]$ is in $(r_i \bowtie r_{j_1}[Z_1 W_1] \bowtie \cdots \bowtie r_{j_s}[Z_s W_s])[V]$. Conversely, let ν be a tuple in $r_i \bowtie r_{j_1}[Z_1 W_1] \bowtie \cdots \bowtie r_{j_s}[Z_s W_s]$. Since $V \subseteq R_i W_1 \cdots W_s$, $\nu[V]$ is in $\text{rep}(I)[V\text{-total}]$.

Let $E_i = R_i \bowtie R_{j_1}[Z_1 W_1] \bowtie \cdots \bowtie R_{j_s}[Z_s W_s]$. By the discussions above, $\mu[V]$ is in $\text{rep}(I)[V\text{-total}]$ if and only if $\mu[V]$ is in $E_i[V](I)$. Note that E_i is a sequence of extension joins. We have the following algorithm.

ALGORITHM 2.

input: a consistent database scheme $\mathbf{R} = \{\langle R_1, F_1 \rangle, \dots, \langle R_n, F_n \rangle\}$ over U , and a subset V of U .

output: a relational expression E over \mathbf{R} such that $E(I) = \text{rep}(I)[V\text{-total}]$ for every database I of \mathbf{R} .

method:

- (1) For each R_i such that $V \subseteq \text{closure}(R_i, F)$, construct a term E_i as follows. Compute a minimal derivation $Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s$ of V from R_i , where each $Z_t \rightarrow W_t$ is in F_{j_t} . Let $E_i = R_i \bowtie R_{j_1}[Z_1 W_1] \bowtie \cdots \bowtie R_{j_s}[Z_s W_s]$.
- (2) Let E be the union of all the terms $E_i[V]$, where E_i is constructed in step (1) above.

Note that every expansion of every tuple in $\text{aug}_U(r_i)$ has constants at most in $\text{closure}(R_i, F)$ columns. Thus if $\text{closure}(R_i, F)$ does not contain V , then for no expansion μ of any tuple in $\text{aug}_U(r_i)$, $\mu[V]$ is in $\text{rep}(I)[V\text{-total}]$.

Example 5. Let

$$\begin{aligned} \mathbf{R} = \{ & \langle ABC, \{AC \rightarrow B\} \rangle, \\ & \langle ABDE, \{AB \rightarrow DE, A \rightarrow D\} \rangle, \\ & \langle CF, \{C \rightarrow F\} \rangle, \\ & \langle DFGHM, \{F \rightarrow G, DF \rightarrow GM\} \rangle, \\ & \langle GIMN, \{GM \rightarrow N\} \rangle, \\ & \langle JKLP, \{J \rightarrow L\} \rangle, \\ & \langle DJM, \{J \rightarrow DM\} \rangle, \\ & \langle KOPQR, \{K \rightarrow R, P \rightarrow OQ\} \rangle, \\ & \langle NQR, \{QR \rightarrow N\} \rangle \}. \end{aligned}$$

Note that \mathbf{R} is consistent. We construct a relational expression E over \mathbf{R} such that $E(I) = \text{rep}(I)[DMN\text{-total}]$ for every database I of \mathbf{R} . Then $\text{closure}(R_1, F)$, $\text{closure}(R_4, F)$, and $\text{closure}(R_6, F)$ contain DMN , where $R_1 = ABC$, $R_4 = DFGHM$,

and $R_6 = JKLP$. For R_1 , sequence $A \rightarrow D$, $C \rightarrow F$, $DF \rightarrow GM$, $GM \rightarrow N$ is a minimal derivation of DMN from R_1 . Thus $E_1 = R_1 \bowtie R_2[AD] \bowtie R_3[CF] \bowtie R_4[DFGM] \bowtie R_5[GMN]$. Similarly, we have $E_4 = R_4 \bowtie R_5[GMN]$ for R_4 , and $E_6 = R_6 \bowtie R_8[POQ] \bowtie R_7[JDM] \bowtie R_8[KR] \bowtie R_9[QRN]$ for R_6 . Thus $E = E_1[DMN] \cup E_4[DMN] \cup E_6[DMN]$.

We consider how to find a minimal derivation of V from R_i for each R_i such that $V \subseteq \text{closure}(R_i, F)$. Suppose that in the execution of EXAM(R_i) in Algorithm 1, the loop of step (2) is repeated p times, and let $G = \{X^{(1)} \rightarrow Y^{(1)}, \dots, X^{(p)} \rightarrow Y^{(p)}\}$, where $X^{(k)} \rightarrow Y^{(k)}$ is the FD selected at the k th execution of step (2-i). A minimal derivation of V from R_i is computed by the following algorithm.

ALGORITHM 3.

(1) Let $H = G (= \{X^{(1)} \rightarrow Y^{(1)}, \dots, X^{(p)} \rightarrow Y^{(p)}\})$.

(2) for $k = p$ step - 1 until 1

do begin

(2-i) If $H - \{X^{(k)} \rightarrow Y^{(k)}\}$ implies $R_i \rightarrow V$, then delete $X^{(k)} \rightarrow Y^{(k)}$ from H .
(Otherwise, leave $X^{(k)} \rightarrow Y^{(k)}$ in H .)

end

(3) Let H_{final} be the final value of H in the loop of step (2). Construct a derivation of V from R_i by reordering the FDs in H_{final} . (Since H_{final} implies $R_i \rightarrow V$, this can be executed in $O(\|H_{\text{final}}\|)$ time by the method of [BB].)

Example 6. Consider the database scheme of Example 5. We compute a minimal derivation of DMN from $R_1 (= ABC)$. By EXAM(R_1), the sequence of the FDs selected in step (2-i) will be $C \rightarrow F$, $A \rightarrow D$, $AB \rightarrow DE$, $F \rightarrow G$, $DF \rightarrow GM$, $GM \rightarrow N$. Let H be the set of the FDs in the sequence. Since $H - \{GM \rightarrow N\}$ does not imply $R_1 \rightarrow DMN$, $GM \rightarrow N$ remains in H . Then $H - \{DF \rightarrow GM\}$ does not imply $R_1 \rightarrow DMN$, and thus $DF \rightarrow GM$ remains in H . Since $H - \{F \rightarrow G\}$ implies $R_1 \rightarrow DMN$, $F \rightarrow G$ is deleted from H , and H becomes $\{C \rightarrow F, A \rightarrow D, AB \rightarrow DE, DF \rightarrow GM, GM \rightarrow N\}$. Next, $H - \{AB \rightarrow DE\}$ implies $R_1 \rightarrow DMN$, and thus $AB \rightarrow DE$ is deleted from H , and H becomes $\{C \rightarrow F, A \rightarrow D, DF \rightarrow GM, GM \rightarrow N\}$. Then neither $A \rightarrow D$ nor $C \rightarrow F$ can be deleted from H . Thus H_{final} becomes $\{C \rightarrow F, A \rightarrow D, DF \rightarrow GM, GM \rightarrow N\}$. Note that in Example 5, we present the minimal derivation $A \rightarrow D$, $C \rightarrow F$, $DF \rightarrow GM$, $GM \rightarrow N$ of DMN from R_1 .

We prove the correctness of Algorithm 3. Suppose that the derivation obtained by Algorithm 3 is not minimal. Then there is an FD $X^{(l)} \rightarrow Y^{(l)}$ in H_{final} such that $R_i \rightarrow V$ is implied by $(H_{\text{final}} - \{X^{(l)} \rightarrow Y^{(l)}\}) \cup \text{proper-cover}(X^{(l)} \rightarrow Y^{(l)})$.

Claim. For a minimal derivation $Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s$ of a subset V of U from R_i , every FD $Z_t \rightarrow W_t$ is in G .

Since the derivation $Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s$ can be transformed into a minimally extending derivation of V from R_i by inserting some of the FDs in $\cup_{1 \leq t \leq s} \text{proper-cover}(Z_t \rightarrow W_t)$, every $Z_t \rightarrow W_t$ is selected in step (2-i) of EXAM(R_i) by Lemma 5. Thus the claim holds.

By the claim above, we assume without loss of generality that $R_i \rightarrow V$ is implied by $(H_{\text{final}} - \{X^{(l)} \rightarrow Y^{(l)}\}) \cup (\text{proper-cover}(X^{(l)} \rightarrow Y^{(l)}) \cap G)$. Let $H^{(l)}$ be the value of H when $k = l$ in the loop of step (2) of Algorithm 3. Since $X^{(k)} \rightarrow Y^{(k)}$ with $l < k$ is not in $\text{proper-cover}(X^{(l)} \rightarrow Y^{(l)})$ by step (2-iii) of EXAM(R_i) in Algorithm 1, we have $\text{proper-cover}(X^{(l)} \rightarrow Y^{(l)}) \cap G \subseteq H^{(l)}$. Since $H_{\text{final}} \subseteq H^{(l)}$, $R_i \rightarrow V$ would be implied by $H^{(l)} - \{X^{(l)} \rightarrow Y^{(l)}\}$. This, however, contradicts that $X^{(l)} \rightarrow Y^{(l)}$ is in H_{final} . This completes the correctness proof of Algorithm 3.

We estimate the time complexity of Algorithm 2. Since G is obtained in $O(\|F\| \|F\|)$ time by EXAM(R_i), a minimal derivation of V from R_i can be computed in $O(p \|G\|) \leq$

$O(|F| \|F\|)$ time by Algorithm 3. Thus Algorithm 2 can be executed in $O(n|F| \|F\|)$ time. We have the following theorem.

THEOREM 2. *Let $\mathbf{R} = \{\langle R_1, F_1 \rangle, \dots, \langle R_n, F_n \rangle\}$ be a consistent database scheme over U and let V be a subset of U . We can construct in $O(n|F| \|F\|)$ time a relational expression E over \mathbf{R} such that $E(I) = \text{rep}(I)[V\text{-total}]$ for every database I of \mathbf{R} .*

Recently, the concept of “boundedness” was proposed [MUV]. Intuitively, a database scheme $\mathbf{R} = \{\langle R_1, F_1 \rangle, \dots, \langle R_n, F_n \rangle\}$ over U is bounded for a subset V of U if for every database I of \mathbf{R} such that $\text{rep}(I)$ satisfies F , every tuple in $\text{rep}(I)[V\text{-total}]$ can be obtained by a bounded number of applications of FD-rules for F to $\text{aug}_U(I)$. By Theorem 2, if \mathbf{R} is consistent, then it is bounded for every subset V of U .

4.3. Simplification of the relational expression. Let E be the relational expression over \mathbf{R} obtained by Algorithm 2. E may contain a *redundant* term $E_i[V]$ in the sense that even if $E_i[V]$ is removed from E , the value of the resulting expression is $\text{rep}(I)[V\text{-total}]$ for every database I of \mathbf{R} .

LEMMA 9. *Suppose that E contains a term $E_i[V]$ which is of the form $(R_i \bowtie R_{j_1}[Z_1 W_1] \bowtie \dots \bowtie R_{j_s}[Z_s W_s])[V]$, where $Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s$ is a minimal derivation of V from R_i . Let $H = \{Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s\}$. Then $E_i[V]$ is redundant for E if and only if $\text{cover}(H)$ implies $Z_i W_i \rightarrow V$ for some $Z_i \rightarrow W_i$.*

Proof. Only if part. Suppose that $\text{cover}(H)$ does not imply $Z_i W_i \rightarrow V$ for any $Z_i \rightarrow W_i$. It suffices to show that there is a database I of \mathbf{R} such that $E_i[V](I)$ contains a tuple but no other $E_j[V](I)$ contains the tuple. We define $I = \{r_1, \dots, r_n\}$ as follows.

(1) r_i consists of a single tuple that has a constant c in all the columns.

(2) For $1 \leq j \leq n$ with $j \neq i$, let $\{P_1 \rightarrow Q_1, \dots, P_l \rightarrow Q_l\}$ be the intersection of F_j and H . Then let $r_j = \{\mu_1, \dots, \mu_l\}$, where each μ_k has the constant c exactly in $P_k Q_k$ columns and distinct constants in all other columns.

We can show that r_j satisfies F_j in the same way as the proof of Lemma 3. Thus I is a database of \mathbf{R} . Let ν_0 be the tuple in $\text{aug}_U(r_i)$. Clearly, there is a chase process

$$\nu_0 \xrightarrow{Z_1 \rightarrow W_1} \nu_1 \xrightarrow{Z_2 \rightarrow W_2} \dots \xrightarrow{Z_s \rightarrow W_s} \nu_s$$

such that ν_s has c exactly in $R_i W_1 \dots W_s$ columns. Thus $E_i[V](I)$ contains $\nu_s[V]$, which has c in all the columns. Suppose that $E_j[V](I)$ with $j \neq i$ contains the tuple $\nu_s[V]$. For a tuple τ_0 in $\text{aug}_U(r_j)$, there is a chase process

$$\tau_0 \xrightarrow[\delta_1]{X_1 \rightarrow Y_1} \tau_1 \xrightarrow[\delta_2]{X_2 \rightarrow Y_2} \dots \xrightarrow[\delta_m]{X_m \rightarrow Y_m} \tau_m$$

such that τ_m agrees with ν_s in V columns. Since $j \neq i$, τ_0 has c exactly in $Z_l W_l$ columns for some $Z_l \rightarrow W_l$ in H . We prove the following claim by induction on the number k .

Claim 1. τ_k has c exactly in $Z_l W_l Y_1 \dots Y_k$ columns.

The claim holds for τ_0 . Suppose that τ_{k-1} has c exactly in $Z_l W_l Y_1 \dots Y_{k-1}$ columns. If $X_k \rightarrow Y_k$ is in F_j , then δ_k has c exactly in R_i columns, and thus the claim follows from

$$\tau_{k-1} \xrightarrow[\delta_k]{X_k \rightarrow Y_k} \tau_k$$

Let $X_k \rightarrow Y_k$ be in F_p with $p \neq i$. Then δ_k has c exactly in $Z_q W_q$ columns for some $Z_q \rightarrow W_q$ in H . Since each constant except c occurs once in I and

$$\tau_{k-1} \xrightarrow[\delta_k]{X_k \rightarrow Y_k} \tau_k$$

tuple δ_k has c in X_k columns. Thus we have $X_k \subseteq Z_q W_q$, which implies that $X_k \rightarrow Y_k$ is in cover $(Z_q \rightarrow W_q)$ by Claim 1 in the proof of Lemma 3. Thus δ_k has c in Y_k columns. The claim follows from

$$\tau_{k-1} \xrightarrow[\delta_k]{X_k \rightarrow Y_k} \tau_k.$$

Let $H' = \{X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m\}$. Then we have the following claim, which contradicts that cover (H) does not imply $Z_t W_t \rightarrow V$.

Claim 2. H' implies $Z_t W_t \rightarrow V$, and is a subset of cover (H) .

Since $X_k \subseteq Z_t W_t Y_1 \dots Y_{k-1}$ by the proof of Claim 1 above, H' implies $Z_t W_t \rightarrow Y_1 \dots Y_m$ by the additivity rule. Since $V \subseteq Z_t W_t Y_1 \dots Y_m$ by Claim 1, H' implies $Z_t W_t \rightarrow V$. Next, we prove the latter half of Claim 2. Since $X_k \rightarrow Y_k$ is either in F_i or in cover $(Z_q \rightarrow W_q)$ for some $Z_q \rightarrow W_q$ by the proof of Claim 1, we have $H' \subseteq F_i \cup \text{cover}(H)$. Suppose that there is an FD $X_l \rightarrow Y_l$ in F_i . Since $X_k \subseteq Z_t W_t Y_1 \dots Y_{k-1}$ and $X_l \subseteq R_i$, sequence $Z_1 \rightarrow W_1, \dots, Z_t \rightarrow W_t, X_1 \rightarrow Y_1, \dots, X_{l-1} \rightarrow Y_{l-1}$ is a derivation of an attribute A in R_i from R_i itself. Let $P \rightarrow Q$ be the first FD in the derivation such that Q contains A . Then the subsequence $Z_1 \rightarrow W_1, \dots, P \rightarrow Q$ is a derivation of A from R_i such that (1) $A \in R_i$ and (2) $P \rightarrow Q$ is irreducible. Thus \mathbf{R} is not consistent by Lemma 3. Contradiction. The claim has been proved.

If part. Suppose that cover (H) implies $Z_t W_t \rightarrow V$. There is a subset $H' = \{X_1 \rightarrow Y_1, \dots, X_m \rightarrow Y_m\}$ of cover (H) such that $V \subseteq Z_t W_t Y_1 \dots Y_m$ and $X_k \subseteq Z_t W_t Y_1 \dots Y_{k-1}$ for $1 \leq k \leq m$ [BB]. Let $I = \{r_1, \dots, r_n\}$ be a database of \mathbf{R} , and suppose that $E_i[V](I)$ contains a tuple μ . For a tuple μ_0 in $\text{aug}_U(r_i)$, there is a chase process

$$\mu_0 \xrightarrow[\nu_1]{Z_1 \rightarrow W_1} \dots \xrightarrow[\nu_s]{Z_s \rightarrow W_s} \mu_s$$

such that $\mu_s[V] = \mu$. Since $H' \subseteq \text{cover}(H)$, we can show that there is a chase process

$$\nu_t \xrightarrow{X_1 \rightarrow Y_1} \delta_1 \xrightarrow{X_2 \rightarrow Y_2} \dots \xrightarrow{X_k \rightarrow Y_k} \delta_k$$

such that δ_k agrees with μ_s in $Z_t W_t Y_1 \dots Y_k$ columns by induction on the number k , in the same way as the proof of Lemma 7. Since $V \subseteq Z_t W_t Y_1 \dots Y_m$, δ_m agrees with μ_s in V columns. Since $E_{j_i}[V](I)$ contains $\delta_m[V]$ ($= \mu$) by Lemma 7, we have $E_i[V](I) \subseteq E_{j_i}[V](I)$. Thus $E_i[V]$ is redundant for E , and Lemma 9 has been proved. We will show a stronger result that there is a database I' of \mathbf{R} such that $E_i[V](I') \neq E_{j_i}[V](I')$. This implies that for each redundant term $E_i[V]$, there is a nonredundant term $E_{j_i}[V]$ such that $E_i[V](I) \subseteq E_{j_i}[V](I)$ for every database I of \mathbf{R} . Let $E_{j_i}[V]$ be of the form $(R_{j_i} \bowtie R_{k_1}[P_1 Q_1] \bowtie \dots \bowtie R_{k_l}[P_l Q_l])[V]$, where $P_1 \rightarrow Q_1, \dots, P_l \rightarrow Q_l$ is a minimal derivation of V from R_{j_i} . By Lemma 6, every $P_u \rightarrow Q_u$ is in cover $(X_k \rightarrow Y_k)$ for some $X_k \rightarrow Y_k$ in H' . Furthermore, since (1) $H' \subseteq \text{cover}(H)$ and (2) $\text{cover}(H)$ is disjoint from F_i , no $P_u \rightarrow Q_u$ is in F_i . Consider a database $I' = \{r_1, \dots, r_n\}$ of \mathbf{R} such that (1) every r_j except r_i consists of a single tuple that has a constant c in all the columns and (2) r_i is empty. Then $E_{j_i}[V](I')$ contains a tuple that has c in all the columns, but $E_i[V](I')$ is empty. \square

By Lemma 9 and the remark above, we can remove all the redundant terms from E . Let E' be the resulting relational expression over \mathbf{R} . Suppose that E' contains a term $E_i[V]$ which is of the form $(R_i \bowtie R_{j_1}[Z_1 W_1] \bowtie \dots \bowtie R_{j_s}[Z_s W_s])[V]$. For a database I of \mathbf{R} , if $E_i[V](I)$ contains a tuple, then all the joins of E_i are necessary in order to obtain the tuple, since $Z_1 \rightarrow W_1, \dots, Z_s \rightarrow W_s$ is minimal. Thus $E_i[V]$ contains no

redundant join. Since it can be determined in $O(s\|\text{cover}(H)\|) \leq O(|F|\|F\|)$ time whether $\text{cover}(H)$ implies $Z_i W_i \rightarrow V$ for some $Z_i \rightarrow W_i$ in H , E' can be obtained from E in $O(n|F|\|F\|)$ time. Thus we have the following corollary of Theorem 2.

COROLLARY. *The relational expression E over \mathbf{R} obtained by Algorithm 2 can be transformed in $O(n|F|\|F\|)$ time into a simplified relational expression over \mathbf{R} that contains neither a redundant union nor a redundant join.*

Example 7. Consider the term $E_1[DMN]$ in E in Example 6. E_1 is of the form $R_1 \bowtie R_2[AD] \bowtie R_3[CF] \bowtie R_4[DFGM] \bowtie R_5[GMN]$, where $A \rightarrow D$, $C \rightarrow F$, $DF \rightarrow GM$, $GM \rightarrow N$ is a minimal derivation of DMN from R_1 ($=ABC$). Let $H = \{A \rightarrow D, C \rightarrow F, DF \rightarrow GM, GM \rightarrow N\}$. Then $\text{cover}(H) = \{A \rightarrow D, C \rightarrow F, DF \rightarrow GM, F \rightarrow G, GM \rightarrow N\}$. For $DF \rightarrow GM$ in H , $\text{cover}(H)$ implies $DFGM \rightarrow DMN$. Thus $E_1[DMN]$ is redundant for E . In fact, we can show that $E_1[DMN](I) \subseteq E_4[DMN](I)$ for every database I of \mathbf{R} . Neither $E_4[DMN]$ nor $E_6[DMN]$ is redundant for E .

Appendix. Proofs of Lemmas 1 and 8. First we show two facts that are useful for the proofs. Let $\mathbf{R} = \{\langle R_1, F_1 \rangle, \dots, \langle R_n, F_n \rangle\}$ be a database scheme over U and let $I = \{r_1, \dots, r_n\}$ be a database of \mathbf{R} . We define that $\text{aug}_U(I)^*$ is a relation obtained by restrictedly applying FD-rules for F to $\text{aug}_U(I)$ until no variable can be replaced with any constant. Suppose that no restricted conflict occurs in $\text{aug}_U(I)^*$. Let $X \rightarrow Y$ be an FD in F_i , and let μ and ν be tuples in $\text{aug}_U(I)^*$ that agree in X columns. Since (1) all the variables of $\text{aug}_U(I)$ are distinct and (2) FD-rules for F are restrictedly applied to $\text{aug}_U(I)$ in order to obtain $\text{aug}_U(I)^*$, all the variables of $\text{aug}_U(I)^*$ are distinct. Thus $\mu[X] = \nu[X]$ implies that μ and ν have the same constants in X columns. Suppose that μ has constants exactly in V columns. (Then $X \subseteq V$.) Let μ_i be a tuple over R_i that agrees with μ in $V \cap XY$ columns and has distinct constants (that do not appear in $\text{aug}_U(I)^*$) in all other columns. We claim that $r_i \cup \{\mu_i\}$ satisfies F_i . Let $Z \rightarrow W$ be an FD in F_i and let τ be a tuple in r_i that agrees with μ_i in Z columns. Since $\tau[Z] = \mu_i[Z]$ implies $Z \subseteq V \cap XY$, τ also agrees with μ in Z columns. Since (1) no restricted conflict occurs in $\text{aug}_U(I)^*$ and (2) no variable can be replaced with any constant by any restricted application of FD-rule for F to $\text{aug}_U(I)^*$, $\mu[Z] = \tau[Z]$ implies $\mu[W] = \tau[W]$, that is, μ and τ have the same constants in W columns. Since $Z \subseteq V \cap XY$, we have $W \subseteq XY$ by Claim 1 in the proof of Lemma 3. Thus μ and μ_i have the same constants in W columns, which implies $\mu_i[W] = \tau[W]$. The claim has been proved. We have the following fact.

Fact 1. $I' = \{r_1, \dots, r_i \cup \{\mu_i\}, \dots, r_n\}$ is a database of \mathbf{R} .

For the database I' , we can obtain $\text{aug}_U(I)^* \cup \text{aug}_U(\mu_i)$ by a number of restricted applications of FD-rules for F to $\text{aug}_U(I')$. Since $\mu[X] = \nu[X] = \text{aug}_U(\mu_i)[X]$, we have the following fact.

Fact 2. (a) FD-rule for $X \rightarrow Y$ can be restrictedly applied to μ and $\text{aug}_U(\mu_i)$.

(b) If there is an attribute A in Y such that $\mu[A] \neq \nu[A]$, and $\nu[A]$ is a constant, then ν and $\text{aug}_U(\mu_i)$ restrictedly conflict for $X \rightarrow Y$. Otherwise, FD-rule for $X \rightarrow Y$ can be restrictedly applied to ν and $\text{aug}_U(\mu_i)$.

LEMMA 1. *If \mathbf{R} is not consistent, then there is a database I of \mathbf{R} such that a restricted conflict occurs by a number of restricted applications of FD-rules for F to $\text{aug}_U(I)$.*

Proof. Suppose that \mathbf{R} is not consistent. There is a database $I = \{r_1, \dots, r_n\}$ of \mathbf{R} such that a conflict occurs by a chase process of $\text{aug}_U(I)$ under F . Without loss of generality, we consider a chase process that computes $\text{aug}_U(I)^*$ and then applies FD-rules for F to $\text{aug}_U(I)^*$ until a conflict occurs. Suppose that a conflict occurs by k applications of FD-rules for F to $\text{aug}_U(I)^*$. We prove Lemma 1 by induction on the number k .

Basis. Let $k = 0$. That is, a conflict occurs in $\text{aug}_U(I)^*$. If the conflict is restricted, then Lemma 1 follows. Suppose that no restricted conflict occurs in $\text{aug}_U(I)^*$. For an FD $X \rightarrow Y$ in F_i , there are two tuples μ and ν in $\text{aug}_U(I)^*$ that agree in X columns but have different constants in a column in Y . If we consider the database I' of \mathbf{R} defined in Fact 1, then Lemma 1 follows from Fact 2(b).

Induction. Let $k \geq 1$. Let the first (nonrestricted) application of FD-rule for F to $\text{aug}_U(I)^*$ be the one for $X \rightarrow Y$ in F_i to μ and ν . Consider the database I' of \mathbf{R} defined in Fact 1. When we compute $\text{aug}_U(I')^*$, the first nonrestricted application can be replaced with two restricted applications of FD-rules for $X \rightarrow Y$ to μ and $\text{aug}_U(\mu_i)$ (and ν and $\text{aug}_U(\mu_i)$) by Fact 2. Since $\text{aug}_U(I') = \text{aug}_U(I) \cup \text{aug}_U(\mu_i)$, for every tuple δ in $\text{aug}_U(I)^*$, there is an expansion of δ in $\text{aug}_U(I')^*$. Thus a conflict occurs by at most $k-1$ applications of FD-rules for F to $\text{aug}_U(I')^*$. From the induction hypothesis, Lemma 1 follows. \square

LEMMA 8. Let $I = \{r_1, \dots, r_n\}$ be a database of \mathbf{R} and suppose that no conflict occurs in $\text{aug}_U(I)^*$. If there is a chase process of $\text{aug}_U(I)^*$ under F such that a variable is replaced with a constant, then \mathbf{R} is not consistent.

Proof. An application of FD-rule for $X \rightarrow Y$ in F to two tuples in r is said to be *minimal* if r satisfies proper-cover($X \rightarrow Y$) (by considering each variable of r as a constant). If a variable v is replaced with a constant c by a chase process of $\text{aug}_U(I)^*$ under F , then v can be replaced with c by a number of minimal applications of FD-rules for F to $\text{aug}_U(I)^*$. Suppose that a variable v is replaced with a constant c by k minimal applications of FD-rules for F to $\text{aug}_U(I)^*$. We prove Lemma 8 by induction on the number k .

Basis. Let $k = 1$. That is, for an FD $X \rightarrow Y$ in F_i , there are two tuples μ and ν in $\text{aug}_U(I)^*$ such that $\mu[X] = \nu[X]$, $\mu[A] = v$, and $\nu[A] = c$ for an attribute A in Y . If we consider the database I' of \mathbf{R} defined in Fact 1, then Lemma 8 follows from Fact 2(b).

Induction. Let $k \geq 2$. Let the first minimal (nonrestricted) application of FD-rule for F to $\text{aug}_U(I)^*$ be the one for $X \rightarrow Y$ in F_i to μ and ν . We assume that no variable can be replaced with any constant by any application of FD-rule for any FD in F_i to any two tuples in $\text{aug}_U(I)^*$. (Otherwise, Lemma 8 follows from the basis above.) For the database I' of \mathbf{R} that is defined in Fact 1, consider the following chase process of $\text{aug}_U(I')$ under F .

- (1) Compute $\text{aug}_U(I)^* \cup \text{aug}_U(\mu_i)$.
- (2) Expand $\text{aug}_U(\mu_i)$ by restrictedly applying FD-rules for F until it can not be expanded anymore. Let μ' be the resulting tuple. (We have $\text{aug}_U(I)^* \cup \{\mu'\}$ by step (2).)
- (3) Expand each tuple τ in $\text{aug}_U(I)^*$ by a restricted application of FD-rule for an FD in F_i to τ and μ' . Let r be the resulting relation.

If a conflict occurs in r , then Lemma 8 follows. Suppose that no conflict occurs in r . We prove the following claims.

Claim 1. $r = \text{aug}_U(I')^*$.

Claim 2. v is not replaced with c by the chase process above.

If a tuple in $\text{aug}_U(I)^*$ remains unchanged in step (3) above, then it can not be expanded by any restricted application of FD-rule for F . Suppose that there is a restricted application in step (3) such that

$$\tau \xrightarrow[\mu']{Z \rightarrow W} \tau'$$

for $Z \rightarrow W$ in F_i . First we show that $ZW = XY$.

Since $\text{aug}_U(I)^*$ satisfies proper-cover($X \rightarrow Y$) by the minimality of the application of FD-rule for $X \rightarrow Y$, so does $\text{aug}_U(I)^* \cup \text{aug}_U(\mu_i)$. Thus r satisfies proper-cover($X \rightarrow$

Y). Thus $Z \rightarrow W$ is not in proper-cover ($X \rightarrow Y$). Since μ_i has new constants (that do not appear in $\text{aug}_U(I)^*$) in $R_i - (V \cap XY)$ columns by the definition, $\tau[Z] = \mu'[Z]$ implies $Z \subseteq XY$, and thus $Z \rightarrow W$ is in cover ($X \rightarrow Y$) by Claim 1 in the proof of Lemma 3. Hence $Z \rightarrow W$ is in cover ($X \rightarrow Y$) - proper-cover ($X \rightarrow Y$), which implies $ZW = XY$.

Since τ' has constants in $ZW (= XY)$ columns by

$$\tau \xrightarrow[\mu']{Z \rightarrow W} \tau',$$

it can not be expanded by any restricted application of FD-rule for cover ($X \rightarrow Y$). Suppose that τ has constants exactly in P columns. Then τ' has constants exactly in PW columns by

$$\tau \xrightarrow[\mu']{Z \rightarrow W} \tau'.$$

Since no variable can be replaced with any constant by any application of FD-rule for any FD in F_i to τ and μ by the assumption, τ' has new constants (that do not appear in $\text{aug}_U(I)^*$) in $PW - P$ columns. (This implies Claim 2.) Thus τ' can not be expanded by any restricted application of FD-rule for F - cover ($X \rightarrow Y$). Thus Claim 1 has been proved.

If v has been replaced with a constant c' with $c' \neq c$ in r , then a conflict must occur by a chase process of r under F . Thus Lemma 8 follows. Suppose that v is not replaced with any constant in r . When we compute $r (= \text{aug}_U(I)^*)$, the first nonrestricted application can be replaced with two restricted applications of FD-rules for $X \rightarrow Y$ to μ and $\text{aug}_U(\mu_i)$ (and ν and $\text{aug}_U(\mu_i)$) by Fact 2. Since $\text{aug}_U(I) = \text{aug}_U(I) \cup \text{aug}_U(\mu_i)$, for every tuple δ in $\text{aug}_U(I)^*$, there is an expansion of δ in $\text{aug}_U(I)^*$. Thus v can be replaced with c by at most $k - 1$ minimal applications of FD-rules for F to r . From the induction hypothesis, Lemma 8 follows. (The reason we consider minimal applications of FD-rules is that if the first nonrestricted application is not minimal, then v may have been replaced with c in $\text{aug}_U(I)^*$.) \square

REFERENCES

- [ABU] A. V. AHO, C. BEERI AND J. D. ULLMAN, *The theory of joins in relational databases*, ACM Trans. Database Syst., 4 (1979), pp. 297-314.
- [Arm] W. W. ARMSTRONG, *Dependency structures of database relationships*, Proc. IFIP 74, North-Holland, Amsterdam, 1974, pp. 580-583.
- [BB] C. BEERI AND P. A. BERNSTEIN, *Computational problems related to the design of normal form relational schemas*, ACM Trans. Database Syst., 4 (1979), pp. 30-59.
- [Cod1] E. F. CODD, *A relational model of data for large shared data banks*, Comm. ACM, 13 (1970), pp. 377-387.
- [Cod2] ———, *Further normalization of the data base relational model*, Data Base Systems, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, 1972.
- [GY] M. H. GRAHAM AND M. YANNAKAKIS, *Independent database schemas* (extended abstract), Proc. ACM Symposium on Principles of Database Systems, March 1982, pp. 199-204.
- [Hon1] P. HONEYMAN, *Testing satisfaction of functional dependencies*, J. Assoc. Comput. Mach., 29 (1982), pp. 668-677.
- [Hon2] ———, *Extension joins*, Proc. Int. Conf. Very Large Data Bases, 1980, pp. 239-244.
- [IIK] M. IWASAKI, M. ITO AND T. KASAMI, *A result on the representative instance in relational databases*, Tech. Rep. of Languages and Automata Symposium, Kyoto, Feb. 1982. (In Japanese.)
- [Ken] W. KENT, *Consequences of assuming a universal relation*, ACM Trans. Database Syst., 6 (1981), pp. 539-556.
- [MMS] D. MAIER, A. O. MENDELZON AND Y. SAGIV, *Testing implications of data dependencies*, ACM Trans. Database Syst., 4 (1979), pp. 455-469.

- [MUV] D. MAIER, J. D. ULLMAN AND M. Y. VARDI, *On the foundations of the universal relation model*, unpublished manuscript.
- [Sag1] Y. SAGIV, *Can we use the universal instance assumption without using nulls?* Proc. ACM SIGMOD International Conference on Management of Data, April 1981, pp. 108-120.
- [Sag2] ———, *A characterization of globally consistent databases and their correct access paths*, ACM Trans. Database Syst., 8 (1983), pp. 266-286.
- [Vas] Y. VASSILIOU, *A formal treatment of imperfect information in database management*, TR CSRG-123, Univ. Toronto, Toronto, Ontario, 1980.

RECTILINEAR GRAPHS AND THEIR EMBEDDINGS*

GOPALAKRISHNAN VIJAYAN† AND AVI WIGDERSON‡

Abstract. The embedding problem for a class of graphs called rectilinear graphs is discussed. These graphs have applications in many VLSI Layout Problems. An interesting topological characterization of these graphs lead to efficient algorithms for recognizing and embedding rectilinear graphs which are embeddable on the plane.

Key words. graphs, embeddings of graphs, VLSI layouts, algorithms for graph embedding

1. Introduction. The problem we address in this paper is an embedding problem for a class of graphs which we call rectilinear graphs. These graphs are important in many VLSI layout problems. In fact, this problem arose in the implementation of ALI [7], [8], a procedural language for VLSI design currently under development at Princeton. An embedding algorithm can be used to automate the production of VLSI layouts in many procedural design systems.

The following is an informal description of rectilinear graphs and their embeddings. The vertices of a rectilinear graph have degree at most four. The edges incident on each vertex are given distinct labels from the set {Left, Right, Up, Down}. Suppose we place the vertices on the grid points of a rectangular grid, and for each edge draw a straight line segment between its endpoints. We call the result an embedding of the graph, if the edges lie along grid lines, no two edges cross, and the directions of the edges at each vertex are consistent with their labels.

Consider the following model for VLSI layout design. A VLSI layout is described hierarchically using cells and wires that connect the cells together. Each cell C is enclosed within a rectangle $R(C)$, and has four lists of pins, one each for the left, top, right, and bottom of rectangle $R(C)$. Each wire w is denoted by a pair of pins (p_i, p_j) , such that p_i and p_j are pins of different rectangles, and are of opposite types. For example, if p_i is a right pin then p_j should be a left pin. Given such a description of a VLSI layout, our aim is to produce an embedding of the description on the plane, such that (i) no two bounding rectangles touch each other, (ii) the pins appear in the correct order on the bounding rectangles, (iii) the wires are straight and rectilinear, and (iv) no two wires cross each other. Later on, we can fill each bounding rectangle $R(C)$ with the embedding of the cell C in the same manner.

The restriction that wires cannot be bent may seem unrealistic, but this is certainly the case in many design systems including ALI. If a wire has to be bent, the user specifies that by breaking up the wire into several straight wires and placing cells at each of the turn points of the wire. In ALI, for example, the user can incorporate routing algorithms in a ALI program to determine how the wires are to be bent. The restriction that wires cannot cross implies that we are dealing with the wires on a single layer. For a layout with multiple layers, it is clearly necessary that the wires on each layer do not cross.

It is easy to observe that the above description of a layout induces a rectilinear graph, whose vertices are the pins and the corners of the bounding rectangles, and

* Received by the editors December 22, 1982, and in revised form December 6, 1983.

† School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332. The research of this author was supported in part by DARPA under grant N0014-82-K-0549.

‡ Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720. The research of this author was supported in part by an IBM fellowship.

whose edges are the wires and the segments created on the bounding rectangles by the vertices.

For VLSI applications, we need efficient algorithms to recognize and then actually embed rectilinear graphs. In this paper, we present an $O(n)$ recognition algorithm and an $O(n^2)$ embedding algorithm, where n is the number of vertices in the graph. Thus, a hierarchically described VLSI layout with cell instances C_1, C_2, \dots, C_m can be embedded in time $O(\sum_{i=1}^m n_i^2)$, where n_i is the number of pins in cell instance C_i .

An embedding of a rectilinear graph is just a relative placement of the vertices (cells) on a rectangular grid, such that no two edges cross. Some of the relative placement information is already present in the description of a rectilinear graph. For example, if (a, b) is the rightgoing edge of vertex a , then a should be to the left of b , and a, b should be on the same horizontal grid line. Hence, an embedding can be viewed as a “completion” of the rectilinear graph description. We showed in a different paper [10] that the completion problem for a slightly more relaxed VLSI layout model is NP-complete. In light of this result, the results in this paper have become more important.

In § 2, we present formal definitions of rectilinear graphs and thier embeddings. In § 3, we mention some properties of rectilinear graphs. We discuss some topological properties of the embeddings in § 4. A necessary and sufficient condition for biconnected rectilinear graphs to be embeddable is presented in § 5. A similar condition for arbitrary rectilinear graphs is the main result in § 6. We also describe a $O(n)$ recognition algorithm in this section. In § 7, we use the ideas of the previous sections to obtain an $O(n^2)$ embedding algorithm. An important subclass of rectilinear graphs is discussed in § 8. In § 9, we discuss extensions and open problems. For definitions of graph theoretic terminology used in this paper, please refer to [1], [2].

2. Definition of the problem. First we give a formal definition of a rectilinear graph.

DEFINITION 2.1. A *rectilinear graph* G is a triple (V, E, λ) , where V is the vertex set, E is the edge set, and

$$\lambda : V \times V \rightarrow \Sigma \cup \{\varepsilon\}, \quad \text{where } \Sigma = \{L, R, D, U\}$$

is a *vertex ordering relation* with the following properties:

for every $a, b, c \in V$ and $X \in \Sigma$

- (i) $\lambda((a, b)) = \varepsilon \Leftrightarrow \{a, b\} \notin E$
(ordering is specified only between adjacent vertices);
- (ii) $\lambda((a, b)) = L \Leftrightarrow \lambda((b, a)) = R, \lambda((a, b)) = D \Leftrightarrow \lambda((b, a)) = U;$
- (iii) $\lambda((a, b)) = X \Rightarrow \lambda((c, b)) \neq X, \forall c \neq a$ (no overlapping edges).

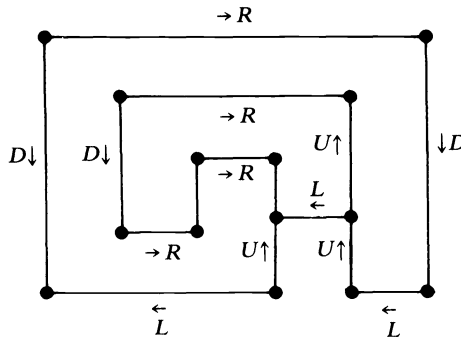


FIG. 2.1. A *rectilinear graph*.

Each vertex in a rectilinear graph has degree at most four, and each edge (a, b) , as it goes from one vertex a to another b , has a nonempty label on it, which in the embedding will indicate the direction (left, right, down, or up) in which the edge leaves a vertex a . There can be at most one edge with a particular label emanating from each vertex. The undirected graph $G(V, E)$ will be referred to as the *underlying graph*. Figure 2.1 (like all other figures) gives an illustration of a rectilinear graph.

Now we define what sort of an embedding we are looking for.

DEFINITION 2.2. An *embedding* of a rectilinear graph $G(V, E, \lambda)$ on a *rectangular grid* is given by two mappings $x, y: V \rightarrow Z$ (the integers) which are the x and y coordinates respectively of the vertices. These mappings obey:

1. *The ordering relation, λ* , i.e. for all edges $\{a, b\} \in E$

$$\lambda((a, b)) = L \Rightarrow y(a) = y(b), x(a) > x(b),$$

$$\lambda((a, b)) = R \Rightarrow y(a) = y(b), x(a) < x(b),$$

$$\lambda((a, b)) = D \Rightarrow x(a) = x(b), y(a) > y(b),$$

$$\lambda((a, b)) = U \Rightarrow x(a) = x(b), y(a) < y(b).$$

2. *Planarity*, no two edges cross, i.e. for each pair of nonadjacent edges $\{a, b\}, \{c, d\}$ such that $\lambda((a, b)) = R$ and $\lambda((c, d)) = U$, the relation

$$x(a) \leq x(c) \leq x(b) \quad \text{and} \quad y(c) \leq y(a) \leq y(d)$$

does not hold.

An embedding of a rectilinear graph on a rectangular grid is one in which the vertices are placed at grid points, the edges run along grid lines in the directions given by their labels, and no two edges cross each other except if they share a vertex. Also, an edge cannot touch a vertex unless it is incident on it. We say that a rectilinear graph is embeddable if it has an embedding. We will show in the next section that not all rectilinear graphs are embeddable.

Now our main problem can be stated simply: Given a rectilinear graph $G(V, E, \lambda)$, is it embeddable, and if it is, find an embedding.

3. Some comments on rectilinear graphs. In this section we list some properties of rectilinear graphs and their embeddings. Some of these properties will give an indication of why our problem is different from other embedding problems, in particular, planar graph embedding [3], [6].

1. Embeddability is a hereditary property. Subgraphs are defined in the usual fashion, but here the labels of edges are inherited. This is obvious, but worth mentioning, because this will be used in the proofs.

2. If each connected component of a rectilinear graph is embeddable then the graph itself is embeddable. So, without loss of generality we will restrict ourselves to connected rectilinear graphs.

3. Rectilinear graphs with nonplanar underlying graphs are clearly not embeddable. So it is not interesting to consider those graphs. However, not every rectilinear graph with a planar underlying graph is embeddable. In Fig. 3.1, we have two simple cycles which are not embeddable.

4. In contrast with planarity, embeddability is *not* a property determined by the biconnected components. Fig. 3.2 provides an illustration of this fact.

5. This problem is a restriction of an NP-complete problem [10], [12]. For each wire w , we are given its orientation (horizontal or vertical), and a set V_w of vertices.

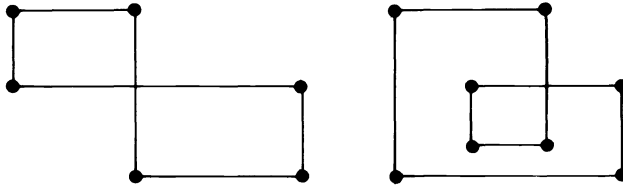


FIG. 3.1. Two nonembeddable rectilinear cycles.

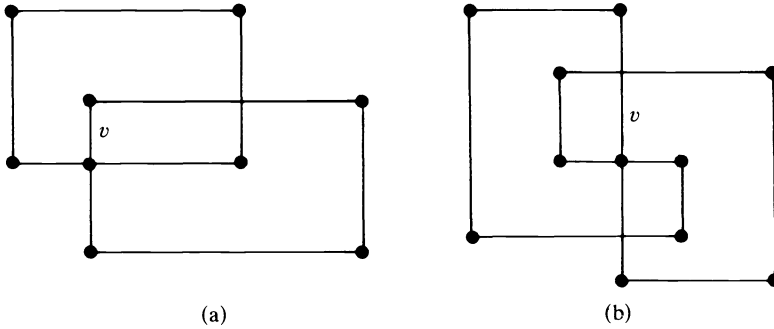


FIG. 3.2. Two nonembeddable rectilinear graphs whose biconnected components are embeddable.

The wire w has to touch each vertex in the set V_w (the vertices could be touched in any order). Then, the embedding problem becomes NP-complete.

6. If we relax the rectilinearity of the edges and impose only the cyclic ordering of the edges at each vertex, then there is an $O(|V|)$ algorithm [11]. The cyclic orderings automatically determines the faces of the embedding (if one exists). Thus a embeddable rectilinear graph has a unique embedding in this sense.

4. Topological structure of embeddings. There is a natural way to extend the function λ to paths and cycles in the graph as follows. Given a path $P = (v_0, v_1, \dots, v_t)$ we define $\lambda(P) = \lambda((v_0, v_1))\lambda((v_1, v_2)) \cdots \lambda((v_{t-1}, v_t))$. We define a similar extension for cycles where now $v_t = v_0$. λ becomes a mapping that associates with each path or cycle in the graph a string in Σ^* which is the concatenation of labels along the path or cycle. Note that strings containing RL, DU, LR, UD as substrings do not represent paths. Also the direction in which we traverse a path and the starting point in a cycle are important. An example of this mapping can be found in Fig. 4.1.

Next we define two topological operations on rectilinear graphs. These operations will simplify a rectilinear graph while preserving its topological structure. Let G be a rectilinear graph.

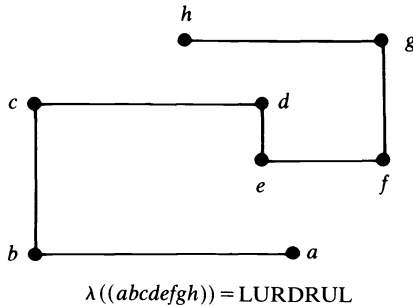


FIG. 4.1. The extension of λ to paths.

Operation 1—Edge contraction. Let $(abcd)$ be a path in G such that both b and c have degree 2, and $\lambda((abcd)) = XYX$ where $X, Y \in \Sigma$. Contract the edge (b, c) to the vertex b . The resulting path (abd) will have $\lambda((abd)) = XX$. We abbreviate this operation by $XYX \rightarrow XX$ (Fig. 4.2(1)).

Operation 2—Vertex deletion. Let (abc) be a path in G such that vertex b has degree 2, and $\lambda((abc)) = XX$ where $X \in \Sigma$. Delete the vertex b and introduce the edge (a, c) . The resulting edge (a, c) will have $\lambda((a, c)) = X$. We abbreviate this operation by $XX \rightarrow X$ (Fig. 4.2(2)).

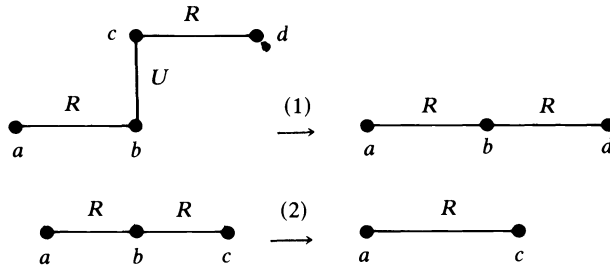


FIG. 4.2. Edge contraction and vertex deletion.

In a natural way we can define inverses for the above two operations which we will refer to as *edge expansion* and *vertex addition* respectively.

LEMMA 4.1. *Let G be a rectilinear graph and G' be the graph resulting from G by the application of a sequence of the above four operations. Then G' is also rectilinear and moreover G' is embeddable if and only if G is embeddable.*

Proof. The proof is easy and is left to the reader. \square

DEFINITION 4.1. Given a string $\gamma \in \Sigma^*$ representing a path or a cycle, the *simplified form* $\tilde{\gamma}$ of γ is obtained by repeatedly applying the reduction rules $XYX \rightarrow XX$ and $XX \rightarrow X$, where $X, Y \in \Sigma$ until they cannot be applied any more. If γ represents a cycle then it is treated as a cyclic string.

In Fig. 4.3 we give a path and a cycle along with their simplified forms.

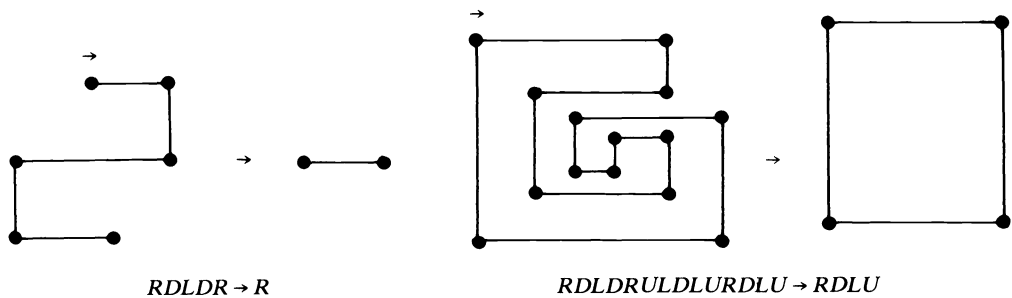


FIG. 4.3. Simplification of a path and a cycle.

LEMMA 4.2. *Every string $\gamma \in \Sigma^*$ has a unique simplified form.*

Proof. The replacement system defined by the two reduction rules have the Church-Rosser property [9]. \square

DEFINITION 4.2. A *square* is one of the cyclic strings $LURD$ or $LDRU$.

Sometimes we may distinguish between two squares by their starting labels.

DEFINITION 4.3. A *spiral* is a substring of $(LURD)^+$ or $(LDRU)^+$.

LEMMA 4.3. *Every path is embeddable.*

Proof. Every spiral is embeddable. Since any path simplifies to a spiral, by Lemma 4.1 it is also embeddable. \square

So it is the cycles which make the problem nontrivial. The following lemma is a crucial fact about cycles.

LEMMA 4.4. *A cycle is embeddable if and only if it simplifies to a square.*

Proof. If. A square is embeddable and hence by Lemma 4.1 any cycle which simplifies to a square is also embeddable.

Only if. Let f be an embeddable cycle and $\lambda(f) = \gamma$. By Lemma 4.1, the cycle defined by $\bar{\gamma}$ is also embeddable. Let $|\bar{\gamma}| = n$. Look at the embedding of $\bar{\gamma}$. Since it has no crossings the embedding is a simple polygon. Therefore the interior angles of this polygon sum to $(n-2) \times 180^\circ$. Since $\bar{\gamma}$ is a spiral all its interior angles are 90° . The only solution to $n \times 90 = (n-2) \times 180$ is $n = 4$. Therefore $\bar{\gamma}$ is a square. \square

The proof of the previous lemma suggests another useful characterization of embeddable cycles. Going along a cycle $f = v_1 v_2 \cdots v_n v_1$ in the counterclockwise direction, let us denote by $\varphi_f(v_i)$ the angle at vertex v_i , which is the angle between (v_{i-1}, v_i) and (v_i, v_{i+1}) , and by $\varphi(f)$ the sum of these angles.

LEMMA 4.5. *A cycle $f = v_1 v_2 \cdots v_n v_1, n \geq 4$ is embeddable if and only if $\varphi(f) = \sum_{i=1}^n \varphi_f(v_i) = (n \pm 2) \times 180^\circ$.*

Proof. Suppose f is embeddable, then its embedding is a simple polygon. Depending on whether we sum the interior angles or exterior angles we should get $(n \pm 2) \times 180^\circ$.

To prove the sufficient part we show by induction on n that f simplifies to a square. The possible values for $\varphi_f(v_i)$ are $90^\circ, 180^\circ, 270^\circ$. The basis for induction is $n = 4$. In this case the given sum of the angles is either 360° or $1,080^\circ$, which implies that each angle is either 90° or 270° respectively. So f must be a square by itself.

Assume that the claim is true for all values less than n and let $n > 4$. If for some $i, \varphi_f(v_i) = 180^\circ$ then $\lambda(v_{i-1} v_i v_{i+1}) = XX$. We can apply vertex deletion at v_i to obtain $f' = v_1 v_2 \cdots v_{i-1} v_{i+1} \cdots v_n v_1$. Then $\varphi(f') = \varphi(f) - \varphi_f(v_i) = ((n-1) \pm 2) \times 180^\circ$, and by induction we are done.

This leaves the case where all angles are either 90° or 270° . Since $n > 4$ and $\varphi(f) = (n \pm 2) \times 180^\circ$ not all the angles can be equal. Hence there must be a k such that $\varphi_f(v_k) \neq \varphi_f(v_{k+1})$. Hence we have $\lambda(v_{k-1} v_k v_{k+1} v_{k+2}) = XYX$. Apply edge contraction to obtain $f' = v_1 \cdots v_{k-1} v_k v_{k+2} \cdots v_n v_1$. The edge contraction removed 360° from the angle sum and added 180° . Hence $\varphi(f') = ((n-1) \pm 2) \times 180^\circ$. \square

DEFINITION 4.4. A complement of a path P with respect to a square σ is any path P^c in the graph such that PP^c is a cycle which simplifies to σ .

LEMMA 4.6. *Given a path P , all its complements with respect to a square σ , which have the same start and end labels, have a unique simplified form.*

Proof. Let $\lambda(\bar{P}) = \alpha = X_1 X_2 \cdots X_k$. Since α is a spiral we have $X_i = X_j$ for $i = j(4)$. Assume that $k > 4$ and that the spiral α and the square σ are either both clockwise or both counterclockwise. Then σ must be a substring of α . Since σ is a cyclic string we can assume that $\sigma = X_1 X_2 X_3 X_4$.

Let P^c be a complement of P with respect to σ and let $\overline{\lambda(P^c)} = \beta$. Since $k > 4, \beta$ must spiral in the opposite direction to α . Since both α and β are simplified $\alpha\beta$ can be simplified only at the borders between the two strings. Write $\beta = \beta_1 \beta_2 \beta_3$, such that $\overline{\beta_3 \alpha \beta_1} = \alpha$. We are allowed to shift β_3 because $\alpha\beta$ is a cyclic string. Then it is clear that $\beta_1 \in \{X_{k-3} X_k, X_k, \varepsilon\}$ and $\beta_3 \in \{\varepsilon, X_1, X_1 X_4\}$. β_2 is the ‘‘essential part’’ of β . Since $|\alpha| = k$ and $\sigma = 4$, we must have $|\beta_2| = k - 4$. From the possible values of β_1 and β_3 , and the fact that β is a spiral opposite in direction to α , we can conclude that $\beta_2 = X_{k-1} X_{k-2} \cdots X_4$. We use $k > 4$ in order for β_2 not to be an empty string. Therefore

$\beta = \{\varepsilon, X_k, X_{k-3}X_k\}X_{k-1} \cdots X_4\{\varepsilon, X_1, X_1X_4\}$, which is unique but for the start and end labels. The arguments in the cases where α and σ are in opposite directions and for $k \leq 4$ are similar. \square

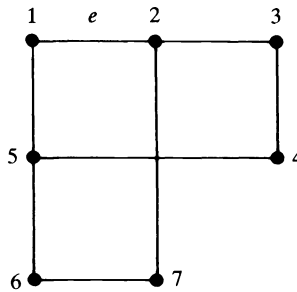
5. Biconnected rectilinear graphs. In this section we discuss an algorithm for recognizing biconnected rectilinear graphs. Note that the ordering relation λ induces a cyclic ordering of the edges incident at each vertex v . For convenience we will need the following definition.

DEFINITION 5.1. Let v be a vertex in a rectilinear graph G . Define $L_G(v)$ to be the cyclic list of the neighbors of v in G in the counterclockwise order.

Using these lists, we can define the essential notion of a candidate face of a biconnected rectilinear graph.

DEFINITION 5.2. Let $G = (V, E, \lambda)$ be a biconnected rectilinear graph. With each edge $e = (v_1, v_2)$, $v_1 > v_2$ ¹ we associate two lists of vertices called *candidate faces* $CF_1(e)$ and $CF_2(e)$ which are defined as follows. $CF_1(e) = v_1, v_2, \dots, v_k, v_{k+1}$ where $v_i \neq v_j$ for $1 \leq i < j \leq k$, and $v_{k+1} = v_i$ for some i , $1 \leq i < k - 1$. Also, for each l , $1 < l < k + 1$, v_{l+1} is the successor of v_{l-1} in the cyclic list $L_G(v_l)$. $CF_2(e)$ is similarly defined but starting with v_2, v_1 .

It is easy to see that CF_1 and CF_2 are uniquely defined. An illustration of this definition is given in Fig. 5.1.



$$L_G(1) = (5, 2) \text{ and } L_G(2) = (1, 7, 3)$$

$$CF_1(e) = 2, 1, 5, 6, 7, 2 \text{ and } CF_2(e) = 1, 2, 7, 6, 5, 4, 3, 2$$

FIG. 5.1. Candidate faces.

We now need a lemma about biconnected undirected graphs. Let us define a biconnected graph to be *minimal* if for every edge e in the graph $G - e$ is not biconnected. The following lemma is taken from [2] and is stated without proof.

LEMMA 5.1. *If G is a minimal biconnected graph having at least four vertices then G contains a vertex of degree two.*

LEMMA 5.2. *In any biconnected graph G which is not a simple cycle, there is a simple path $P = (v_1, v_2), (v_2, v_3), \dots, (v_{r-1}, v_r)$, $r \geq 2$, with the intermediate vertices (if any) v_i , $1 < i < r$ all having degree 2, such that the graph $G' = G - P$ is biconnected.*

Proof. Transform the given graph G to another graph G'' by replacing all paths of the form $P = (v_1, v_2), (v_2, v_3), \dots, (v_{r-1}, v_r)$ where the vertices v_i , $i \neq 1, r$ all have degree 2, by the edge (v_1, v_r) . So for each edge e in G'' we have a corresponding path P_e in G . Note that the degree of any vertex in G'' is at least three. If G'' has multiple edges between some two vertices, say u and w , then in G there must be at least two parallel paths between u and w . Since G is not a simple cycle any one of those paths

¹ For convenience we assume that V is a set of integers.

will serve our purpose. If G'' does not have multiple edges then it must have at least 4 vertices. By Lemma 5.1 G'' cannot be minimal. Therefore there is an edge e in G'' such that $G'' - e$ is biconnected, which implies that $G - P_e$ is also biconnected. \square

The following theorem gives a necessary and sufficient condition for a biconnected rectilinear graph to be embeddable.

THEOREM 5.1. *Let $G = (V, E, \lambda)$ be a biconnected rectilinear graph with at least three edges. Then G is embeddable if and only if for each edge e in the graph both the candidate faces $CF_1(e)$ and $CF_2(e)$ represent simple embeddable cycles in the graph (i.e. the starting and ending vertices are identical, and it simplifies to a square). Moreover, if the graph is embeddable each such distinct candidate face corresponds to a face in the planar embedding.*

Proof. Only if. Supposing for some edge $e = (v_1, v_2)$, $CF_1(e)$ is not a cycle, i.e. $CF_1(e) = v_1, v_2, \dots, v_k, v_{k+1}$ with $v_i = v_{k+1}$ for some $i, 1 < i < k - 1$. Suppose G is embeddable. Look at the cycle v_i, \dots, v_k, v_{k+1} in the embedding. Suppose that the edge (v_{i-1}, v_i) is inside this cycle. There can be no other edges $(u, v_j), i < j \leq k$ inside this cycle, otherwise u would have appeared instead of v_{j+1} in $CF_1(e)$. From this observation and the fact that the embedding is planar, it follows that v_i is an articulation vertex, which contradicts the biconnectedness of G . The case where (v_{i-1}, v_i) is outside the cycle is similar (both cases are depicted in Fig. 5.2).

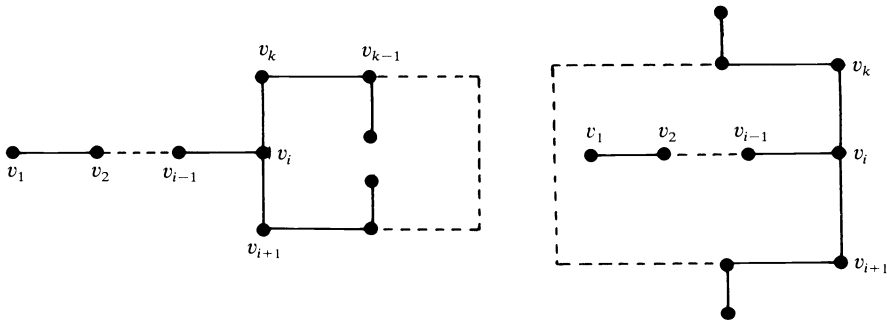


FIG. 5.2. Two possible embeddings of $CF_1(e)$.

Suppose $CF_1(e)$ is a cycle but is not embeddable. Since $CF_1(e)$ is a subgraph of G , G itself cannot be embeddable. Similar arguments hold for $CF_2(e)$.

If. The proof of this part is by induction on the number of edges. The basis for the induction are simple embeddable cycles. Assume that the claim is true for any biconnected rectilinear graph which has less than k edges. Let G be a biconnected rectilinear graph which is not a simple cycle and which has k edges. By Lemma 5.2, there is a simple path $P = (v_1, v_2), (v_2, v_3), \dots, (v_{r-1}, v_r)$ with the vertices $v_i, i \neq 1, r$ all having degree 2, such that the graph $G' = G - P$ is biconnected. v_1 and v_r will have degree greater than two. Also assume that $v_1 > v_2$ and $e_{12} = (v_1, v_2)$.

Since all our candidate faces are cycles, if an edge e lies on a candidate face f then either $CF_1(e) = f$ or $CF_2(e) = f$. So each edge will be present in exactly two of these candidate faces. Hence the path P will appear in $CF_1(e_{12})$ and its reverse path will appear in $CF_2(e_{12})$. Let

$$f_1 = CF_1(e_{12}) = v_1, v_2, \dots, v_r, u_1, \dots, u_j, v_1,$$

$$f_2 = CF_2(e_{12}) = v_r, v_{r-1}, \dots, v_1, w_1, \dots, w_k, v_r, \text{ and}$$

$$f_s = v_1, w_1, \dots, w_k, v_r, u_1, \dots, u_j, v_1.$$

It follows from the definition of the candidate faces f_1 and f_2 that the vertices u_j, v_2, w_1 appear consecutively in that order in $L_G(v_1)$ and that w_k, v_{r-1}, u_1 appear similarly in $L_G(v_r)$ (see Fig. 5.3). Therefore for each edge in f_1 or f_2 which is not in P , the new candidate face in G' will be f_3 which is a simple cycle.

We still have to show that f_3 is embeddable. Since f_1 and f_2 are both embeddable $\varphi(f_1) = (r + j \pm 2) \times 180^\circ$ and $\varphi(f_2) = (r + k \pm 2) \times 180^\circ$. However, since f_1 and f_2 share the edge e_{12} it is implied by the definition of candidate faces that $\varphi(f_1) = (r + j + 2) \times 180^\circ$ and $\varphi(f_2) = (r + k + 2) \times 180^\circ$ is impossible. With a little bit of algebraic manipulation we can show that $\varphi(f_3) = ((j + k + 2) \pm 2) \times 180^\circ$. Since f_3 has $j + k + 2$ vertices by Lemma 4.5, it is embeddable. Thus the candidate faces for G' are the same as those for G , excepting for f_3 replacing the two faces f_1 and f_2 . So for each edge in G' its two candidate faces are again simple embeddable cycles. By the induction hypothesis G' is embeddable and each distinct candidate face corresponds to a face in its embedding. The orderings of the edges at the vertices v_1 and v_r imply that the end edges (v_1, v_2) and (v_{r-1}, v_r) of the path P are both trying to go inside the face corresponding to f_3 .

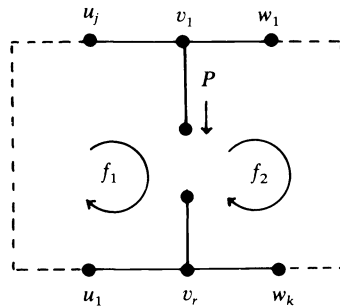


FIG. 5.3. The two cycles f_1, f_2 and the path P .

We are left to show that we can add the path P back without destroying embeddability. Find any rectilinear path P' in the face corresponding to f_3 in the embedding of G' , that starts and ends with $\lambda((v_1, v_2))$ and $\lambda((v_{r-1}, v_r))$ respectively. This is clearly possible although we may have to extend the grid in order for P' to lie on the grid lines. P' creates a face in the embedding with the path $P_1 = v_r u_1 u_2 \cdots u_j v_1$. If f_3 is not the outside face then $\lambda(P_1 P') = \lambda(f_1) = \lambda(P_1 P) = \sigma$. The case when f_3 is the outside face is slightly more complicated. There are two such different paths P' depending on the new outer face that is created. However, for one of the two the above holds and suppose this is the one we chose. By definition both P and P' are complements of P_1 with respect to σ , they also share the same start and end labels, and by lemma 4.6 we have $\lambda(P) = \lambda(P')$. Therefore $G' + P'$ can be obtained from G by applying a sequence of the four topological operations, and since $G' + P'$ is embeddable, by Lemma 4.1 G is also embeddable. It is easy to see that the two new faces we get after inserting P in the embedding of G' correspond to f_1 and f_2 . \square

The above theorem leads to the following algorithm for recognizing embeddable biconnected rectilinear graphs. The algorithm also outputs the faces of the embedding if the graph happens to be embeddable.

Algorithm *check-biconnected*(G);
begin
if G is an edge **then return**;

```

if  $|E| > 3|V| - 6$  then
  begin
    write (“not embeddable”);
    quit
  end;
for each edge  $e$  do
  begin
    mark  $[e, 1] := \text{false}$ ;
    mark  $[e, 2] := \text{false}$ 
  end;
for each edge  $e$  do
  for  $i := 1$  to 2 do
  begin
    if not mark  $[e, i]$  then
      begin
         $f := \text{candidate-face}(e, i)$ ;
        if not embed-cycle ( $f$ ) then
          begin
            write (“not embeddable”);
            quit
          end;
        for each edge  $e' = (v_1, v_2)$  in  $f$  do
          if  $v_1 > v_2$  then mark  $[e', 1] := \text{true}$ 
          else mark  $[e', 2] := \text{true}$ ;
        output ( $f$ )
      end
    end
  end
end.

```

Boolean function *embed-cycle* (f) returns value *true* if f is an embeddable cycle. If f is a cycle then we simplify using the reduction rules and check if we end up with a square. This can be done in time linear in the size of f . Function call *candidate-face* (e, i) returns the candidate face $CF_i(e)$ and the function can be implemented exactly as described in Definition 5.2. In the calls to this function, each edge e can be traversed at most twice, due to the flags mark $[e, 1]$ and mark $[e, 2]$. Therefore the algorithm runs in time $O(|V|)$. We conclude this section with a lemma which will let us identify the outer face in a rectilinear graph.

LEMMA 5.3. *Let G be an embeddable biconnected rectilinear graph. For all interior faces f in the embedding of G , $\varphi(f) = (n-2) \times 180^\circ$, and for the unique exterior face f_e , $\varphi(f_e) = (n+2) \times 180^\circ$.*

Proof. Consider the embedding of G . The faces of the embedding are determined by G , and are simple polygons in the plane. By the definition of φ , for every interior we count the interior angles, and for the exterior face we count the exterior angles. The lemma follows. (Remember that if G is a simple cycle, the embedding has two faces). \square

LEMMA 5.4. *Let G be an embeddable biconnected rectilinear graph, f_e the exterior face in its embedding and v a vertex on f_e . If $\varphi_{f_e}(v) = 180^\circ$, then G can be embedded inside a polygon of shape **U**, as shown in Fig. 5.4a. If $\varphi_{f_e} = 270^\circ$, then G can be embedded inside a polygon of shape **W**, as shown in Fig. 5.4b.*

Proof. The proof is easy and left to the reader. \square

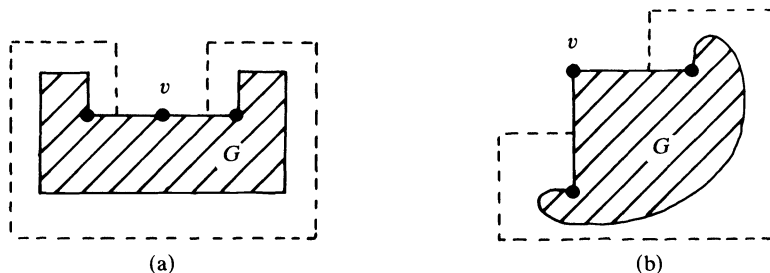


FIG. 5.4. Shapes U and W.

6. Articulation vertices. In this section we examine the conditions under which the embeddability of the biconnected components of the graph imply the embeddability of the graph itself. Clearly, this will depend on the way components meet at articulation vertices. In Fig. 3.2, we showed two examples of nonembeddable rectilinear graphs, each of which decomposes into two embeddable biconnected rectilinear graphs.

In those cases, the two biconnected components are not “compatible” at the articulation vertex. However, the situation need not be so local. Fig. 6.1 depicts two nonembeddable graphs, each of which decomposes into three embeddable biconnected components, so that the components meeting at each articulation vertex are compatible. Note that an edge is a (trivial) biconnected component.

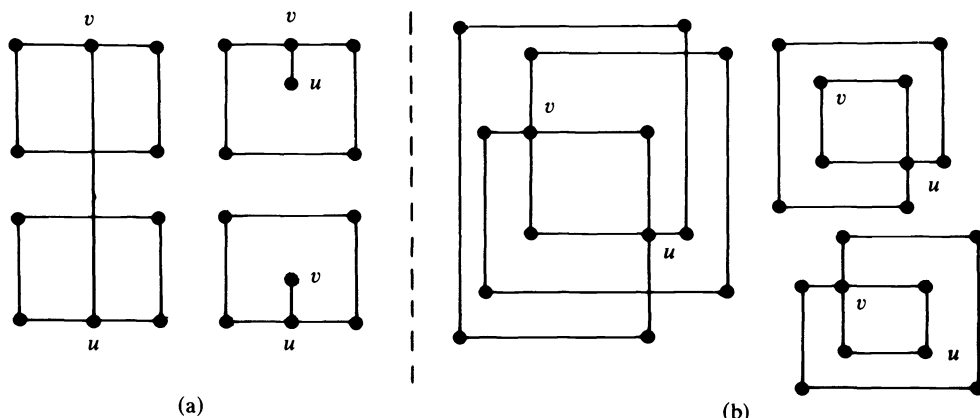


FIG. 6.1. Decompositions of nonembeddable graphs.

If v is an articulation vertex in a graph G , then its removal results in several connected subgraphs G_i of G . We will refer to the subgraphs $G_i + v$, as the subgraphs *meeting* at v . Throughout this section we will implicitly assume that we are dealing with rectilinear graphs whose biconnected components are embeddable.

DEFINITION 6.1. Let B_1 and B_2 be two nontrivial biconnected components of a rectilinear graph G that share an articulation vertex v . Then B_1 and B_2 are said to *interlace* if the horizontal edges at v belong to B_1 and the vertical edges belong to B_2 (Fig. 3.2a). We also say that v is an *interlace vertex*. Any articulation vertex that does not have this property is said to be *interlace-free*.

LEMMA 6.1. A rectilinear graph G which has an interlace articulation vertex v is not embeddable.

Proof. Let B_1 and B_2 be the two biconnected components sharing the vertex v . Since B_1 and B_2 are nontrivial, the horizontal edges at v lie on a cycle in B_1 and the

vertical edges lie on a cycle in B_2 . It is impossible to draw G on the plane without these two cycles crossing. \square

DEFINITION 6.2. Let B_1 and B_2 be two noninterlacing biconnected components of G that share an articulation vertex v , and assume B_1 is nontrivial. Then B_1 is said to *dominate* B_2 at v (or, B_2 is *inside* B_1) if either (i) v is not on the exterior face of B_1 , or (ii) edges (v, u) and (v, w) at the vertex v are on the exterior face of B_1 , and u, w are consecutive in that order in $L_G(v)$ (note that they are always consecutive in $L_{B_1}(v)$). If neither B_1 dominates B_2 nor B_2 dominates B_1 , then B_1 and B_2 are said to be *outside* each other.

The intuition behind the above definition is that in the embedding, one biconnected component must lie wholly inside some face of the other if one edge of it does. This is due to the planarity criterion. Clearly, if biconnected components B_1 and B_2 that share an articulation vertex v dominate each other, the graph is not embeddable (this is the case in Fig. 3.2b).

Let B_1 and B_2 be two biconnected components of a graph G that share an articulation vertex v , such that B_1 dominates B_2 . Let G' be the subgraph of G meeting at v that contains B_2 . If G is embeddable then in any embedding of G , all of G' should lie inside one face of B_1 . This suggests extending the relation “dominate” as follows:

DEFINITION 6.3. Let $\mathbf{B} = \{B_1, B_2, \dots, B_m\}$ be the set of biconnected components of G . We say that B_i *dominates* B_j if there exists a biconnected component B_k and an articulation vertex v , such that (i) B_k and B_i share v , (ii) B_i dominates B_k at v , and (iii) B_j and B_k are both subgraphs of one of the connected subgraphs meeting at v .

Let us denote by $V(G)$ the vertex set of the graph G and by $E(G)$ the edge set.

LEMMA 6.2. *If in a rectilinear graph G , there exists some pair of biconnected components B_1 and B_2 that dominate each other, then G is not embeddable.*

Proof. If B_1 and B_2 share an articulation vertex v , then as mentioned earlier G is not embeddable. Suppose that B_1 and B_2 are disjoint. Since B_1 and B_2 dominate each other, there must be articulation vertices v_1, v_2 , biconnected components B'_1, B'_2 , and subgraphs G_1, G_2 , such that for $i = 1, 2$, (i) B_i and B'_i share v_i , (ii) B_i dominates B'_i at v_i , and (iii) G_i is one of the subgraphs meeting at v_i and contains B'_i . Let us assume that G is embeddable. From (i) $v_2 \in V(G_1)$, (ii) G_1 lies wholly inside B_1 in the embedding, and (iii) $V(G_1) \cap V(B_1) = \{v_1\}$, we can conclude that v_2 must be properly inside a polygon defined by the face f_1 of B_1 containing v_1 . Similarly v_1 should be properly inside the polygon defined by a face f_2 of B_2 containing v_2 . Therefore some vertices of f_2 must lie outside f_1 and the two faces must intersect, and hence G is not embeddable. \square

Given a rectilinear graph G , with a set of biconnected components \mathbf{B} and a set of articulation vertices \mathbf{A} , we can construct a tree T of biconnected components such that

$$V(T) = \mathbf{A} \cup \mathbf{B}, \quad \text{and} \quad E(T) = \{(v, B) \mid v \in \mathbf{A}, B \in \mathbf{B}, v \in V(B)\}.$$

LEMMA 6.3. *Let G be a rectilinear graph with the set of biconnected components \mathbf{B} and tree of biconnected components T . Let B be a leaf in the tree T which is adjacent to an articulation vertex v of degree 2 in T . If B dominates B' the other biconnected component adjacent to v in T , then B dominates every other biconnected component in \mathbf{B} .*

Proof. The only two subgraphs meeting at v are B and $G - B + v$ and the proof follows from Definition 6.3. \square

If no two biconnected components dominate each other, then the relation “dominate” induces a partial order on \mathbf{B} . A nondominating element in this partial order is a biconnected component which does not dominate any biconnected component.

COROLLARY 6.1. *If for a rectilinear graph G “dominate” is a partial order, then there exists a nondominating biconnected component which is a leaf in the tree T of biconnected components.*

Proof. Any trivial biconnected component (which is just an edge) must be non-dominating. If any vertex in T (corresponding to an articulation vertex in G) is adjacent to two leaves, then either the two leaves are nontrivial and not dominating, or one of them is a trivial biconnected component. If no vertex in T is adjacent to two leaves, then all leaves are adjacent to vertices of degree 2, and there are at least two such leaves. If two of these leaves are dominating, then by Lemma 6.3 the two leaves dominate each other which is a contradiction that “dominate” is a partial order. In fact all of these leaves must be nondominating. \square

THEOREM 6.1. *Let G be a rectilinear graph and \mathbf{B} its set of biconnected components. G is embeddable if and only if*

- (i) *every biconnected component B in \mathbf{B} is embeddable,*
- (ii) *every articulation vertex in G is interlace-free, and*
- (iii) *“dominate” induces a partial order on \mathbf{B} .*

Proof. The necessary part follows from Lemma 6.1 and Lemma 6.2.

The sufficient part is shown by induction on the number of vertices. The basis for induction is any biconnected rectilinear graph. Let G be not biconnected with $|V(G)| = n$. Assume that the claim is true for all smaller graphs. Look at the tree T of biconnected components. By Corollary 6.1, there exists a leaf B in T which is nondominating. Let v be the articulation vertex shared by B and $G' = G - B + v$, the rest of the graph. G' being a subgraph of G also satisfies the conditions of the claim. By induction hypothesis G' is embeddable. By condition (i), B is also embeddable. If B is a single edge it is easy to add the edge to the embedding of G' . Assume B is nontrivial. Since B is nondominating, v must lie on the exterior face f_e of B and $\varphi_f(v) \neq 90^\circ$ (why?).

Embed G' and B separately and consider the vertex v in both embeddings. If $\varphi_f(v) = 180^\circ$, then v is only one edge in G' . Add six new grid lines to the embedding of G' , create the shape \mathbf{U} as shown in Fig. 6.2a, magnify the embedding, and embed B in the \mathbf{U} as in Lemma 5.4. If $\varphi_f(v) = 270^\circ$, then v is either on just one edge in G' , or on two perpendicular edges in G' . In both cases, add six new grid lines, create the shape \mathbf{W} and embed B as shown in Fig. 6.2b. \square

Before we describe an algorithm for testing embeddability, we need an algorithm for testing whether “dominate” is a partial order on the set of biconnected components. From the tree T of biconnected components, we construct \bar{T} a partially directed tree as follows. Assume that no biconnected component dominates and is dominated at the same vertex. If so then “dominate” is not a partial order. Direct edge (v, B) from

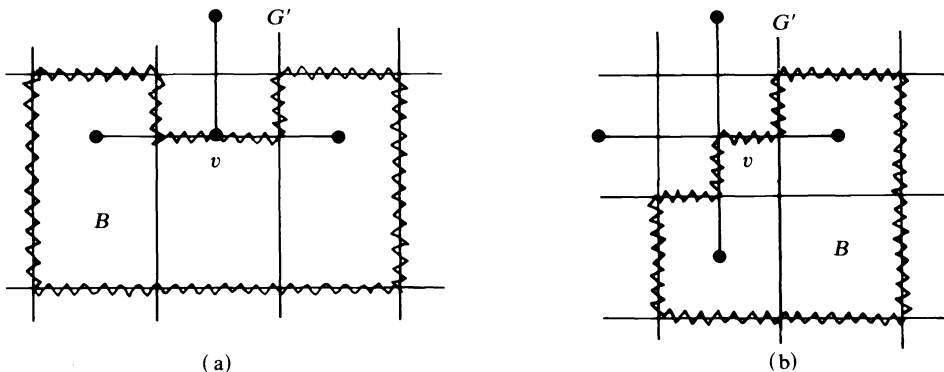


FIG. 6.2. Adding B to the embedding of G' .

B to v if B dominates at v . Direct edge (v, B) from v to B if B is dominated at v . Leave all the other edges undirected.

This partially directed tree \bar{T} can be constructed in linear time as follows. Find the faces of each of the biconnected components using the algorithm *check-biconnected*. This takes $O(|V|)$ time. Check for dominations at each articulation vertex as described in Definition 6.2. There are at most 4 biconnected components at each articulation vertex and hence there are at most 12 (ordered) pairs to be tested for domination (in fact only 2 tests are necessary, how?). Construct \bar{T} by directing the edges of T as described earlier. Note that articulation vertices and biconnected components can be found in $O(|V|)$ [1]. For each vertex x in \bar{T} , denote by $d_{in}(x)$, $d_{out}(x)$, and $d(x)$, the number of incoming arcs, the number of outgoing arcs, and the number of undirected edges of x respectively. The rest of the algorithm is given below.

```

Algorithm check-dominate-po ( $G$ );
begin
  construct  $\bar{T}$ ;
  for each vertex  $x$  in  $\bar{T}$  do
    if  $d_{in}(x) > 1$  then
      begin
        write (“not a partial order”);
        quit
      end;
    if search ( $\bar{T}$ ) then write (“yes, partial order”)
    else write (“not a partial order”)
  end;

function search ( $\bar{T}$ ): boolean;
begin
  if  $T = \Phi$  then search := true
  else begin
    if  $\exists B \in \mathbf{B}$  with  $d_{out}(B) = 0$ ,  $d_{in}(B) + d(B) = 1$  then
      begin
        Let  $v$  be the neighbor of  $B$ ;
        if  $d_{in}(v) + d_{out}(v) + d(v) = 1$  then search := search ( $\bar{T} - \{B\}$ )
        else search := search ( $\bar{T} - \{B, v\}$ )
      end else search := false
    end
  end.

```

The above algorithm can be easily shown to be correct using Definition 6.3 and Corollary 6.1. The boolean function *search* can be implemented nonrecursively to run in linear time by maintaining a queue of the leaves of \bar{T} .

Given the biconnected components and articulation vertices, checking that the articulation vertices are interlace-free can be done in $O(|V|)$ time. Let *check-interlace-free* be a procedure that checks a given articulation vertex for interlace-freedom. We end this section with a $O(|V|)$ algorithm for testing embeddability of rectilinear graphs.

```

Algorithm check-rectilinear ( $G$ );
begin
  Decompose  $G$  into its biconnected components;
  for each biconnected component  $B$  do check-biconnected ( $B$ );
  for each articulation vertex  $v$  do check-interlace-free ( $v$ );
  check-dominate-po ( $G$ )
end.

```

7. An embedding algorithm. In the previous section we gave an algorithm for testing embeddability. This algorithm can be easily modified into an algorithm which gives an embedding. However, the complexity of this naive algorithm would be $O(|V|^3)$. The reasoning is as follows. The path P' that we find in the proof of Theorem 5.1 could be $O(|V|)$ long. For each topological operation that we apply on this path to transform it to the path P , we update the coordinates of the vertices in the embedding once. Thus for each path added we require $O(|V|^2)$ time. There can be $O(|V|)$ such paths and hence the complexity of the algorithm is $O(|V|^3)$. To reduce the complexity to $O(|V|^2)$, we have to make sure that the path P' is never longer (asymptotically) than the path P . In this case the sum of the lengths of all such paths P' is $O(|V|)$, and the $O(|V|^2)$ complexity follows. In the following, we show how we can always find such paths, describe the algorithm, and analyze its complexity.

LEMMA 7.1. *Let G be a planar biconnected multigraph with minimum degree three. Then any embedding of G has an interior face of size at most five.*

Proof. The dual G^d of G is also a planar graph. Since G has minimum degree 3, G^d is a simple graph. Hence G^d has at least two vertices of degree ≤ 5 [2]. G is biconnected and hence one of the vertices must correspond to a face whose size is less than or equal to 5. \square

LEMMA 7.2. *Given an embedding of a planar biconnected graph G , which is not a cycle, there is a simple path P , such that (i) the interior vertices of P all have degree 2, (ii) the end vertices of P have degree ≥ 2 , (iii) P appears in an interior face f in the planar embedding, and (iv) $5 \cdot |P| \geq |f|$.*

Proof. As in the proof of Lemma 5.2, transform G to G' by replacing all paths with property (i) and (ii) by edges. By Lemma 7.1, G' has an interior face f of size at most 5. The longest of all the paths in G corresponding to the edges of f will satisfy conditions (iii) and (iv). \square

To get an embedding of a given rectilinear graph, we first test if the graph is embeddable and then apply the following algorithm.

Algorithm embed-rectilinear (G);

begin

for each biconnected component B **do** embed-biconnected (B);
 join-the-embeddings;

end.

Algorithm embed-biconnected (B);

begin

 get-long-path (P, P_1, σ);
 embed-rectilinear ($B - P$);
 find-path-in-embedding (P', P_1, σ);
 apply-operations-and-transform (P', P)

end.

Procedure *get-long-path* returns paths P, P_1 , and square σ , such that P satisfies the conditions of Lemma 7.2, and the interior face $f = PP_1$ simplifies to σ . By Lemma 7.2 such a path exists.

Procedure *find-path-in-embedding* traces a path P' in the embedding of $B - P$, such that P' starts and ends in the same directions as P , and $P'P_1$ simplifies to σ . P' and P are both complements of P_1 with respect to the square σ . Since PP_1 is an interior face, P' can be obtained by starting in the required direction, then following the path P_1 in the embedding of $B - P$, and ending in the required direction (Fig. 7.1). This will result in P' being a complement of P_1 with respect to σ . We have $|P'| = O(|P_1|) = O(|P|)$.

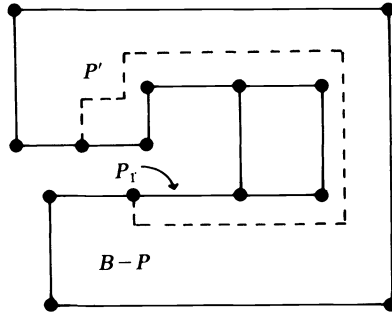


FIG. 7.1. Finding the path P' in the embedding of $B - P$.

Procedure *apply-operations-and-transform* applies a sequence of the four topological operations to P' in the embedding of $B - P + P'$ and transforms it to P thus resulting in a embedding of B . This is done by first simplifying the path P' and then expanding the simplified path to get the path P (Fig. 7.2). The number of operations applied will be $O(|P| + |P'|) = O(|P|)$.

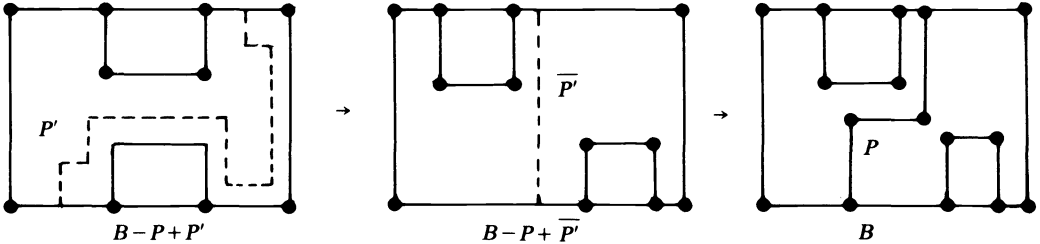


FIG. 7.2. Path addition, simplification and expansion.

Procedure *join-embeddings* takes the embeddings of the biconnected components and puts them together to get an embedding for G . This is done essentially following the proof of Theorem 6.1. Find a nondominating component B . Recursively embed $G' = G - B$. Join the embeddings of B and G' using the shapes **U** or **W** as shown in Fig. 6.2.

The algorithm can be shown to be correct using the material developed in the previous three sections. We now analyze the complexity of each step in the algorithm and show that the total complexity is $O(|V|^2)$.

Procedure *join-embeddings* updates each coordinate at most once per recursive call. The total number of calls is bounded by the number of biconnected components. Hence this procedure takes $O(|V|^2)$ time.

Procedure *get-long-path* can be implemented to run in $O(|V|)$ time each time it is called. Remember that we can get the faces of a biconnected graph from the testing algorithm, and searching all faces to get the required face takes linear time. Procedure *find-path-in-embedding* takes $O(|P_1|) = O(|V|)$ time. These two procedures will be invoked at most $O(|V|)$ time. Hence total time spent in these calls is $O(|V|^2)$.

Procedure *apply-operations-and-transform* applies a sequence of $O(|P|)$ operations. Each edge in G will appear in only one such path P . Hence the sum of the lengths of all such paths P is $O(|V|)$. Each operation updates at most $O(|V|)$ coordinates. Therefore the time spent in calls to this procedure is $O(|V|^2)$.

8. Consistent rectilinear graphs. Certain rectilinear graphs cannot be drawn on the grid even if we relax the planarity criterion. We say that a rectilinear graph $G(V, E, \lambda)$ is *consistent* if it can be drawn on the grid satisfying the ordering relation λ . In other words, G is consistent if the set of equality and inequality constraints generated in part 1 of Definition 2.2 is consistent.

The equality constraints define an equivalence relation on the set of coordinates of the vertices of G . Let us denote by $e(x)$ the equivalence class containing the coordinate x . Denote by I_x and I_y the sets of x -coordinate and y -coordinate inequality constraints respectively. Construct two directed graphs $G_x(V_x, E_x)$ and $G_y(V_y, E_y)$ as follows:

$$V_x = \{e(x) | x = x(a), a \in V\} \quad \text{and} \quad E_x = \{(x_1, x_2) | x_1 > x_2 \in I_x\}.$$

V_y and E_y are similarly defined.

It can be easily shown that G is consistent if and only if the two directed graphs G_x and G_y are both acyclic. A solution to the coordinates, which satisfies the constraints will correspond to a (possibly) nonplanar embedding of G on the grid. This can be obtained by performing the topological sort operation [5] on the two acyclic digraphs. In fact this will yield a solution that minimizes the area of the rectangle bounding the embedding.

In a nonplanar embedding of a consistent rectilinear graph on the grid, all crossings are between horizontal edges and vertical edges. The vertical edges can be assigned one layer, and the horizontal edges can be assigned a second layer. In other words the “thickness” [2] of a consistent rectilinear graph is less than or equal to two.

9. Extensions, open problems, and conclusions.

1. It can be easily shown that the area of the embedding given by the algorithm in this paper can be made $O(|V|^2)$ without extra time penalty. There are graphs that require this much area. To minimize the area is NP-complete if the input graph is allowed to be disconnected. The minimization problem is open for connected rectilinear graphs.

2. The embedding problem of appropriately defined graphs for other grids (triangular, hexagonal, etc.), seems to be interesting in light of certain systolic layouts for VLSI [4] that use them. It originally seemed to us that the ideas of this paper will carry through without much change to other grids. They do not. “Triangular” graphs, for example, may have triangles which must be equilateral in any embedding. This rigidity (which does not appear in the rectilinear case), makes some of our results false for these graphs.

3. If we allow two layers for the embedding (each of which must be planar), then assigning horizontal and vertical edges to different layers easily solves the problem. However, in reality the user decides which wire will be on which layer. The results in this paper give only a necessary condition for the embeddability of such a multilayered rectilinear graph. Under what conditions can we obtain compatible embeddings on the different layers (i.e. embeddings that have corresponding vertices at the same grid points)?

4. An interesting class of graphs that contains all rectilinear graphs is the class of graphs in which the edges incident on each vertex are cyclically ordered (now there is no degree or direction constraints). The corresponding problem is whether a graph in this class can be laid on the plane consistent with the cyclic orderings of the edges at each vertex, so that no edges cross. This problem can be solved in linear time [11].

We conclude by observing that even linear time and space algorithms may not be considered efficient for VLSI applications, due to the huge size of the graphs involved.

However, if the layout is given by a “good” hierarchical description, then both time and space complexity of our algorithms can be reduced considerably. ALI allows hierarchical description of layouts through its *cell* mechanism [8], and our algorithms will be implemented in ALI.

10. Acknowledgments. This problem was originally raised by Professors Bob Sedgewick and Dick Lipton. We wish to thank Professors Dick Lipton and Jacobo Valdes for several useful discussions. We are grateful to Professor George Lueker for pointing out an omission in an earlier version of this paper. Our thanks are also due to Edna Wigderson, Vijaya Ramachandran and the referee for their comments.

REFERENCES

- [1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
- [2] M. BEHZAD, G. CHARTRAND AND L. LESNIAK-FOSTER, *Graphs and Digraphs*, Wadsworth International Group, London, 1981.
- [3] J. E. HOPCROFT AND R. E. TARJAN, *Efficient planarity testing*, J. Assoc. Comput. Mach., 21 (1974), pp. 549-568.
- [4] H. T. KUNG AND C. E. LEISERSON, *Systolic arrays (for VLSI)* in Sparse Matrix Proceedings, I. S. Duff and G. W. Stewart eds., Society for Industrial and Applied Mathematics, Philadelphia, 1979.
- [5] D. E. KNUTH, *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1971.
- [6] C. E. LEISERSON, *Area efficient graph embeddings (for VLSI)*, Proc. 21st Symposium on the Foundations of Computer Science, October 1980.
- [7] R. J. LIPTON, S. C. NORTH, R. SEDGEWICK, J. VALDES AND G. VIJAYAN, *VLSI layout as programming*, ACM Trans. Programming Languages and Systems, 5 (1983), pp. 405-421.
- [8] ———, *ALI: a procedural language to describe VLSI layouts*, Proc. of the 19th Design Automation Conference, Las Vegas, June 1982.
- [9] R. SETHI, *Testing for the Church-Rosser property*, J. Assoc. Comput. Mach., 21 (1974), pp. 671-679; *errata*, 22 (1975), p. 424.
- [10] G. VIJAYAN, *Completeness of VLSI layouts*, VLSI Memo #1, Dept. of Electrical Engineering and Computer Science, Princeton Univ., Princeton, NJ, September 1982.
- [11] G. VIJAYAN AND A. WIGDERSON, *Planarity of edge ordered graphs*, Technical Report #307, Dept. Electrical Engineering and Computer Science, Princeton Univ., Princeton, NJ, December 1982.
- [12] A. WIGDERSON, *The complexity of the Hamiltonian circuit problem for maximal planar graphs*, Technical Report #298, Dept. Electrical Engineering and Computer Science, Princeton Univ., Princeton, NJ, February 1982.

AXIOMS FOR THE THEORY OF LAMBDA-CONVERSION*

GYORGY REVESZ†

Abstract. In the standard presentations of λ -calculus (e.g., in [H. Barendregt, *The Lambda Calculus, Its Syntax and Semantics*, North-Holland, Amsterdam, 1981] or [J. R. Hindley, B. Lecher, J. P. Seldin, *Introduction to Combinatory Logic*, Cambridge Univ. Press, London, 1972]) the operation of substitution is defined as a primitive operation and used in the definition of convertibility. In the present paper we show that the axioms for the theory of lambda-conversion can be simplified in such a way that substitution is not needed at all, as it is reduced to a more elementary operation of replacement without giving up the intuitive simplicity of the lambda-notation. This is achieved by making essential use of the properties of substitution in formulating the axiom system. Also, another unusual axiom system will be presented which uses renaming that replaces every (free or bound) occurrence of a variable by another. Finally, we give the outline of a program written in PL/I that computes the normal form (if any) of λ -terms by using our axioms.

Key words. lambda-calculus, combinatory logic

1. Introduction. Interest in lambda-calculus has been growing rapidly in recent years especially among computer scientists. Its significance with respect to the semantics of programming languages has been widely recognized. The appeal of lambda-calculus lies in its ability to handle the problem of substitution in the most general setting.

Substitution has always been a stumbling-block with mathematical logic and with programming languages, as well. Even in the simplest logical systems it is a problem how to define the substitution operation exactly. The problem becomes far more complex if we have to deal with different types of variables, and if some of the variables may be bound, as is the case with most programming languages. In the introduction of the book *Combinatory Logic* (see [6, pp. 3-4]), the authors express their view this way: "The extent of the complications in such cases may be seen from the fact that most formulations of the rule for substitution for a functional variable in the first-order predicate calculus which were published before 1940, were demonstrably incorrect; and there is little doubt that one of the first correct formulations, that were given by Church {IML}, p. 57, was derived by the aid of the theory of lambda-conversion, the form of combinatory logic which is his specialty." (See [5].)

Various attempts have been made at simplifying the operation of substitution. The most radical approach is represented by the theory of combinators, developed mainly by Curry, where bound variables are eliminated altogether using constants, called combinators, to describe the properties of substitution. But, for this technical advantage we have to sacrifice the intuitive simplicity of the lambda-notation. Actually, the idea of eliminating all variables from mathematical logic goes back to Schönfinkel [11]. Unfortunately, all attempts to provide a foundation for mathematics via lambda-calculus or combinatory logic failed. Tarski and others have shown, however, that substitution can be eliminated from the predicate logic by making essential use of the properties of identity in formulating the axiom system. (See [8], [9], and [13].)

In the present paper we use a similar approach to the pure lambda-calculus. Namely, we show that the axioms for the theory of lambda-conversion can be formulated in such a way that substitution is not mentioned. This is achieved by making essential use of the properties of substitution in formulating the axiom system. At the same time, in contrast with the theory of combinators, we keep the intuitive simplicity of the lambda-notation. Our axiom system corresponds, in fact, to the decomposition

* Received by the editors November 9, 1981, and in final revised form November 28, 1983.

† Computer Science Department, Tulane University, New Orleans, Louisiana 70118.

of the substitution operation into a sequence of more elementary steps. Thus, we do not eliminate the bound variables. Instead, we generalize the notion of reduction to cover incomplete substitution, as well.

Hence, our approach differs significantly from that of de Bruijn [3], [4], or Berkling and Fehr [2], or Staples [12]. De Bruijn developed a notational system, where occurrences of variables are indicated by integers giving the “distance” to the binding λ instead of a name attached to that λ . (Different occurrences of the same variable will thus often be represented by different integers and conversely, different occurrences of the same integer may represent different variables.) This means that the elimination of the variables is made at the cost of readability by humans. This drawback is acknowledged in the introduction of [3], but the system is claimed to be easy for metalingual discussion and for computer programming.

Berkling used a similar approach by introducing a new unbinding operator, denoted by $\#$, which neutralizes the effect of one preceding lambda binding. The n -fold iteration of $\#$, denoted by $\#^n$, would play the same role as the integers of de Bruijn. Namely, an occurrence of $\#^n x$ for some variable x is obviously bound by the $n + 1$ st preceding λ , if any. (Here precedence is used, of course, in the structural sense.) The complete equivalence to the system developed by de Bruijn is recognized in the introduction of [2]. The definition of the β -reduction is rather complicated in both systems, because of the difficulty of updating those numerals everywhere inside the affected part of the lambda expression being reduced.

A somewhat complementary approach is taken by Staples in [12]. He eliminates only the bound variables by incorporating a list of the relative positions of the occurrences of the bound variable into the binding λ operator. Thus, he needs only a placemaker (dummy) at the occurrences of the bound variables since he is using forward references to their positions, whereas in de Bruijn’s or Berkling’s representation the occurrence of a bound variable is indicated by an integer which refers back to the corresponding λ operator. The representation developed by Staples requires also a fairly complicated updating of those lists when making β -reduction. The system is not particularly suitable for computer implementation nor is it simple for humans.

It is clear that the complexity of the substitution operation lies behind the efforts to eliminate bound variables. Curry has solved this problem with his combinators. The above mentioned other approaches do not seem to have made substitution and β -reduction (which should be taken together for making comparisons) any easier than with the original lambda calculus. Such technical matters are of little concern for the semantical (i.e., model theoretical) investigations but they are of primary importance for computer implementations as shown, for instance, by Turner in [14].

We shall see below that technical simplicity can be achieved without giving up the intuitive appeal of the standard lambda-notation. Section 2 is devoted to elimination of substitution from the definition of β -reduction. The corresponding axiom system was presented first in [10]. In § 3 we will introduce another axiom system which makes use of an absolutely unorthodox way of substitution that replaces every occurrence (free or bound) of a variable by another. Such a brute force substitution, called renaming, appears to have certain advantages for implementing β -reduction. (It may be viewed as the automatic though partial enforcement of the variable convention as specified in [1, p. 26].) Section 4 briefly describes our computer program which computes the normal form (if any) of lambda expressions.

2. The elimination of substitution. We define the set of λ -terms as the formal language generated by the context-free grammar:

$$\langle \lambda\text{-term} \rangle ::= \langle \text{variable} \rangle | \lambda \langle \text{variable} \rangle . \langle \lambda\text{-term} \rangle | (\langle \lambda\text{-term} \rangle) \langle \lambda\text{-term} \rangle$$

where the set of variables is an infinite sequence of identifiers, say, in the sense of ALGOL 60.

The above syntax is unambiguous as it generates a fully parenthesized form for each λ -term. We do not allow for redundant parentheses and do not use additional rules for omitting unnecessary ones. Since we are using identifiers, i.e., strings of characters to represent variables, we have to distinguish somehow between xy as a single variable and $x\ y$ as the application of x to y . For the latter case we have, therefore, the notation $(x)y$ as required by our syntax. Also, it can be observed that according to our grammar functional application associates to the right. Namely, $(x)(y)z$ is the application of x to the term $(y)z$, while $((x)y)z$ denotes the application of $(x)y$ to z .

In this paper we shall use small x, y, z , as generic names for arbitrary variables, and thus, they may occasionally stand for the same variable. Arbitrary λ -terms will be denoted by capital letters. Two λ -terms, P and Q , are equal, in symbols $P = Q$, if Q is an exact (symbol by symbol) copy of P . Note that the relation denoted by $=$ is the syntactical identity of λ -terms based on the given context-free grammar only. Each λ -term has a unique parsing tree in that grammar which can be used to determine its subterms. A λ -term may, of course, have several occurrences of the same subterm.

An occurrence of a variable x in a λ -term P is *bound* if it is inside a subterm with the form $\lambda x.Q$, otherwise it is *free*. The set of free variables of a λ -term P will be denoted by $\varphi(P)$. This can be defined inductively as follows.

DEFINITION 1 (*the set of free variables of a λ -term*).

- (i) $\varphi(x) = \{x\}$.
- (ii) $\varphi(\lambda y.P) = \varphi(P) - \{y\}$.
- (iii) $\varphi((P)Q) = \varphi(P) \cup \varphi(Q)$.

Next we define the axiom system A_0 where no substitution occurs.

DEFINITION 2. The axiom system A_0 consists of the following axiom-schemes.

α -rule:

- (α) $\lambda x.P \rightarrow \lambda y.(\lambda x.P)y$ for any $y \notin \varphi(P)$.

β -rules:

- ($\beta 1$) $(\lambda x.x)Q \rightarrow Q$.
- ($\beta 2$) $(\lambda x.y)Q \rightarrow y$ if $x \neq y$.
- ($\beta 3$) $(\lambda x.\lambda x.P)Q \rightarrow \lambda x.P$.
- ($\beta 4$) $(\lambda x.\lambda y.P)Q \rightarrow \lambda y.(\lambda x.P)Q$ if $x \neq y$: and $x \notin \varphi(P)$ or $y \notin \varphi(Q)$.
- ($\beta 5$) $(\lambda x.(P_1)P_2)Q \rightarrow ((\lambda x.P_1)Q)(\lambda x.P_2)Q$.

A term of the form $\lambda x.P$ is called an *α -redex* without any restriction. However, a term of the form $(\lambda x.P)Q$ is a *β -redex* if and only if it has the form of the left-hand side of a β -rule and satisfies its conditions. In particular a λ -term of the form

$$(\lambda x.\lambda y.P)Q$$

with $x \neq y$, $x \in \varphi(P)$, and $y \in \varphi(Q)$ is not a β -redex.

Replacing a redex by the right-hand side of the corresponding rule is called a *contraction*. It should be clear that the *contractum* of a β -redex (i.e., the result of its contraction) is unique, since for every β -redex there is at most one β -rule that can be applied to it.

Our α -rule above is, in fact, an expansion rather than contraction. Nevertheless, we shall use the word "contraction" in a technical sense for every axiom. An axiom of A_0 , i.e., an instance of one of its axiom-schemes, is a simple replacement rule (or rewriting rule) in the usual sense. Note that our α -contraction alone would not replace a bound variable by another but the β -rules can take care of it thereafter.

On the basis of our axiom system A_0 we can define the convertibility of λ -terms and thus, the theory of λ -conversion can be developed without explicitly using substitution.

DEFINITION 3 (*reduction in one step*). The relation $M \Rightarrow N$ (read M reduces directly to N) is defined inductively as follows:

- (i) $M \Rightarrow N$ whenever $M \rightarrow N$ is an axiom.
- (ii) If $M \Rightarrow N$ for some λ -terms M and N , then also $\lambda x.M \Rightarrow \lambda x.N$ for any variable x .
- (iii) If $M \Rightarrow N$ for some λ -terms M and N , then both $(M)T \Rightarrow (N)T$ and $(T)M \Rightarrow (T)N$ for any λ -term T .
- (iv) $M \Rightarrow N$ only in those cases as specified by (i) through (iii).

DEFINITION 4 (*reduction*). We say that M reduces to N , in symbols $M \stackrel{*}{\Rightarrow} N$, if $M = N$ or there is a λ -term T such that $M \stackrel{*}{\Rightarrow} T$ and $T \Rightarrow N$. (In other words the relation $\stackrel{*}{\Rightarrow}$ is the reflexive and transitive closure of \Rightarrow .)

DEFINITION 5 (*λ -convertibility, λ -equality*). We say that M and N are convertible λ -terms, in symbols $M \Leftrightarrow N$, iff $M \stackrel{*}{\Rightarrow} N$ or $N \stackrel{*}{\Rightarrow} M$ or there exists a λ -term T such that $M \Leftrightarrow T$ and $T \Leftrightarrow N$. (In other words the λ -convertibility is the symmetric and transitive closure of \Rightarrow .)

Another way of defining λ -convertibility would be to change first the axioms into equalities (i.e., two-way rules), and then define convertibility in one step similarly to the reduction in one step. Then, its reflexive and transitive closure will define the relation of λ -convertibility.

The relation \Leftrightarrow is clearly an equivalence (reflexive, symmetric, and transitive) relation on λ -terms defining the classes of convertible λ -terms. For variables x and y , $x \Leftrightarrow y$ iff $x = y$. Also, if $x = y$ then $\lambda x.\lambda y.x \Leftrightarrow \lambda x.\lambda y.y$. The latter assertion is not obvious and is in fact the consequence of the Church-Rosser theorem.

Before comparing our A_0 with the conventional axiom system A_1 , we show some basic lemmas.

LEMMA 1. *If $P \stackrel{*}{\Rightarrow} Q$ and $x \in \varphi(Q)$ then $x \in \varphi(P)$. The proof follows immediately from the fact that free variables may only disappear but can never be introduced by any contraction.*

LEMMA 2. *For any variable x and λ -term P we have $(\lambda x.P)x \stackrel{*}{\Rightarrow} P$.*

Proof. This can be shown by induction on the structure of P . Namely,

$(\lambda x.x)x \rightarrow x$ by $\beta 1$

$(\lambda x.y)x \rightarrow y$ by $\beta 2$

$(\lambda x.\lambda x.R)x \rightarrow \lambda x.R$ by $\beta 3$

$(\lambda x.\lambda y.R)x \rightarrow \lambda y.(\lambda x.R)x$ for $x \neq y$ by $\beta 4$

where $(\lambda x.R)x \stackrel{*}{\Rightarrow} R$ by the induction hypothesis, and finally

$$(\lambda x.(P_1)P_2)x \rightarrow ((\lambda x.P_1)x)(\lambda x.P_2)x \stackrel{*}{\Rightarrow} (P_1)P_2$$

by $\beta 5$ and the induction hypothesis.

LEMMA 3. *If $x \notin \varphi(P)$ then for every Q we have $(\lambda x.P)Q \stackrel{*}{\Rightarrow} P$.*

Proof. Again we use induction on the structure of P . If $P = y$ then $y \neq x$ and $(\lambda x.y)Q \rightarrow y$ by $\beta 2$. If $P = \lambda y.R$ for some y and R then either $y = x$ and thus,

$$(\lambda x.\lambda x.R)Q \rightarrow \lambda x.R$$

by $\beta 3$, or else $y \neq x$ and $x \notin \varphi(R)$ which implies

$$(\lambda x.\lambda y.R)Q \rightarrow \lambda y.(\lambda x.R)Q \stackrel{*}{\Rightarrow} \lambda y.R$$

by $\beta 4$ and by the induction hypothesis. For $P = (P_1)P_2$ we use $\beta 5$ and the induction hypothesis which completes the proof.

In the standard presentations of λ -calculus (e.g., in [7]) the operation of substitution is defined first and used in the definition of convertibility. But the usual definition of substitution already involves the idea of α -conversion (or congruence) which makes it unclear from a theoretical point of view. Here we define substitution without worrying about the exact choice of the new bound variable.

DEFINITION 6 (*Substitution*). The substitution of Q for the free occurrences of the variable x in P , denoted by $[Q/x]P$, is defined recursively as follows:

- (1) $[Q/x]x = Q$
- (2) $[Q/x]y = y$ for $x \neq y$
- (3) $[Q/x]\lambda x.P = \lambda x.P$
- (4) $[Q/x]\lambda y.P = \lambda y.[Q/x]P$ if $x \neq y$ and $y \notin \varphi(Q)$
- (5) $[Q/x]\lambda y.P = \lambda z.[Q/x][z/y]P$ if $x \neq y$ and $y \in \varphi(Q)$, for any $z \notin \varphi((P)Q) \cup \{x\}$,
- (6) $[Q/x](P_1)P_2 = ([Q/x]P_1)[Q/x]P_2$.

Note that in (5) we have not specified precisely the variable z . It can be shown that our definition is equivalent to the standard one. Now, we prove an important lemma.

LEMMA 4. $(\lambda x.P)Q \stackrel{\#}{\Rightarrow} [Q/x]P$ for every x, P , and Q .

Proof. We use induction on the number of occurrences of variables in P .

Basis. If P has a single occurrence of a variable then the assertion is trivial by β_1 and β_2 and Definition 6.

Induction step: Assume that the assertion is true for all λ -terms with at most n occurrences of variables and let P have $n+1$ of them. Then P has the form either $(P_1)P_2$ or $\lambda y.R$. For the former case the result follows immediately from the induction hypothesis. For $P = \lambda y.R$ we have three subcases:

- (a) $x = y$. In this case the assertion is trivial.
- (b) $x \neq y$ and $y \notin \varphi(Q)$. Here we have

$$(\lambda x.\lambda y.R)Q \rightarrow \lambda y.(\lambda x.R)Q \stackrel{\#}{\Rightarrow} \lambda y.[Q/x]R$$

by β_4 and the induction hypothesis.

- (c) $x \neq y$ and $y \in \varphi(Q)$. In this case we get

$$\begin{aligned} (\lambda x.\lambda y.R)Q &\Rightarrow (\lambda x.\lambda z.(\lambda y.R)z)Q \\ &\Rightarrow \lambda z.(\lambda x.(\lambda y.R)z)Q \stackrel{\#}{\Rightarrow} \lambda z.(\lambda x.[z/y]R)Q \stackrel{\#}{\Rightarrow} z.[Q/x][z/y]R \end{aligned}$$

by α , β_4 , and by the induction hypothesis. In the last step we have assumed that $[z/y]R$ has no more occurrences of variables than R does. This is easy to show separately by induction which completes the proof.

Note that in case (c) the variable z can be chosen for the α -conversion to be the same as requested by Definition 6.

We conclude this section by a theorem relating our axiom system A_0 to the conventional one. Let us recall the conventional axiom system.

DEFINITION 7. The axiom system A_1 consists of the following axiom-schemes:

- (α') $\lambda x.P \rightarrow \lambda y.[y/x]P$ for any $y \notin \varphi(P)$,
- (β') $(\lambda x.P)Q \rightarrow [Q/x]P$.

Reduction and convertibility can be defined as before simply by using A_1 in place of A_0 .

THEOREM 1. For any two λ -terms, M and N , if $M \stackrel{\#}{\Rightarrow} N$ holds in A_1 then it also holds in A_0 .

Proof. It is enough to show that $M \stackrel{\#}{\Rightarrow} N$ holds in A_0 whenever $M \rightarrow N$ is an axiom in A_1 . If $M \rightarrow N$ is an instance of α' then $M = \lambda x.P$, and $N = \lambda y.[y/x]P$ for some x, y, P , such that $y \notin \varphi(P)$. But then, in A_0 we have

$$(\lambda x.P) \rightarrow \lambda y.(\lambda x.P)y \stackrel{\#}{\Rightarrow} \lambda y.[y/x]P$$

by α and Lemma 4. If, on the other hand, $M \rightarrow N$ is an instance of β' then $M = (\lambda x.P)Q$, $N = [Q/x]P$ for some x , P , and Q and thus,

$$(\lambda x.P)Q \xrightarrow{*} [Q/x]P$$

by Lemma 4 which completes the proof.

The converse of Theorem 1 is obviously false. For instance, $\lambda x.P \xrightarrow{*} \lambda z.(\lambda x.P)z$ is not true in A_1 . Convertibility, however, is the same in both systems. Indeed, if $M \rightarrow N$ is an axiom in A_0 then there is always some λ -term T , such that both M and N are reducible to T in A_1 . For instance, both $(\lambda x.(P_1)P_2)Q$ and $((\lambda x.P_1)Q)(\lambda x.P_2)Q$ are reducible to $([Q/x]P_1)[Q/x]P_2$. Hence they are convertible not only in A_0 but also in A_1 , and thus, the Church–Rosser property is valid also for A_0 . (A_0 is a compatible refinement of A_1 .) Incidentally, the Church–Rosser theorem could be shown directly for A_0 , but it does not seem to be any easier than for A_1 .

3. A new technique to perform substitution. The actual implementation of β -reduction using a conventional programming language has led us to the idea of renaming.

DEFINITION 8 (renaming). The renaming of a variable x in P by z , denoted by $[z//x]P$, is defined recursively as follows:

- (1) $[z//x]x = z$
- (2) $[z//x]y = y$ if $x \neq y$
- (3) $[z//x]\lambda x.P = \lambda z.[z//x]P$
- (4) $[z//x]\lambda y.P = \lambda y.[z//x]P$ if $x \neq y$
- (5) $[z//x](P_1)P_2 = ([z//x]P_1)[z//x]P_2$

Renaming can be viewed as a brute force replacement of all occurrences of x by z without any respect to the possible bindings of x . Interestingly enough, such a renaming seems to be useful in improving our axiom system. First we prove a basic lemma about renaming.

LEMMA 5. For every x , z , and P such that z is neither free nor bound in P we have

$$(\lambda x.P)z \xrightarrow{*} [z//x]P$$

with respect to A_0 .

Proof. We use induction on the structure of P . If P is a variable then the assertion is trivial.

For $P = \lambda x.R$ we get

$$(\lambda x.\lambda x.R)z \rightarrow \lambda x.R \rightarrow \lambda z.(\lambda x.R)z \xrightarrow{*} \lambda z.[z//x]R = [z//x]\lambda x.R$$

by the induction hypothesis and Definition 8.

For $P = \lambda y.R$ with $x \neq y \neq z$ we have

$$(\lambda x.\lambda y.R)z \rightarrow \lambda y.(\lambda x.R)z \xrightarrow{*} \lambda y.[z//x]R = [z//x]\lambda y.R.$$

For $P = (P_1)P_2$ the result follows immediately from the induction hypothesis and this completes the proof.

Now, we define our axiom system A_2 which is the enhanced version of A_0 .

DEFINITION 9. The axiom system A_2 consists of the following axiom-schemes:

- (α'') $\lambda x.P \rightarrow \lambda z.[z//x]P$ for any z which is neither free nor bound in P ,
- ($\beta 1''$) $(\lambda x.x)Q \rightarrow Q$,
- ($\beta 2''$) $(\lambda x.P)Q \rightarrow P$ if $x \notin \varphi(P)$,
- ($\beta 3''$) $(\lambda x.\lambda y.P)Q \rightarrow \lambda z.(\lambda x.[z//y]P)Q$ if $y \neq x \in \varphi(P)$, for any $z \notin \varphi((P)Q) \cup \{x\}$ which is not bound in P ,
- ($\beta 4''$) $(\lambda x.(P_1)P_2)Q \rightarrow ((\lambda x.P_1)Q)(\lambda x.P_2)Q$ if $x \in \varphi((P_1)P_2)$.

In order to decide whether to apply $\beta 2''$ or $\beta 3''$ we have to scan through P to check if $x \in \varphi(P)$. At the same time we can also perform the renaming $[z//y]P$. But the point is that we never have to examine Q to check whether or not $y \in \varphi(Q)$, we only have to insure that $z \notin \varphi(Q)$. This can be done by choosing a fresh "system" variable each time we need one in the course of the reduction. The classical definition of substitution requires the checking of Q as well as of P . (It would be possible to cancel (4) from Definition 6 and use always (5) instead, but then the definition of $[z/x]\lambda y.P$ would become

$$[z/x]\lambda y.P = \lambda v.[z/x][v/y]P$$

which results in an excessive number of new bound variables.) That is why our renaming is more efficient.

THEOREM 2. *For any two λ -terms, M and N , if $M \Rightarrow N$ holds in A_2 then it also holds in A_0 .*

Proof. It is enough to show that $M \Rightarrow N$ holds in A_0 whenever $M \rightarrow N$ is an axiom in A_2 .

If $M \rightarrow N$ is an instance of α'' then the assertion follows from Lemma 5.

If $M \rightarrow N$ is an instance of $\beta 1''$ or $\beta 4''$ then the assertion is trivial while for $\beta 2''$ it follows from Lemma 3.

If $M \rightarrow N$ is an instance of $\beta 3''$ then we have

$$\begin{aligned} (\lambda x.\lambda y.P)Q &\rightarrow (\lambda x.\lambda z.(\lambda y.P)z)Q \\ &\rightarrow \lambda z.(\lambda x.(\lambda y.P)z)Q \Rightarrow \lambda z.(\lambda x.[z//y]P)Q \end{aligned}$$

by α , $\beta 4$ and Lemma 5 which completes the proof.

This means that renaming (which is an extremely primitive operation) combined with the axioms of A_2 is a convenient tool for computing normal forms of λ -terms.

On the basis of our renaming we can also modify the definition of substitution although substitution need not be mentioned at all in our development of lambda-conversion.

DEFINITION 10 (substitution). The substitution of Q for the free occurrences of the variable x in P can be defined recursively as follows:

$$(1) [Q/x]x = Q$$

$$(2) [Q/x]P = P \text{ if } x \notin \varphi(P)$$

(3) $[Q/x]\lambda y.P = \lambda z.[Q/x][z//y]P$ if $y \neq x \in \varphi(P)$, for any $z \notin \varphi((P)Q) \cup \{x\}$ which is not bound in P

$$(4) [Q/x](P_1)P_2 = ([Q/x]P_1)[Q/x]P_2$$

It can be shown that this definition is equivalent to Definition 6, only the choice of the new bound variables remains open in both cases. (This is usually reflected in the so called α -congruence, or α -convertibility.) Definition 10 actually provides for a new technique to perform substitution in a different, but equally correct way.

It is also easy to show that a reduction $M \Rightarrow N$ in the conventional lambda-calculus implies that $M \Rightarrow N$ holds also in A_2 .

4. Computing normal forms of lambda-terms. Our program for computing normal forms of lambda-terms has been designed in such a way that it is easy to implement in any programming language. It is written in PL/I but makes no use of the specific features of that language. This means, for instance, that input characters are read one by one and coded immediately as integers.

The overall structure of the program is extremely simple. It is based on a slightly modified version of the Turing machine called two-pushdown automaton (see Fig. 1)

where the nonblank portion of the Turing tape is represented by the contents of the two pushdown stacks. The top of each stack is scanned by a read-write head of the finite control device to determine the next move.

Initially the lambda-term to be processed is placed into the second stack such that its first symbol is on the top and the last one is at the bottom of the second stack, while the first stack is empty. In the course of processing the term left-to-right, changes are made at the tops of the stacks and symbols will be copied from the second stack into the first one. Occasionally, a reverse scan is made by copying the first stack into the second.

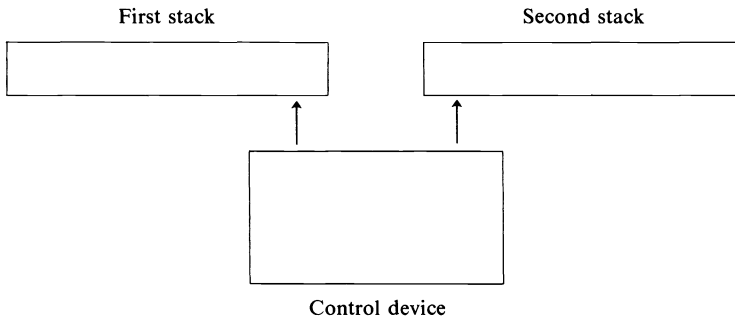


FIG. 1.

Each cell in the stacks contains an integer representing a variable or a special symbol like λ , $.$, $($, $)$, or $;$. The latter are encoded as negative numbers while the code of a variable is a pointer to the location of its identifier in a vocabulary.

The computing process is based on our axiom system A_2 with simple replacement rules. For any given redex $(\lambda x.P)Q$ it is easy to determine which of the β -rules is applicable. First we check if $x \in \varphi(P)$. (The search stops at the first free occurrence of x .) If $x \notin \varphi(P)$ then we apply β_2'' . Otherwise the symbol occurring next to the first dot in the redex determines the type of the redex. If that symbol is the same as the bound variable then the redex must have the form $(\lambda x.x)Q$. If that symbol is λ , or $($ then we have a β_3'' , resp. β_4'' redex.

The application of β_1'' is just the popping of $(\lambda x.x)$ from the first stack. Applying β_2'' means shifting P into the first stack and popping Q from the second. The application of β_3'' involves a scan for all occurrences of y in P . They will be replaced by a fresh system variable. But, as we have said, we do not have to check Q and thus, it remains untouched in the second stack. Finally, the application of β_4'' requires copying Q into the first stack from below P_2 on the second stack.

The contraction of the redexes is performed essentially from left to right. This means that the algorithm works with the leftmost redex except when it has just made a β_3'' contraction. Note that the β_3'' rule is more like the preparation for an actual contraction than a contraction by itself. Therefore, the redex to be contracted next after a β_3'' contraction is always its "trace", i.e., the resulting redex of the form $(\lambda x.[z//y]P)Q$. Otherwise, after each contraction we continue with the leftmost redex in the entire expression. We claim that this order of evaluation results in the normal form whenever it exists. To see this, note that a sequence of β_3'' contractions followed by an "actual" contraction as described, amounts to a contraction of the leftmost redex in the usual sense.

Several λ -terms separated by semicolons can be processed by the program one after the other. Another very useful feature of the program is the possibility of using

definitions. Namely, in the input we can assign λ -terms to variables this way:

$$I = \lambda x.x,$$

$$K = \lambda x.\lambda y.x,$$

$$S = \lambda x.\lambda y.\lambda z.((x)z)(y)z.$$

Then, if we write

$$(((K)I)(K)A)B,$$

we get the result B which is computed as the normal form of

$$(\lambda I.(\lambda K.(((K)I)(K)A)B)\lambda x.\lambda y.x)\lambda x.x.$$

Or we can define

$$\text{DELTA} = \lambda x.(x)x,$$

$$\text{OMEGA} = (\text{DELTA})\text{DELTA},$$

and compute

$$(\lambda x.((x)\lambda x.\lambda y.\lambda z.x)\text{OMEGA})\lambda y.((y)z)\text{OMEGA};$$

which gives the result z as its normal form. All the program does with a definition like

$$\text{DELTA} = \lambda x.(x)x,$$

is that it places the string

$$(\lambda \text{ DELTA.}$$

in the first stack and

$$)\lambda x.(x)x$$

in the second. The λ -term that follows will then be placed automatically in between, that is on top of the second part. Hence, each occurrence of DELTA in that λ -term will be replaced by $\lambda x.(x)x$ during the reduction that follows. Recursive definitions, however, cannot be processed automatically by the program in its present form. They will be detected and signaled by the program, but it would not have recourse to the fixpoint combinator in order to resolve recursive equations.

λ -terms being in normal form are easily compared by using renaming in place of α -conversion.

The above described rudimentary algorithm can be improved upon in various ways. The most important improvements could be achieved by a highly parallel reduction strategy working concurrently on different parts of the λ -term. We feel that our reduction rules are very helpful in designing such a strategy.

REFERENCES

- [1] H. BARENDREGT, *The Lambda Calculus, Its Syntax and Semantics*, North-Holland, Amsterdam, 1981.
- [2] K. J. BERKLING AND E. FEHR, *A modification of the λ -calculus as a base for functional programming languages*, Proc. 9th ICALP Conference, Lecture Notes in Computer Science 140, Springer-Verlag, Berlin-Heidelberg-New York, 1982, pp. 35-47.
- [3] N. G. DE BRUIJN, *Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation with application to the Church-Rosser theorem*, Indag. Math., 34 (1972), pp. 381-392.
- [4] ———, *Lambda calculus with namefree formulas involving symbols that represent reference transforming mappings*, Indag. Math., 40 (1978), pp. 348-356.

- [5] A. CHURCH, *Introduction to Mathematical Logic, Part I*, Ann. of Math. Studies 13, Princeton Univ. Press, Princeton, NJ, 1944.
- [6] H. B. CURRY, R. FEYS AND W. CRAIG, *Combinatory Logic, Volume I*, North-Holland, Amsterdam, 1958.
- [7] J. R. HINDLEY, B. LECHER AND J. P. SELDIN, *Introduction to Combinatory Logic*, London Mathematical Society Lecture Note Series 7, Cambridge Univ. Press, London, 1972.
- [8] D. KALISH AND R. MONTAGUE, *On Tarski's formalization of predicate logic with identity*, Arch. Math. Logik. Grundl., 7 (1965), pp. 81-101.
- [9] D. MONK, *Substitutionless predicate logic with identity*, Arch. Math. Logik. Grundl., 7 (1965), pp. 102-121.
- [10] GY. REVESZ, *λ -calculus without substitution* (abstract), Bull. EATCS, 11 (June 1980), p. 140.
- [11] M. SCHÖNFINKEL, *Über die Bausteine der mathematischen Logik*, Math. Annal., 92 (1924), pp. 305-316.
- [12] J. STAPLES, *A lambda calculus with naive substitution*, J. Austral. Math. Soc., Ser. A, 28 (1979), pp. 269-282.
- [13] A. TARSKI, *A simplified formalization of predicate logic with identity*, Arch. Math. Logik. Grundl., 7 (1965), pp. 61-79.
- [14] D. A. TURNER, *A new implementation technique for applicative languages*, Software Practice and Experience, 9 (1979), pp. 31-49.

ON THE MULTIPLICATIVE COMPLEXITY OF MODULES OVER ASSOCIATIVE ALGEBRAS*

W. HARTMANN†

Abstract. The complexity $L(A, M)$ of a finite-dimensional module M over a finite-dimensional associative algebra A is the number of nonscalar multiplications/divisions of an optimal algorithm to compute the product of an element of the algebra with an element of the module.

It is known that

$$L(A, A) \geq 2 \cdot \dim A - s,$$

where s is the number of maximal two-sided ideals of A . We give a generalization of this lower bound to arbitrary A -modules M .

Key words. multiplicative complexity, modules over associative algebras, structure theorem of Wedderburn

1. Introduction. Let k be an infinite field, A an associative k -algebra (with 1) of dimension n and M a (left-) A -module of dimension m . Let e_1, \dots, e_n be a basis of the vector space A , f_1, \dots, f_m a basis of the vector space M and let

$$e_i \cdot f_j = \sum_{l=1}^m \tau_{ijl} f_l$$

with $\tau_{ijl} \in k$. Then we have

$$\left(\sum_{i=1}^n \xi_i e_i \right) \cdot \left(\sum_{j=1}^m \eta_j f_j \right) = \sum_{l=1}^m \left(\sum_{i,j=1}^{n,m} \tau_{ijl} \xi_i \eta_j \right) f_l.$$

Let $x_1, \dots, x_n, y_1, \dots, y_m$ be indeterminates over k . In the following we make use of Ostrowski's model of computation in $k(x_1, \dots, x_n, y_1, \dots, y_m)$, i.e. we allow linear operations, such as additions, subtractions and scalar multiplications at no cost, and we minimize the number of nonscalar multiplications/divisions.

DEFINITION. The complexity of the A -module M is

$$(1) \quad L(A, M) = L \left(\left\{ \sum_{i,j=1}^{n,m} \tau_{ijl} x_i y_j \mid 1 \leq l \leq m \right\} \right).$$

$L(A, M)$ does not depend on the choice of the basis. In [1] Alder-Strassen prove a general lower bound for the complexity of multiplication in an algebra A :

$$L(A, A) \geq 2 \cdot \dim A - \text{the number of maximal two-sided ideals of } A.$$

We prove a generalization of this lower bound to arbitrary A -modules M .

2. Results. For the facts about algebras and modules, which will be needed later, see e.g. [3, Chaps. 2, 3].

First we take into account, that isomorphic A -modules have the same complexity. Furthermore, given an A -module M and a morphism of algebras $f: A' \rightarrow A$, then M can also be regarded as an A' -module, and for f surjective we have

$$L(A', M) = L(A, M).$$

* Received by the editors July 25, 1983.

† Institut für Angewandte Mathematik der Universität Zurich, Rämistrasse 74, CH-8001 Zürich, Switzerland.

$(L(A', M) \leq L(A, M))$ is trivial. $L(A', M) \cong L(A, M)$ follows using a section of the linear map f .

In particular for an isomorphism $f: A' \rightarrow A$ we have $L(A', M) = L(A, M)$. Let $\text{ann}(M) = \{a \in A \mid aM = 0\}$ be the annihilator of M . $\text{ann}(M)$ is a two-sided ideal of A , M regarded as an $A/\text{ann}(M)$ -module is faithful, and we have

$$L(A, M) = L(A/\text{ann}(M), M).$$

Therefore, in the following we may restrict ourselves to faithful A -modules. Given an A -module M and a B -module N , we can regard M and N as $A \times B$ -modules by the projections $A \times B \rightarrow A$ and $A \times B \rightarrow B$ respectively. Hence $M \oplus N$ can be regarded as an $A \times B$ -module. Vice versa, each $A \times B$ -module P is of the form $M \oplus N$ with M an A -module and N a B -module (namely $M = (1, 0) \cdot P$ and $N = (0, 1) \cdot P$). By analogy with the strategy of Alder-Strassen, we now reduce the problem to semisimple algebras. We denote the radical of A by $\text{rad } A$, and the radical of M by $\text{rad } M$. We have $\text{rad } M = \text{rad } A \cdot M$, and $M/\text{rad } M$ can be considered as an $A/\text{rad } A$ -module.

THEOREM 1. *Let M be a faithful A -module. Then*

$$L(A, M) \cong L(A/\text{rad } A, M/\text{rad } M) + \dim(\text{rad } A) + \dim(\text{rad } M).$$

$A/\text{rad } A$ is semisimple and is thus, as a consequence of the structure theorem of Wedderburn, isomorphic to a direct product of full matrix algebras $K_i^{n_i \times n_i}$ over k -division algebras K_i . Up to isomorphisms the modules of such a factor $K^{n \times n}$ are of the form $K^{n \times m}$ (scalar multiplication = matrix product).

Repeated application of the next theorem therefore yields lower bounds for the complexity of modules over semisimple algebras.

THEOREM 2. *Let K be a k -division algebra, $\dim_k K = \lambda$, B an arbitrary algebra, N an arbitrary B -module. Then for $n, m \geq 1$*

$$L(K^{n \times n} \times B, K^{n \times m} \oplus N) \cong f(n, m, \lambda) + L(B, N)$$

where

$$f(n, 1, \lambda) = \lambda n^2 + (\lambda - 1)n,$$

$$f(n, m, \lambda) = \lambda \cdot \max\{n^2, nm - m + n\} + \lambda nm - 1 \quad \text{for } n, m > 1,$$

$$f(1, m, \lambda) = (2\lambda - 1)m.$$

The above theorem can also be formulated coordinate-free.

Let A be a simple algebra and $M \neq 0$ an A -module. Let E be a simple submodule of M , F a simple submodule of the $\text{End}_A M$ -module M and K^{op} the algebra of endomorphisms of E . Further let B be an arbitrary algebra and N an arbitrary B -module. Then

$$L(A \times B, M \oplus N) \cong \dim_k A + \dim_k M - \dim_k M + L(B, N), \quad \dim_k F = 1;$$

$$L(A \times B, M \oplus N) \cong \max\{\dim_k A, \dim_k M - (\dim_k F - \dim_k E)\} + \dim_k M - 1 + L(B, N), \quad \dim_k E, \dim_k F > 1;$$

$$L(A \times B, M \oplus N) \cong 2 \dim_k M - \dim_k M + L(B, N), \quad \dim_k E = 1.$$

Using the above theorems we obtain the following corollaries.

COROLLARY 1 (A. Alder, V. Strassen [1]). *Let A be an arbitrary algebra. Then*

$$L(A, A) \cong 2 \cdot \dim A - s,$$

where s is the number of maximal two-sided ideals of A .

Proof. Considered as an A -module, A is faithful. Theorem 1 yields

$$L(A, A) \cong L(A/\text{rad } A, A/\text{rad } A) + 2 \dim(\text{rad } A).$$

$A/\text{rad } A$ is isomorphic to a direct product of simple algebras, say

$$A \cong \prod_{i=1}^s K_i^{n_i \times n_i}, \quad \dim_k K_i = \lambda_i.$$

s coincides with the number of maximal two-sided ideals of A . Repeated applications of Theorem 2 with $n = m$ yields

$$\begin{aligned} L(A/\text{rad } A, A/\text{rad } A) &\cong \sum_{n_i=1} (2\lambda_i - 1) + \sum_{n_i > 1} (2\lambda_i n_i^2 - 1) \\ &= 2 \cdot \dim(A/\text{rad } A) - s. \end{aligned}$$

Together we get

$$L(A, A) \cong 2 \cdot \dim A - s.$$

The lower bound for $n = 1$ together with the well-known upper bound yields the following corollary.

COROLLARY 2 (L. Auslander, S. Winograd [2]). *Let K be a k -division algebra, $\dim_k K = \lambda$. Then*

$$L(K, K^m) \cong (2\lambda - 1)m.$$

For the multiplication of matrices over k , we get the following corollary.

COROLLARY 3.

$$L(k^{n \times n}, k^{n \times m}) \cong \begin{cases} n^2, & m = 1, \\ \max \{n^2 + nm - 1, 2nm + n - m - 1\}, & m > 1. \end{cases}$$

Thus for $m = 1$ and $m = n$ we get the well-known bounds. For $n = 2$ we obtain particularly the next corollary.

COROLLARY 4.

$$L(k^{2 \times 2}, k^{2 \times m}) \cong 3m + 1.$$

It is known that

$$L(k^{2 \times 2}, k^{2 \times m}) \leq 3m + 2$$

(S. Winograd, [5]). Since $L(k^{2 \times 2}, k^{2 \times 2}) = 7$ the lower bound is sharp at least for $m = 2$.

Let D_{2n} be the dihedral group of order $2n$, $\mathbb{C}[D_{2n}]$ its group algebra over $k = \mathbb{C}$. If n is odd we have (using character theory)

$$\mathbb{C}[D_{2n}] \cong \mathbb{C}^2 \times (\mathbb{C}^{2 \times 2})^{(n-1)/2},$$

if n is even we have

$$\mathbb{C}[D_{2n}] \cong \mathbb{C}^4 \times (\mathbb{C}^{2 \times 2})^{n/2-1},$$

i.e. the irreducible representations of D_{2n} are either one-dimensional or two-dimensional.

Let M be an arbitrary $\mathbb{C}[D_{2n}]$ -module. Considering a decomposition of M into a direct sum of simple modules and denoting by s the number of the one-dimensional modules in this decomposition and by t the number of isotypical components belonging to two-dimensional representations, we get the following lower bound.

COROLLARY 5. $L(\mathbb{C}[D_{2n}], M) \geq s + \frac{3}{2}(\dim M - s) + t$.

Proof. Let M be a faithful $\mathbb{C}^l \times (\mathbb{C}^{2 \times 2})^t$ -module, M' an isotypical component, $M = M' \oplus M''$ and s as above. For M' belonging to a one-dimensional representation Theorem 2 with $n = \lambda = 1$ and $m = \dim M'$ yields

$$L(\mathbb{C}^l \times (\mathbb{C}^{2 \times 2})^t, M) \geq \dim M' + L(\mathbb{C}^{l-1} \times (\mathbb{C}^{2 \times 2})^t, M'').$$

For M' belonging to a two-dimensional representation Theorem 2 with $n = 2, \lambda = 1$ and $m = \frac{1}{2} \dim M'$ yields

$$\begin{aligned} L(\mathbb{C}^l \times (\mathbb{C}^{2 \times 2})^t, M) &\geq 3m + 1 + L(\mathbb{C}^l \times (\mathbb{C}^{2 \times 2})^{t-1}, M'') \\ &= \frac{3}{2} \dim M' + 1 + L(\mathbb{C}^l \times (\mathbb{C}^{2 \times 2})^{t-1}, M''). \end{aligned}$$

Taking into account, that s is the sum of the dimensions of the isotypical components belonging to one-dimensional representations, $\dim M - s$ the sum of the dimensions of the isotypical components belonging to two-dimensional representations, we get by induction

$$L(\mathbb{C}^l \times (\mathbb{C}^{2 \times 2})^t, M) \geq s + \frac{3}{2}(\dim M - s) + t.$$

Analogously we get (using $L(k^{2 \times 2}, k^{2 \times m}) \leq 3m + 2$) the upper bound

$$L(\mathbb{C}[D_{2n}], M) \leq s + \frac{3}{2}(\dim M - s) + 2t.$$

In the next corollary we determine the complexity of $k[x]/(f)$ -modules in the case, that f is squarefree. In this case $k[x]/(f)$ is semisimple. Denoting by s the number of simple modules in a direct sum decomposition of M we get the following corollary.

COROLLARY 6. *Let $f \in k[x]$ be squarefree. Then*

$$L(k[x]/(f), M) = 2 \dim M - s.$$

Proof. Let $f = p_1 \cdots p_r$ be a prime decomposition. For

$$k[x]/(f) \simeq \prod_{i=1}^r K_i$$

with $K_i = k[x]/(p_i)$ M is (up to isomorphism) a direct sum of K_i -modules $K_i^{s_i}$, $i = 1, \dots, r$. Then $s = s_1 + \dots + s_r$. By induction for $l \leq r$ we show

$$L\left(\prod_{i=1}^l K_i, \bigoplus_{i=1}^l K_i^{s_i}\right) \geq 2 \sum_{i=1}^l \dim_k K_i \cdot s_i - (s_1 + \dots + s_l).$$

(In particular $L(\prod_{i=1}^r K_i, M) \geq 2 \dim M - s$.)

The case $l = 0$ is trivial. Theorem 2 with $n = 1, \lambda = \dim_k K_l$ and $m = s$ yields

$$\begin{aligned} L\left(\prod_{i=1}^l K_i, \bigoplus_{i=1}^l K_i^{s_i}\right) &\geq (2 \dim_k K_l - 1)s_l + L\left(\prod_{i=1}^{l-1} K_i, \bigoplus_{i=1}^{l-1} K_i^{s_i}\right) \\ &\geq 2 \cdot \sum_{i=1}^l \dim_k K_i \cdot s_i - (s_1 + \dots + s_l) \quad (\text{induction hypothesis}). \end{aligned}$$

By an analogous induction we show (using Corollary 2) that the upper and lower bounds coincide.

Let C_n be the cyclic group of order n . Considering that

$$k[C_n] \simeq k[x]/(x^n - 1)$$

and that $x^n - 1$ is squarefree for $\text{char } k \nmid n$, Corollary 6 in particular describes the complexity of arbitrary $k[C_n]$ -modules for $\text{char } k \nmid n$.

3. Proofs. The multiplication $A \times M \rightarrow M$ is a bilinear map. We will consider the computational complexity of slightly more general maps, namely homogeneous quadratic maps.

DEFINITION. Let E, W be finite-dimensional k -vector spaces with bases e_1, \dots, e_n resp. f_1, \dots, f_m . A map

$$h : E \rightarrow W$$

is called quadratic, if there are quadratic forms h_1, \dots, h_m in $k[x_1, \dots, x_n]$ such that for all $\xi_1, \dots, \xi_n \in k$

$$h\left(\sum_{i=1}^n \xi_i e_i\right) = \sum_{l=1}^m h_l(\xi_1, \dots, \xi_n) f_l.$$

$L(h) = L(h_1, \dots, h_m)$ is called the complexity of h . The notion of a quadratic map and $L(h)$ do not depend on the chosen bases. If $\Phi : E' \rightarrow E, \psi : W \rightarrow W'$ are linear maps, then $\psi \circ h \circ \Phi$ is again quadratic and

$$(1) \quad L(h) \geq L(\psi \circ h \circ \Phi).$$

PROPOSITION (see [1]). *Let $h : E \rightarrow W$ be quadratic. Then $L(h) \leq r$, iff there are $u_\rho, v_\rho \in E^*, w_\rho \in W (\rho = 1, \dots, r)$ such that for all $x \in E$*

$$h(x) = \sum_{\rho=1}^r u_\rho(x) v_\rho(x) w_\rho,$$

where E^* denotes the dual of E .

The following technical lemma, which was communicated to me by V. Strassen, unifies the methods used in [1].

LEMMA 1. *Let A, B and W be finite-dimensional k -vector spaces and $h : A \times B \rightarrow W$ a bilinear map, written as multiplication.*

For all $a \in A, b \in B$ let

$$(2) \quad a \cdot b = \sum_{\rho=1}^r u_\rho(a, b) v_\rho(a, b) w_\rho$$

with $u_\rho, v_\rho \in (A \times B)^*, w_\rho \in W$ and let $X \subset A, Y' \subset Y \subset B, P \subset W$ be linear subspaces, such that the u_ρ 's with $w_\rho \notin P$ separate the points of $X \times Y'$, i.e. their restrictions to $X \times Y'$ generate the dual of $X \times Y'$. Then one of the following conditions holds.

(i) *After possibly interchanging some u_ρ with v_ρ the u_ρ 's with $w_\rho \notin P$ separate the points of $X \times Y$.*

(ii) *There exists $y \in Y \setminus Y'$ with $Ay \subseteq P + X \cdot Y$.*

(Analogous with sides interchanged.)

Proof. Assume (i) to be false. By permuting the terms of the sum (2) and by interchanging some u_ρ with v_ρ , we can assume w.l.o.g.

$$w_1, \dots, w_p \in P, w_{p+1}, \dots, w_r \in P,$$

$$u_{p+1}, \dots, u_r \text{ separate the points of } X \times Y',$$

$$u_{p+1}, \dots, u_q \text{ are linearly independent on } X \times Y,$$

$$u_{q+1}, \dots, u_r, v_{q+1}, \dots, v_r \text{ are linearly dependent on}$$

$$u_{p+1}, \dots, u_q \text{ as linear forms on } X \times Y,$$

$$u_{p+1}, \dots, u_r \text{ do not separate the points of } X \times Y.$$

In particular u_{p+1}, \dots, u_q separate the points of $X \times Y'$, but not those of $X \times Y$. That is, there exist $x \in X, y \in Y \setminus Y'$ with $u_{p+1}(x, y) = \dots = u_q(x, y) = 0$. It follows

$$\begin{aligned} u_{q+1}(x, y) &= \dots = u_r(x, y) = 0, \\ v_{q+1}(x, y) &= \dots = v_r(x, y) = 0. \end{aligned}$$

If $a \in A, b \in B$ are arbitrary we use the linear independence of u_{p+1}, \dots, u_q on $X \times Y$ to find $s \in X, t \in Y$ such that

$$\forall \rho (p+1 \leq \rho \leq q) \quad u_\rho(a, b) = -u_\rho(s, t)$$

thus $u_{p+1}(a+s, b+t) = \dots = u_q(a+s, b+t) = 0$. So we get

$$\begin{aligned} (a+s)y + x(b+t+y) &= (a+s+x)(b+t+y) - (a+s)(b+t) \\ &= \sum_{\rho=1}^r (u_\rho(a+s, b+t) + u_\rho(x, y))(v_\rho(a+s, b+t) + v_\rho(x, y))w_\rho \\ &\quad - \sum_{\rho=1}^r u_\rho(a+s, b+t)v_\rho(a+s, b+t)w_\rho \\ &= \sum_{\rho=1}^r (u_\rho(a+s, b+t)v_\rho(x, y) \\ &\quad + u_\rho(x, y)v_\rho(a+s, b+t) + u_\rho(x, y)v_\rho(x, y))w_\rho. \end{aligned}$$

Since

$$\begin{aligned} u_{p+1}(x, y) &= \dots = u_r(x, y) = 0, \\ u_{p+1}(a+s, b+t) &= \dots = u_q(a+s, b+t) = 0, \\ v_{q+1}(x, y) &= \dots = v_r(x, y) = 0, \end{aligned}$$

it follows that

$$\begin{aligned} (a+s)y + x(b+t+y) &= \sum_{\rho=1}^p (u_\rho(a+s, b+t)v_\rho(x, y) \\ &\quad + u_\rho(x, y)v_\rho(a+s, b+t) + u_\rho(x, y)v_\rho(x, y))w_\rho \in P. \end{aligned}$$

Setting $b=0$, we obtain

$$ay + xt + (s+x)y \in P$$

hence

$$Ay \subset P + X \cdot Y.$$

Proof of Theorem 1. Let $L(A, M) = r$. Then there are $u_\rho, v_\rho \in (A \times M)^*, w_\rho \in M$ such that

$$(3) \quad \forall a \in A, x \in M \quad a \cdot x = \sum_{\rho=1}^r u_\rho(a, x)v_\rho(a, x)w_\rho.$$

It suffices to find a representation (3) with the additional property that

$$(4) \quad u_1, \dots, u_r \text{ separate the points of } \text{rad } A \times \text{rad } M.$$

Assume (4) and let $q_1 = \dim(\text{rad } A), q_2 = \dim(\text{rad } M)$. W.l.o.g. we can assume

$$u_1, \dots, u_{q_1+q_2} \text{ linearly independent on } \text{rad } A \times \text{rad } M.$$

Let $E = \{u_1 = \dots = u_{q_1+q_2} = 0\} \subset A \times M$. Then we have $E \cap (\text{rad } A \times \text{rad } M) = 0$.

Let $h : E \rightarrow M$ be the restriction of the multiplication $A \times M \rightarrow M \cdot h$ is a quadratic map and

$$L(h) \leq r - (q_1 + q_2) \quad \left(\text{for } h(a, x) = \sum_{\rho=q_1+q_2+1}^r u_\rho(a, x)v_\rho(a, x)w_\rho \right).$$

Let μ (resp. μ') be the multiplication $A \times M \rightarrow M$ (resp. $A/\text{rad } A \times M/\text{rad } M \rightarrow M/\text{rad } M$). The commutative diagram

$$\begin{array}{ccc} A \times M & \xrightarrow{\mu} & M \\ \downarrow & & \downarrow \\ A/\text{rad } A \times M/\text{rad } M & \xrightarrow{\mu'} & M/\text{rad } M \end{array}$$

yields (by restriction) a commutative diagram

$$\begin{array}{ccc} E & \xrightarrow{h} & M \\ \alpha \downarrow & & \downarrow \\ A/\text{rad } A \times M/\text{rad } M & \xrightarrow{\mu'} & M/\text{rad } M \end{array}$$

Since $E \cap (\text{rad } A \times \text{rad } M) = 0$, α is an isomorphism. Therefore we have

$$\begin{aligned} L(A/\text{rad } A, M/\text{rad } M) &= L(\mu') \leq L(h) \leq r - (q_1 + q_2) \\ &= L(A, M) - \dim(\text{rad } A) - \dim(\text{rad } M). \end{aligned}$$

This shows that it is sufficient to have (3) with the property (4). After interchanging some u_ρ with v_ρ, u_1, \dots, u_r , separate the points of $\text{rad } A \times 0$.

Otherwise Lemma 1 with $P = 0, X' = 0, X = \text{rad } A, Y = 0$ yields $a \in \text{rad } A, a \neq 0$ with $aM = 0$, a contradiction (since M is faithful). Since $\text{rad } A$ is nilpotent there exists k with $(\text{rad } A)^k = 0$. By inverse induction for $i, 1 \leq i \leq k$, we show that after interchanging some u_ρ with v_ρ, u_1, \dots, u_r , also separate the points of $\text{rad } A \times (\text{rad } A)^i M$, in particular the points of $\text{rad } A \times \text{rad } M$.

The case $i = k$ was shown above.

Assuming that we cannot conclude case $i - 1$ from case i , Lemma 1 with $P = 0, X = \text{rad } A, Y' = (\text{rad } A)^i M, Y = (\text{rad } A)^{i-1} M$ yields $x \in (\text{rad } A)^{i-1} M \setminus (\text{rad } A)^i M$, such that

$$Ax \subset \text{rad } A \cdot (\text{rad } A)^{i-1} M = (\text{rad } A)^i M.$$

Then $x \in (\text{rad } A)^i M$, a contradiction.

For the proof of Theorem 2, we will need the following lemma.

LEMMA 2. *Let K be a finite-dimensional k -division algebra, $\dim_k K = \lambda, w_1, \dots, w_{\lambda n}$ a k -base of the K -module K^n . Then setting $t = n(\lambda - 1)$, there exist $(i_1, \dots, (i_t)) 1 \leq i_1 < \dots < i_t \leq \lambda n$ such that $P = kw_{i_1} + \dots + kw_{i_t}$ does not contain a one-dimensional K -subspace.*

Proof. By induction on $l \leq t$ we show: There are i_1, \dots, i_l such that $1 \leq i_1 < \dots < i_l \leq \lambda n$ and that $P = kw_{i_1} + \dots + kw_{i_l}$ does not contain a one-dimensional K -subspace.

The case $l = 1$ is trivial.

For the induction step, we can assume w.l.o.g. that $P = kw_1 + \dots + kw_{l-1}$ does not contain a one-dimensional K -subspace. It suffices to show that there exists $j \geq l$ such that $P + kw_j$ does not contain a one-dimensional K -subspace. Assuming that for all $j \geq l, P + kw_j$ contains a one-dimensional K -subspace g_j we obtain, that the sum $g_l + \dots + g_{\lambda n}$ cannot be direct (since $\lambda n - l + 1 \geq \lambda n - t + 1 = n + 1$ and $\dim_k g_j = \lambda$). Thus w.l.o.g.

$$g_{\lambda n} \cap (g_l + \dots + g_{\lambda n-1}) \neq 0,$$

hence

$$g_{\lambda n} \subset g_l + \dots + g_{\lambda n-1}$$

and therefore

$$g_{\lambda n} \subset (P + kw_{\lambda n}) \cap (P + kw_l + \dots + kw_{\lambda n-1}) = P.$$

This means P contains a one-dimensional K -subspace, a contradiction.

Proof of Theorem 2. Let $L(K^{n \times n} \times B, K^{n \times m} \oplus N) = r$. Then there are $u_\rho, v_\rho \in ((K^{n \times n} \times B) \times (K^{n \times m} \oplus N))^*$, $w_\rho \in K^{n \times m} \oplus N$ such that

$$\forall (a, b) \in K^{n \times m} \times B, (x, y) \in K^{n \times m} \oplus N,$$

$$(ax, by) = \sum_{\rho=1}^r u_\rho(a, b; x, y) v_\rho(a, b; x, y) w_\rho.$$

In the following $K^{n \times n}$ -submodules of $K^{n \times m}$ are called left modules and $\text{End}_{K^{n \times n}} K^{n \times m}$ -submodules of $K^{n \times m}$ are called right modules.

Case $m = 1$. Associating with each $\mu \in K^{op}$ the right-multiplication in K^n with μ as endomorphism we get (because of $m = 1$ and $K^{n \times m} = K^n$)

$$\text{End}_{K^{n \times n}} K^n \simeq K^{op}.$$

Furthermore w_1, \dots, w_r generate $K^n \oplus N$, hence $r \geq \lambda n$ and w.l.o.g. we can assume that $w_1, \dots, w_{\lambda n}$ are k -linearly independent and that, taking

$$W = kw_1 + \dots + kw_{\lambda n},$$

we have $W \cap N = 0$. ($w_1, \dots, w_{\lambda n}$ are to be chosen such that the projections on K^n are k -linearly independent.)

In particular, we have

$$K^n \oplus N = W \oplus N.$$

Let $w_\rho = w'_\rho \oplus w''_\rho$, $w'_\rho \in K^n$, $w''_\rho \in N$ ($1 \leq \rho \leq r$). $w'_1, \dots, w'_{\lambda n}$ are a k -base of K^n . Let $t = (\lambda - 1)n$. By Lemma 2 (applied on $\text{End}_{K^{n \times n}} K^n = K^{op}$), we may assume w.l.o.g. that

$$P' = kw'_1 + \dots + kw'_t \subset K^n$$

does not contain a one-dimensional $\text{End}_{K^{n \times n}} K^n$ -subspace, and thus no right submodule $\neq 0$ at all.

Let $P = kw_1 + \dots + kw_r$.

We claim: After interchanging some u_ρ with v_ρ the u_ρ 's with $\rho > t$ separate the points of $(K^{n \times n} \times 0) \times 0$. (In particular $r \geq \lambda n^2 + t = \lambda n^2 + (\lambda - 1)n$.) Otherwise Lemma 1 with P as above, $X' = 0 \times 0$, $X = K^{n \times n} \times 0$, $Y = 0$ yields $a \in K^{n \times n}$, $a \neq 0$ such that

$$(a, 0) \cdot (K^n \oplus N) = ak^n \subset P,$$

hence $aK^n \subset P'$.

aK^n is a right submodule of K^n , therefore

$$aK^n = 0,$$

i.e. a is an element of the annihilator of the $K^{n \times n}$ -module K^n , a contradiction.

Therefore w.l.o.g. we can assume that the restrictions of $u_{t+1}, \dots, u_{t+\lambda n^2}$ are a base of $((K^{n \times n} \times 0) \times 0)^*$. Thus if $b \in B, y \in N$ are arbitrary, we find a uniquely determined $a \in K^{n \times n}$ such that

$$(5) \quad \forall \rho (t + 1 \leq \rho \leq t + \lambda n^2) \quad u_\rho(a, 0; 0, 0) = -u_\rho(0, b; 0, y).$$

Therefore

$$u_{t+1}(a, b; 0, y) = \dots = u_{t+\lambda n^2}(a, b; 0, y) = 0,$$

thus

$$a \cdot 0 \oplus b \cdot y = b \cdot y = \sum_{\rho=1}^t u_{\rho}(a, b; 0, y)v_{\rho}(a, b; 0, y)w_{\rho} + \sum_{\rho=t+\lambda n^2+1}^r u_{\rho}(a, b; 0, y)v_{\rho}(a, b; 0, y)w_{\rho}.$$

Denoting by $\sigma: W \oplus N \rightarrow N$ the projection along W we get

$$b \cdot y = \sum_{\rho=t+\lambda n^2+1}^r u_{\rho}(a, b; 0, y)v_{\rho}(a, b; 0, y)\sigma(w_{\rho}).$$

(5) describes a system of linear equations for a . Therefore we can consider u_{ρ} and v_{ρ} in the above representation as linear forms on $B \times N$. Finally we have

$$L(B, N) \leq r - (t + \lambda n^2).$$

Case $n = 1$. As in the preceding case, we can assume w.l.o.g. that $w_1, \dots, w_{\lambda m}$ are k -linearly independent and that, taking

$$W = kw_1 + \dots + kw_{\lambda m}$$

we have

$$W \cap N = 0,$$

thus

$$K^m \oplus N = W \oplus N.$$

Let $t = (\lambda - 1)m$. Proceeding as in the case $m = 1$, we can assume this time w.l.o.g. that

$$P' = kw'_1 + \dots + kw'_t \subset K^m$$

contains no left submodule $\neq 0$.

Let $P = kw_1 + \dots + kw_t$.

We claim: After interchanging some u_{ρ} with v_{ρ} the u_{ρ} 's with $\rho > t$ separate the points of $0 \times K^m$. (In particular $r \geq \lambda m + t = (2\lambda - 1)m$.)

Otherwise Lemma 1 yields $x \in K^m, x \neq 0$ such that

$$(K \times B)(x, 0) = Kx \subset P,$$

hence

$$Kx \subset P'.$$

Kx is a left submodule of K^m , thus

$$Kx = 0,$$

that means $x = 0$, a contradiction.

Therefore w.l.o.g. we can assume that the restrictions of $u_{t+1}, \dots, u_{t+\lambda m}$ are a base of $(0 \times K^m)^*$.

If $b \in B, y \in N$ are arbitrary we get, proceeding as in the case $m = 1$, a uniquely determined $x \in K^m$ such that

$$u_{t+1}(0, b; x, y) = \dots = u_{t+\lambda m}(0, b; x, y) = 0.$$

Denoting the projection along W by $\sigma: W \oplus N \rightarrow N$, we get proceeding as in the case $m = 1$

$$b \cdot y = \sum_{\rho=t+\lambda m+1}^r u_\rho(0, b; x, y) v_\rho(0, b; x, y) \sigma(w_\rho),$$

and therefore

$$L(B, N) \leq r - (t + \lambda m).$$

Case $n \geq m > 1$.

$$E = \begin{bmatrix} K & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ K & 0 & \cdots & 0 \end{bmatrix}$$

is a simple left submodule of $K^{n \times m}$. Let

$$t = \text{codim}_k E - 1 = \lambda n(m - 1) - 1.$$

Again we can assume w.l.o.g. that

$$w_1, \dots, w_{t+1} \text{ are } k\text{-linearly independent}$$

and that taking

$$W = kw_1 + \dots + kw_{t+1}$$

we have

$$W \cap (E \oplus N) = 0,$$

in particular for $P = kw_1 + \dots + kw_t$

$$(6) \quad P \cap (E \oplus N) = 0.$$

$P + (E \oplus N)$ is a hyperplane in $K^{n \times m} \oplus N$. Therefore

$$R = \{a \in K^{n \times n} \mid (a, 1)(K^{n \times m} \oplus N) \subset P + (E \oplus N)\}$$

is a proper right ideal of $K^{n \times n}$.

In three steps we try to achieve, after interchanging in each step some u_ρ with v_ρ , the following statement:

$$(7) \quad u_\rho \text{ with } \rho > t \text{ separate the points of } (K^{n \times n} \times 0) \times E.$$

Step 1. u_ρ with $\rho > t$ separate the points of $(R \times 0) \times E$. Otherwise Lemma 1 with above $P, X' = 0 \times 0, X = R \times 0, Y = 0$ yields $a \in R, a \neq 0$ such that

$$(a, 0) \cdot (K^{n \times m} \oplus N) \subset P$$

thus

$$aK^{n \times m} \subset P.$$

$aK^{n \times m}$ is a right module. (6) implies

$$aK^{n \times m} \cap E = 0,$$

hence $aK^{n \times m} = 0$ and $a = 0$, a contradiction.

Step 2. u_ρ with $\rho > t$ separate the points of $(R \times 0) \times E$. Otherwise Lemma 1 with $X = R \times 0, Y' = 0, Y = E$ yields $x \in E, x \neq 0$ such that

$$(K^{n \times n} \times B) \cdot x \subset P + RE.$$

But $K^{n \times n}x = E$, thus

$$E \subset P + RE.$$

$RE \subset E$ and (6) imply

$$E \subset RE,$$

a contradiction.

Step 3. u_ρ with $\rho > t$ separate the points of $(K^{n \times n} \times 0) \times E$. Otherwise Lemma 1 with $X' = R \times 0$, $X = K^{n \times n} \times 0$, $Y = E$ yields $a \in K^{n \times n} \setminus R$ such that

$$(a, 0) \cdot (K^{n \times m} \oplus N) \subset P + E,$$

hence $a \in R$ by definition of R , a contradiction. Thus we have proved statement (7) and we can assume w.l.o.g. that the restrictions of

$$u_{t+1}, \dots, u_t \quad (t' = t + \lambda n^2 + \lambda n)$$

are a base of $((K^{n \times n} \times 0) \times E)^*$.

Corresponding to the preceding cases, we find for arbitrary $b \in B$, $y \in N$ uniquely determined $a \in K^{n \times n}$, $e \in E$ such that

$$u_{t+1}(a, b; e, y) = \dots = u_{t'}(a, b; e, y) = 0,$$

thus

$$\begin{aligned} a \cdot e \oplus b \cdot y &= \sum_{\rho=1}^t u_\rho(a, b; e, y) v_\rho(a, b; e, y) w_\rho \\ &+ \sum_{\rho=t'+1}^r u_\rho(a, b; e, y) v_\rho(a, b; e, y) w_\rho. \end{aligned}$$

Denoting the projection along $W \oplus E$ by $\sigma: W \oplus E \oplus N \rightarrow N$, we get, since $a \cdot e \in E$,

$$b \cdot y = \sum_{\rho=t'+1}^r u_\rho(a, b; e, y) v_\rho(a, b; e, y) \sigma(w_\rho).$$

Again we can regard u_ρ and v_ρ in above representation as linear forms on $B \times N$.

Finally we obtain

$$L(B, N) \leq r - t' = r - \lambda n^2 - \lambda nm + 1.$$

Case $m \geq n > 1$.

$$F = \begin{bmatrix} K & \dots & K \\ 0 & \dots & 0 \\ \vdots & & \vdots \\ 0 & \dots & 0 \end{bmatrix}$$

is a simple right submodule of $K^{n \times m}$.

For the proof, we proceed as in the case $n \geq m > 1$.

Let $t = \text{codim}_k F - 1 = \lambda(n - 1)m - 1$. W.l.o.g. we can assume that

$$w_1, \dots, w_{t+1} \text{ are } k\text{-linearly independent}$$

and that taking

$$W = kw_1 + \dots + kw_{t+1}$$

we have

$$W \cap (F \oplus N) = 0.$$

In particular taking $P = kw_1 + \dots + kw_t$ we get

$$(8) \quad P \cap (F \oplus N) = 0.$$

$P + (F \oplus N)$ is a hyperplane in $K^{n \times m} \oplus N$. Therefore

$$L = \{x \in K^{n \times m} \mid (K^{n \times n} \times B) \cdot x \subset P + (F \oplus N)\}$$

is a proper left module in $K^{n \times m}$.

Let

$$I = \begin{bmatrix} K & \dots & K \\ 0 & \dots & 0 \\ \vdots & & \vdots \\ 0 & \dots & 0 \end{bmatrix} \subset K^{n \times n}.$$

In three steps we will show that, after interchanging some u_ρ with v_ρ , the following holds:

$$(9) \quad u_\rho \text{ with } \rho > t \text{ separate the points of } (I \times 0) \times K^{n \times m}.$$

Step 1. u_ρ with $\rho > t$ separate the points of $0 \times L$. Otherwise Lemma 1 yields $x \in L$, $x \neq 0$ such that

$$(K^{n \times n} \times B) \cdot x \subset P.$$

$K^{n \times n} \cdot x$ is a left module.

Equation (8) implies

$$K^{n \times n} \cdot x \cap F = 0,$$

hence $K^{n \times n} \cdot x = 0$, a contradiction.

Step 2. u_ρ with $\rho > t$ separate the points of $(I \times 0) \times L$. Otherwise Lemma 1 yields $a \in I$, $a \neq 0$ such that

$$(a, 0) \cdot (K^{n \times m} \oplus N) \subset P + IL.$$

But $aK^{n \times m} = F$, thus

$$F \subset P + IL.$$

$IL \subset F$ and (8) imply

$$F \subset IL.$$

Therefore,

$$F \subset L,$$

which is impossible because $L \subset K^{n \times m}$ is a proper left module.

Step 3. u_ρ with $\rho > t$ separate the points of $(I \times 0) \times K^{n \times m}$. Otherwise Lemma 1 yields $x \in K^{n \times m} \setminus L$ such that

$$(K^{n \times n} \times B) \cdot x \subset P + IK^{n \times m}.$$

Since $IK^{n \times m} = F$ we have

$$(K^{n \times n} \times B) \cdot x \subset P + F,$$

thus $x \in L$ by definition of L , a contradiction. Now we proceed as in the case $n \geq m > 1$

and finally obtain

$$\begin{aligned}L(B, N) &\leq r - t - \lambda n - \lambda nm \\ &= r - 2\lambda nm + \lambda(m - n) + 1.\end{aligned}$$

Acknowledgment. I am deeply indebted to Volker Strassen, whose contribution to the realization of the present work has been considerable. I should like to thank him very much for his advice and his constant encouragement.

REFERENCES

- [1] A. ALDER AND V. STRASSEN, *On the algorithmic complexity of associative algebras*, Theoret. Comput. Sci., 15 (1981), pp. 201-211.
- [2] L. AUSLANDER AND S. WINOGRAD, *The multiplicative complexity of certain semi-linear systems defined by polynomials*, Adv. Appl. Math., 1 (1980), pp. 257-299.
- [3] R. S. PIERCE, *Associative Algebras*, Springer-Verlag, New York, 1982.
- [4] V. STRASSEN, *Vermeidung von Divisionen*, J. Reine Angew. Math., 264 (1973), pp. 184-202.
- [5] S. WINOGRAD, *A new algorithm for inner products*, IEEE Trans. Comp., 17 (1968), pp. 693-694.

PROBABILISTIC PARALLEL ALGORITHMS FOR SORTING AND SELECTION*

RÜDIGER REISCHUK†

Abstract. Probabilistic parallel algorithms are described to sort n keys and to select the k -smallest element among them. For each problem we construct a probabilistic parallel decision tree. The tree for selection finishes with high probability in constant time and the sorting tree in time $O(\log n)$. The same time bound for sorting can also be achieved by a probabilistic parallel machine consisting of n RAMs, each with small private memory, and a common memory of size $O(n)$. These algorithms meet the information theoretic lower bounds.

Key words. parallel algorithms, probabilistic algorithms, sorting, selection, parallel random access machines, decision trees, efficient algorithms

1. Introduction and known results. This paper deals with parallel algorithms for comparison problems. The input is always a set $X = \{x_1, \dots, x_n\}$ of keys belonging to a linear ordered set of arbitrary size and the only operations involving keys are comparisons of a pair and store- and fetch-operations. Depending on the machine model the *sorting* problem is either to determine a permutation Π of $\{1, \dots, n\}$ such that $x_{\pi(1)} \leq \dots \leq x_{\pi(n)}$ or to write the sorted sequence in n output registers. $x_{\pi(k)}$ is called the k -smallest element of X or the element in X of rank k . The *selection* problem with additional input k demands either to determine $\pi(k)$ or to give out $x_{\pi(k)}$.

In the sequential case $O(n)$ algorithms are known for selection and $O(n \log n)$ algorithms for sorting, which meet (up to constant factors) the trivial lower bounds for these problems. Obviously these lower bounds also hold for parallel algorithms for the product of time and number of processors, but up to now no parallel algorithm is known to the author that achieves these bounds.

With respect to lower bounds the most general model is the *decision tree*. A decision tree algorithm for inputs of size n is specified by a binary tree where each internal node is labelled with a pair (i, j) of $\{1, \dots, n\}$. A run of the algorithm on input X is a path from the root to a leaf. If the algorithm reaches a node u with label (i, j) the comparison between x_i and x_j is performed and the left (resp. right) branch from u is taken if $x_i \leq x_j$ (resp. $x_i > x_j$). An output is assigned to each leaf. We say that a decision tree algorithm solves a given problem if for each possible input one reaches a leaf with the correct output. The time-complexity of a decision tree algorithm is defined as the depth of the corresponding tree.

If $p > 1$ comparisons can be performed at a time we get a *parallel decision tree of order p* . Here each node is labelled with $q \leq p$ elements of $\{1, \dots, n\}^2$ and a node has 2^q sons, one for each of the possible outcomes of the corresponding comparisons. The q simultaneous comparisons may involve a key more than once.

A *probabilistic parallel decision tree of order p* is a tree with two different kinds of internal nodes which alternate on every path from the root to a leaf. Each choice node has $\sum_{q=1}^p \binom{n}{q}$ sons, corresponding to the subsets of $\{1, \dots, n\}^2$ of size at most p , and a probability distribution assigned to its set of outedges. Reaching a choice node a son is chosen at random according to the given probability distribution. At the new node, which is a comparison node, the corresponding comparisons between the input

* Received by the editors January 29, 1982, and in revised form March 1, 1984. Research supported by DFG-grant Pa 248/1. A preliminary report, *A Fast Probabilistic Parallel Sorting Algorithm*, was presented at the 22nd IEEE Conference, Nashville, 1981.

† Universität Bielefeld, Fakultät für Mathematik, 4800 Bielefeld 1, West Germany.

elements are made and one advances to one of its sons which is determined by the outcome of these comparisons. For a given input X the average time-complexity is defined as the average length of paths from the root to leaves. We say that the decision tree algorithm runs in time T with probability α if for each input the probability of paths from the root to leaves of length greater than T is at most $1 - \alpha$.

Saying that out of a set Y an element y is chosen *uniformly at random* means that equal probability is assigned to each element of Y . In [5] Valiant proves an upper and lower bound $n/p + \log \log p + O(1)$ for finding the maximum of n elements by a deterministic parallel decision tree of order p . For $p = n$ this gives a $\log \log n$ lower bound. Hence for this problem the optimal speed-up by making n comparisons at a time instead of 1 is only $n/\log \log n$. One cannot achieve the general bound n . [5] also describes a parallel decision tree of order n to sort n keys in time $O(\log n \log \log n)$, which again gives a speed-up of only $O(n/\log \log n)$.

Reference [3] considers the problem how many comparisons have to be performed in parallel by a parallel decision tree that sorts in constant time $d \geq 1$. A lower bound $\Omega(n^{1+1/d})$ and an upper bound $O(n^{\alpha_d} \log n)$ is shown where $\alpha_d = (3 \cdot 2^{d-1} - 1)/(2^d - 1)$. Also results on merging are mentioned.

Here we are only interested in algorithms where the number of simultaneous comparisons does not exceed the size of the input by much, because otherwise a good speed-up is obviously impossible.

For a more realistic model of computation Preparata [4] obtains a parallel sorting algorithm which sorts n keys using $n \log n$ processors in $O(\log n)$ steps. His model is a single-instruction, multiple-data stream computer (SIMD) with random access capabilities to a common memory. Simultaneous reading in the same storage location is allowed, but no writing. Each processor performs arithmetic operations $+$, $-$, $*$, $/$, $\lfloor \cdot \rfloor$, $\lceil \cdot \rceil$, $\lceil \log \cdot \rceil$ as well as the comparison of two keys in unit time. The algorithm recursively sorts smaller subsets and then merges pairs of ordered sequences of size m in $O(\log \log m)$ steps together to determine the rank of each element. The above time bound includes the arithmetical computations, the number of comparisons and the number of read/write operations in the common memory, but does not take into consideration the problem of processor assignment when merging the subsets together. When the assignment is done in an elementary way and the number of these steps are also counted one needs additional $O(\log m)$ steps when merging sequences of size m , which gives a total time bound $O(\log^2 n/\log \log n)$. For this algorithm the speed-up for $N = n \log n$ processors is $O(N/\log N)$, resp. $O(N \log \log N/\log^2 N)$.

For the same computation model [4] describes another algorithm which uses $n^{1+\alpha}$ processors and runs in time $O((1/\alpha) \log n)$ for any $0 < \alpha \leq 1$, which gives a speed-up of $O(N^{1/(1+\alpha)})$, $N = n^{1+\alpha}$. The assignment of processors is easier in this algorithm and different processors do not read in a storage location at the same time. Valiant's results show that for worst-case time complexity not every problem can be solved n times faster if we use n instead of 1 processor. He also conjectures that the attainable speed-up for the selection problem is still smaller. The question arises whether a speed-up of order n can be achieved at least on the average, whether probabilistic steps can improve parallel algorithms substantially. Obviously for selection or sorting adding probabilism cannot help much in the sequential case. In the following we will show that for both problems a speed-up of order n can be achieved by probabilistic parallel algorithms that use n processors.

We assume that the keys are pairwise distinct. This is no restriction since if we are given n arbitrary keys x_1, \dots, x_n replace x_i by (x_i, i) and define an order of the tuples by $(x_i, i) < (x_j, j)$ iff $x_i < x_j$ or $x_i = x_j$ and $i < j$.

2. A probabilistic parallel decision tree for selection. We first consider the selection problem and describe a probabilistic algorithm that almost always runs in constant time. This contrasts to the $\log \log n$ lower bound [5] for finding the maximum in the deterministic case.

THEOREM 1. *For any n and $1 \leq k \leq n$ there is a probabilistic parallel decision tree algorithm of order n that selects the k -smallest element of a set of n elements and finishes for any input with probability greater than $1 - \exp(-\frac{1}{4}n^{3/16} + O(\ln n))$ in c_1 steps for some constant c_1 .*

Proof. In [2] a probabilistic sequential selection algorithm of small average runtime is described. Using some of those ideas we will construct a parallel algorithm PPSELECT and prove that it has the properties stated above. We start with an informal description of the algorithm.

Let $X = \{x_1, \dots, x_n\}$ be the given set and k the rank of the element to be selected. When calling PPSELECT recursively two variables are used: Y , which denotes a subset of X of size $m \leq n$, and a number l , $1 \leq l \leq m$, the rank of that element in Y which has to be selected. Thus PPSELECT(X, k) will do the job. For technical reason we associate to a set of size s element $-\infty$ (resp. element $+\infty$) of rank 0 (resp. $s+1$), which are smaller (resp. bigger) than every element in the given set.

Procedure PPSELECT(Y, l)

if $|Y| \leq \sqrt{2n}$ **compare** in parallel every pair of elements in Y (since $\binom{|Y|}{2} \leq n$, this can be done in one step)

if $|Y| > \sqrt{2n}$ **then do**

1. let m denote the size of Y , $s = \lfloor \sqrt{m} \rfloor$ and $f = f(m)$ be some number which will be specified later,
choose uniformly at random a subset S of Y of size s
2. **compare** in parallel every pair of elements in S (this determines for each $s \in S$ its rank in S); define

$$t_1 = \max \left\{ 0, \left\lfloor l \frac{s+1}{m+1} - f \right\rfloor \right\}$$

$$t_2 = \min \left\{ s+1, \left\lceil l \frac{s+1}{m+1} + f \right\rceil \right\}$$

let s_i ($i=1, 2$) denote the element of rank t_i in S

3. **compare** in parallel every element of Y with s_1
4. **compare** in parallel every element of Y with s_2
5. define

$$A = \{y \in Y \mid y < s_1\}$$

$$B = \{y \in Y \mid s_1 \leq y \leq s_2\}$$

$$C = \{y \in Y \mid s_2 < y\}$$

if $|A| < l$ and $|A| + |B| \geq l$ **then call** PPSELECT($B, l - |A|$)

if $|A| \geq l$ **then call** PPSELECT(A, l)

if $|A| + |B| < l$ **then call** PPSELECT($C, l - |A| - |B|$)

end

end procedure

Since the correctness of PPSELECT can easily be checked we will only prove that it terminates with high probability after some number of steps independent of the input.

It obviously suffices to show that starting with X of size n with high probability after a constant number of recursive calls a set of size at most $\sqrt{2n}$ is generated, which will be the input parameter for the next call.

Let n be large enough for the following analysis and let Y of size m with $\sqrt{2n} < m \leq n$ and $1 \leq l \leq m$ be inputs of PPSELECT. We will only deal with the case $t_1 \geq 1$ and $t_2 \leq s$. The easier cases ($t_1 = 0$ or $t_2 = s + 1$) can be handled similarly. Let R_i ($i = 1, 2$) be random variables denoting the rank of s_i in Y .

LEMMA 1. For any $f \in \mathbb{R}$,

$$P\left(R_i = \frac{m+1}{s+1}(t_i + f)\right) \leq \exp\left(-\frac{1}{4} \frac{f^2}{s+1} + O(\ln m)\right).$$

Proof. For $f = 0$ the claim is obvious; therefore assume $f \neq 0$. Let $r = (t_i + f)(m + 1)/(s + 1)$. For $r \in \mathbb{N}$, $1 \leq r \leq m$ holds

$$(2.0) \quad P(R_i = r) = \frac{\binom{r-1}{t_i-1} \binom{m-r}{s-t_i}}{\binom{m}{s}} = \frac{t_i(s+1-t_i)(m+1)}{r(m+1-r)(s+1)} \frac{\binom{r}{t_i} \binom{m+1-r}{s+1-t_i}}{\binom{m+1}{s+1}}.$$

If $-t_i \geq f$ then $r \leq 0$ and if $f \geq s + 1 - t_i$ then $r \geq m + 1$; hence in both cases $P(R_i = r) = 0$. Thus we may assume $-t_i < f < s + 1 - t_i$. Using

$$\binom{N}{K} = \exp\left(K \ln \frac{N}{K} + (N - K) \ln \frac{N}{N - K} + O(\ln N)\right)$$

one can transform (2.0) into

$$(2.1) \quad -\ln(P(R_i = r)) = t_i \ln\left(\frac{m+1}{s+1} \frac{t_i}{r}\right) + (s+1-t_i) \ln\left(\frac{m+1}{s+1} \frac{s+1-t_i}{m+1-r}\right) + O(\ln m).$$

For $0 < p < 1$ and $-p < x < 1 - p$ define

$$H_p(x) = p \cdot \ln \frac{p}{p+x} + (1-p) \ln \frac{1-p}{1-p-x}.$$

If in (2.1) r is replaced by $(t_i + f)(m + 1)/(s + 1)$ the main terms on the right side become

$$(2.2) \quad \begin{aligned} & t_i \ln\left(\frac{m+1}{s+1} \frac{t_i}{r}\right) + (s+1-t_i) \ln\left(\frac{m+1}{s+1} \frac{s+1-t_i}{m+1-r}\right) \\ &= t_i \ln \frac{t_i}{t_i+f} + (s+1-t_i) \ln \frac{s+1-t_i}{s+1-t_i-f} \\ &= (s+1)H_{t_i/(s+1)}\left(\frac{f}{s+1}\right) \quad \text{with } 0 < \frac{t_i}{s+1} < 1 \text{ and } \frac{-t_i}{s+1} < \frac{f}{s+1} < \frac{s+1-t_i}{s+1}. \end{aligned}$$

Differentiating H_p gives

$$(2.3) \quad \begin{aligned} H'_p(x) &= \frac{x}{(p+x)(1-p-x)}, \\ H''_p(x) &= \frac{p-p^2+x^2}{(p+x)^2(1-p-x)^2}. \end{aligned}$$

For $0 < p < 1$ and $-p < x < 1 - p$ follows

$$(2.4) \quad H_p(x) \geq 0, \quad H''_p(x) > 0.$$

$H_p(x)$ can be developed into a Taylor series at the point $x/2$, which gives

$$H_p(x) = H_p\left(\frac{x}{2}\right) + \frac{x}{2} \cdot H'_p\left(\frac{x}{2}\right) + \left(\frac{x}{2}\right)^2 \frac{H''_p(z)}{2}$$

for some z between $x/2$ and x . From (2.3) we get the bound

$$(2.5) \quad H_p(x) \cong \frac{x}{2} \cdot H'_p\left(\frac{x}{2}\right) = \frac{1}{4} \frac{x^2}{(p+x/2)(1-p-x/2)}.$$

Hence

$$\begin{aligned} (s+1)H_{t_i/(s+1)}\left(\frac{f}{s+1}\right) &\cong \frac{s+1}{4} \frac{f^2}{(t_i+f/2)(s+1-t_i-f/2)} \\ &\cong \frac{1}{4} \frac{f^2}{(t_i+f/2)} \quad \text{since } -t_i - \frac{f}{2} \cong 0, \\ \text{or} \\ &\cong \frac{1}{4} \frac{f^2}{s+1} \quad \text{since } t_i + \frac{f}{2} \cong s+1. \end{aligned}$$

From (2.2) and (2.1) now follows

$$P\left(R_i = \frac{m+1}{s+1}(t_i+f)\right) \leq \exp\left(-\frac{1}{4} \frac{f^2}{s+1} + O(\ln m)\right). \quad \square$$

COROLLARY. *Let f be a positive real number. Then:*

- i) $P\left(R_i \cong \frac{m+1}{s+1}(t_i+f)\right) \leq \exp\left(-\frac{1}{4} \frac{f^2}{s+1} + O(\ln m)\right)$
- ii) $P\left(R_i \leq \frac{m+1}{s+1}(t_i-f)\right) \leq \exp\left(-\frac{1}{4} \frac{f^2}{s+1} + O(\ln m)\right).$

The proof follows from Lemma 1 and the fact that there are at most m values r with $P(R_i = r) > 0$.

Let E_0 be the event that for a run of PPSELECT on input (Y, l) the element of rank l in Y does not belong to set B which is generated in step 5, and E_1 be the event that the size of B exceeds $4(f+1)(m+1)/(s+1)$.

LEMMA 2.

- i) $P(E_0) \leq \exp\left(-\frac{1}{4} \frac{f^2}{s+1} + O(\ln m)\right).$
- ii) $P(E_1) \leq \exp\left(-\frac{1}{4} \frac{f^2}{s+1} + O(\ln m)\right).$

Proof. The event E_0 implies the event $[(R_1 > l) \text{ or } (R_2 < l)]$. Since

$$\frac{m+1}{s+1}(t_1+f) = \frac{m+1}{s+1} \left[l \cdot \frac{s+1}{m+1} - f \right] + \frac{m+1}{s+1} f \cong l$$

and

$$\frac{m+1}{s+1}(t_2-f) = \frac{m+1}{s+1} \left[l \cdot \frac{s+1}{m+1} + f \right] - \frac{m+1}{s+1} f \cong l,$$

$R_1 > l$ implies $R_1 > (t_1+f)(m+1)/(s+1)$ and $R_2 < l$ implies $R_2 < (t_2-f)(m+1)/(s+1)$.

Similarly, if the size of B exceeds $4(f+1)(m+1)/(s+1)$ then

$$R_1 < l - 2(f+1) \frac{m+1}{s+1} \quad \text{or} \quad R_2 > l + 2(f+1) \frac{m+1}{s+1}.$$

This implies

$$R_1 < \frac{m+1}{s+1}(t_1 - f) \quad \text{or} \quad R_2 > \frac{m+1}{s+1}(t_2 + f).$$

Claims i) and ii) then follow from the corollary. \square

Now let $f = f(m)$ be increasing with m . Hence with probability at most $\alpha(m) := \exp(-\frac{1}{4}f(m)^2/(s+1) + O(\ln m))$ the first recursive call within PPSELECT (Y, l) uses inputs (Y', l') with $|Y'| > 4(f(m)+1)(m+1)/(s+1) := \beta(m)$.

If we choose $f(m) = m^{7/16}$ then for all large m $\beta(m)$ can be bounded by $5m^{15/16}$. Thus if $\sigma(i)$ for $i \geq 1$ denotes the size of the current input parameter Y in the i th recursive call of PPSELECT on input (X, k) then

$$\sigma(1) = n$$

and

$$\sigma(i+1) > 5 \sigma(i)^{15/16} \quad \text{with probability at most } \alpha(\sigma(i)).$$

Since $(\frac{15}{16})^{11} < \frac{1}{2}$, for all large n $\sigma(12) > n^{1/2}$ with probability at most

$$\max_{\sigma(1) \geq \dots \geq \sigma(11) > n^{1/2}} \sum_{i=1}^{11} \alpha(\sigma(i)) \leq 11 \alpha(n^{1/2}) = \exp(-\frac{1}{4}n^{3/16} + O(\ln n)). \quad \square$$

3. Sorting by a parallel decision tree. We now want to show that a probabilistic selection algorithm running in constant time implies a $O(\log n)$ probabilistic sorting algorithm. Unlike the sequential case this is not that easy for parallel algorithms, since expected runtime is given by the expected maximum runtime over all components working in parallel.

THEOREM 2. *For $n \in \mathbb{N}$ there exists a probabilistic parallel decision tree of order n that sorts n keys and for any input does not use more than $c_2 \log n$ steps for some constant c_2 with probability at least $1 - \exp(-\Omega((\ln n)^{19/16}))$.*

Proof. Let X of size n be the set which has to be sorted. We may assume that $n = 2^u$ and $u = 2^v$ with $u, v \in \mathbb{N}$. The decision tree uses a divide and conquer strategy until the problem size is reduced to $u = \log n$. Using PPSELECT as a subroutine the element x_{med} of rank $n/2$ in X is determined. Then, by comparing in parallel every element of X with x_{med} X can be split into sets X_0 and X_1 of those elements which are smaller or equal (resp. greater) than x_{med} . Call these the *lower* (resp. *upper*) half of X . Both halves are then sorted recursively in parallel each by a parallel decision tree of order $n/2$. Subsets of size u , call them *small*, are sorted by a deterministic parallel decision tree of order u by comparing successively each element with every other element in parallel. This takes parallel time u .

To analyze the runtime consider the binary tree T of subsets of X which are generated in the course of the algorithm. The root of T is X and leaves are the small subsets of X . Sons of a set are its lower and upper half. T has depth $\tau := u - v$.

To show that the algorithm sorts in $O(u)$ steps it suffices to show that on every path Q from the root to a leaf the time spent for finding the medians of the corresponding subsets, let us denote it by $M(Q)$, is bounded by $O(u)$.

Let $Q = Y_1, Y_2, \dots, Y_\tau$ and $g(i) = 5c_1 \lceil u/(\tau - i)^2 \rceil$ for $1 \leq i < \tau$ where c_1 is the constant from Theorem 1. If the median of every Y_i , $i < \tau$ is found in at most $g(i)$

steps then $M(Q)$ is bounded by

$$\sum_{i=1}^{\tau-1} g(i) = \sum_i 5c_1 \left[\frac{u}{(\tau-i)^2} \right] = O(u).$$

From the preceding theorem it follows that the probability $P(y, g)$ that the median of a set of size y is not found in $g = \alpha c_1$ steps, where $\alpha \in \mathbb{N}$ is bounded by

$$[\exp(-\frac{1}{4}y^{3/16} + O(\ln y))]^\alpha.$$

For y large enough this gives $P(y, g) \leq \exp(-(\alpha/5)y^{3/16})$.

Hence for some Y_i the median is not found in $g(i)$ steps with probability at most

$$\begin{aligned} \sum_{i=1}^{\tau-1} P(2^{u-i}, g(i)) &\leq \sum_i \exp\left(-\left[\frac{u}{(\tau-i)^2}\right] 2^{3(u-i)/16}\right) \\ &\leq \sum_i \exp\left(-u 2^{3v/16} \cdot \left[\frac{1}{i^2}\right] 2^{3i/16}\right) \\ &\leq \tau \exp\left(-u^{1+3/16} \min_{1 \leq i < \tau} \left\{ \left[\frac{1}{i^2}\right] 2^{3i/16} \right\}\right) \\ &= \tau \exp(-\gamma u^{19/16}) \quad \text{for some constant } \gamma > 0. \end{aligned}$$

Therefore the probability that for each of the 2^τ paths Q from the root to a leave $M(Q)$ is bounded by $O(u)$ is at least

$$1 - 2^\tau \tau \exp(-\gamma u^{19/16}) = 1 - n \exp(-\gamma u^{19/16}) = 1 - \exp(-\Omega((\ln n)^{19/16})). \quad \square$$

4. Sorting by a parallel RAM. We now describe a probabilistic parallel sorting algorithm for a more realistic computation model than the decision tree. The parallel computer consists of n processors P_1, \dots, P_n and a common memory of size $O(n)$. Each processor is a RAM [1] with a constant number of private registers and has random access to the common memory. In the common memory different processors may read a cell simultaneously, but in each step at most one processor can write into it. The arithmetic instruction set contains $+, -, *, [/]$. Each processor needs one unit of time to perform an arithmetic operation or to compare two keys. This implies that functions like $\lfloor \sqrt{n} \rfloor$, $\lfloor \log n \rfloor$, 2^n all can be computed in $O(\log n)$ steps. The only probabilistic step a processor can perform is choosing 0 or 1 with equal probability.

Suppose we are given a set $X = \{x_1, \dots, x_n\}$ of n distinct keys which at the beginning are stored in the common memory in an array $K = K_1, \dots, K_n$, $K_i = x_i$. This sequence will be rearranged to get a partition of K into subsequences B^0, \dots, B^s called *boxes* such that for $i < j$ every element of box B^i is less than every element of B^j .

Such a partition of K is uniquely defined by the indices of the first and last element of each box. These indices are stored in arrays $L = L_1, \dots, L_n$ and $U = U_1, \dots, U_n$ in the following way: if key K_i belongs to box $B^l = K_{l+1}, \dots, K_u$ then $L_i = l+1$ and $U_i = u$. Now the global structure of the sorting algorithm is the following:

1. Select a subset Y of K of size $s = \lfloor \sqrt{n} \rfloor$ at random and sort Y by comparing every pair of keys in Y .

With the help of this ordered subset K can be partitioned into $s+1$ boxes B^0, \dots, B^s where B^i contains those keys that are bigger than the i th smallest but not bigger than the $(i+1)$ -smallest element of Y .

2. By binary insertion determine in parallel for each K_i to which box it belongs.
3. Rearrange the sequence K such that $i < j$ and $K_i \in B^r$ and $K_j \in B^t$ implies $r \leq t$.

4. Recursively in parallel for each $0 \leq j \leq s$ sort the subsequence of K containing B^j . A box of size at most $\log n$, call it *small*, is sorted deterministically by comparing each element with every other element in parallel.

While the basic structure of this algorithm is simple, the problem is to find an efficient implementation for a parallel computer of the above type and to show that with high probability each of the about \sqrt{n} boxes B^i are sorted fast.

The following notation will be used:

$$\text{in parallel } [P_i | a \leq i \leq b]: \langle \text{commands} \rangle.$$

This means that in parallel each processor P_i with index between a and b carries out the commands.

The algorithm uses two subroutines which will be described and analyzed beforehand. In the first one a key is selected from a subsequence of K almost uniformly at random and stored in an array S .

SEL ($l+1, u, S$)

comment: P_{l+1} chooses one element from K_{l+1}, \dots, K_u at random and writes it into S .

processor P_{l+1} :

1. **choose** $2 \lceil \log(u-l) \rceil$ bits at random and interpret them as a binary number N between 0 and $2^{2 \lceil \log(u-l) \rceil} - 1$.
 2. **compute** $m := N \bmod (u-l) = N - (u-l)(N / \lfloor u-l \rfloor)$
 3. **set** $S_{l+1} \leftarrow K_{l+1+m}$
- end SEL**

Procedure **SEL** ($l+1, u, S$) can be executed in $O(\log(u-l))$ steps. To show that it selects every key with nearly the same probability we prove the following lemma.

LEMMA 3. *If $u-l \geq 2$ then for any subset Q of K_{l+1}, \dots, K_u the probability that **SEL** ($l+1, u, S$) selects an element of Q is at least $|Q|/(u-l+2)$.*

Proof. Let x equal $2^{2 \lceil \log(u-l) \rceil}$. For at least $|Q| \cdot \lfloor x/(u-l) \rfloor$ numbers N between 0 and $x-1$ $N \bmod (u-l)$ denotes the index of an element in Q . Thus the probability can be estimated by

$$|Q| \cdot \frac{\lfloor x/(u-l) \rfloor}{x} \geq |Q| \frac{x/(u-l) - 1}{x} = |Q| \left(\frac{1}{u-l} - \frac{1}{x} \right) \geq |Q| \left(\frac{1}{u-l} - \frac{1}{(u-l)^2} \right).$$

It is easy to see that for $z \geq 2$ $1/z - 1/z^2 \geq 1/(z+2)$. \square

The next procedure computes all partial sums of a sequence of m numbers in $O(\log m)$ steps.

ALLSUM ($l+1, u, C, D$)

comment: processors P_{l+1}, \dots, P_u compute all sums $\sum_{j=l+1}^i C_j$ for $i = l+1, \dots, u$ and store the results in array D . F and H are additional local arrays.

1. **in parallel** [$P_i | l+1 \leq i \leq u$]: **set** $F_i \leftarrow C_i, H_i \leftarrow l, D_i \leftarrow 0$
2. **for** $k = 1, \dots, \lfloor \log(u-l) \rfloor$ **do**
 in parallel [$P_{l+2^k \cdot i} | 1 \leq i \leq \lfloor (u-l)/2^k \rfloor$]:

$$F_{l+2^k \cdot i} \leftarrow F_{l+2^k \cdot i} + F_{l+2^k \cdot i - 2^{k-1}}$$

end

comment: now for $l+1 \leq j \leq u$ with $j = l+2^k \cdot i$ and i odd holds:

$$F_j = \sum_{p=1}^{2^k} C_{l+2^k(i-1)+p}$$

3. **in parallel** [$P_i | l+1 \leq i \leq u$]:
 for $k = \lfloor \log(u-l) \rfloor, \dots, 0$ **do**
 if $H_i + 2^k \leq i$ **then** $H_i \leftarrow H_i + 2^k$
 $D_i \leftarrow D_i + F_{H_i}$
 end
end ALLSUM.

We now describe the procedure PPSORT to sort a box $B = K_{l+1}, \dots, K_u$ with processors P_{l+1}, \dots, P_u . As input parameters only $l+1$ and u are specified, the lower and upper boundary of B . Global variables for this procedure are the sequence K of keys and the arrays L and U from which each processor gets $l+1$ and u . Before starting PPSORT ($l+1, u$) to sort the original sequence L is set identically to $l+1$ and U to u .

To store intermediate results some additional global arrays S, C, D, G^j are used, each of length n . During PPSORT ($l+1, u$) processors P_{l+1}, \dots, P_u only access global arrays in the range between $l+1$ and u . Let n be large enough for the following analysis.

PPSORT ($l+1, u$)

if $u-l \leq \log n$ **then do**

0. **in parallel** [$P_i | l+1 \leq i \leq u$]:
 set $C_i \leftarrow 0$
 for $j \in \{l+1, \dots, u\}$ **do**
 if $K_i \geq K_j$ **then** $C_i \leftarrow C_i + 1$
 end
 set $K_{l+C_i} \leftarrow K_i$.

end

if $u-l > \log n$ **then do**

1. **in parallel** [$P_i | l+1 \leq i \leq u$]:
 compute and store in private memory

$$w := \lfloor \sqrt{u-l} \rfloor, \quad v := \left\lfloor \frac{u-l}{w} \right\rfloor, \quad z := 2^{2 \lfloor \log w \rfloor + 2}$$

in steps 2–5 a subset Y of K_{l+1}, \dots, K_u of size w is selected at random and sorted

2. **call in parallel for** $0 \leq j < w$ **SEL** ($l+jv+1, l+(j+1)v, S$)
 3. **in parallel** [$P_{l+iv+j+1} | 0 \leq i, j < w$]:
 if $S_{l+iv+1} \geq S_{l+jv+1}$ **then** $C_{l+iv+j+1} \leftarrow 1$
 else $C_{l+iv+j+1} \leftarrow 0$
 4. **call in parallel for** $0 \leq i < w$ **ALLSUM** ($l+iv+1, l+(i+1)v, C, D$)
 (D_{l+iv+1} is the rank of S_{l+iv+1} in Y)
 5. **in parallel** [$P_{l+iv+1} | 0 \leq i < w$]: **set** $S_{l+D_{l+(i+1)v}} \leftarrow S_{l+iv+1}$
 (S_{l+1}, \dots, S_{l+w} is the ordered sequence of keys in Y)

in steps 6–7 it is determined into which box B^j each key K_{l+1}, \dots, K_u falls, where

$$B^0 := \{K_i | l+1 \leq i \leq u, K_i < S_{l+1}\},$$

$$B^j := \{K_i | l+1 \leq i \leq u, S_{l+j} \leq K_i < S_{l+j+1}\} \text{ for } 1 \leq j < w \text{ and}$$

$$B^w := \{K_i | l+1 \leq i \leq u, S_{l+w} \leq K_i\}.$$

6. **in parallel** [$P_i | l+1 \leq i \leq u$]: $C_i \leftarrow 0$
 7. **for** $k = \lfloor \log w \rfloor, \dots, 0$ **do**
 in parallel [$P_i | l+1 \leq i \leq u$]:
 if $C_i + 2^k \leq w$ **and** $S_{l+C_i+2^k} \leq K_i$ **then** $C_i \leftarrow C_i + 2^k$
end

(C_i is the index of that box to which K_i belongs)
 in steps 8-13 the sizes of the boxes B^j and a relative position for each element of B^j is computed.

8. **in parallel** [$P_i | l+1 \leq i \leq u$]: **set** $D_i \leftarrow z^{C_i}$
 9. **call** ALLSUM ($l+1, u, D; E$)
 (if $E_s = \sum_{i=l+1}^s D_i$ for any $l+1 \leq s \leq u$ is expanded to the base z , that means $E_s = \sum_{j=0}^w \alpha_{j,s} z^j$ with $0 \leq \alpha_{j,s} \leq u-l < z$, then the coefficient $\alpha_{j,s}$ equals the number of elements among K_{l+1}, \dots, K_s that belong to box B^j).
 10. **in parallel** [$P_{l+1+j} | 0 \leq j \leq w$]: $G_{l+1+j}^1 \leftarrow \alpha_{j,u} = \lfloor E_u / z^j \rfloor - z \lfloor E_u / z^{j+1} \rfloor$
 11. **call** ALLSUM ($l+1, l+1+w, G^1, G^2$)
 (G_{l+1+j}^1 (resp. $G_{l+1+j}^2 = \sum_{k=0}^j G_{l+1+k}^1$) is the number of elements among K_{l+1}, \dots, K_u belonging to B^j (resp. $B^0 \cup \dots \cup B^j$))
 12. **in parallel** [$P_i | l+1 \leq i \leq u$]:
set $G_i^3 \leftarrow \alpha_{C_i, i} = \lfloor D_i / z^{C_i} \rfloor - z \lfloor D_i / z^{(C_i+1)} \rfloor$
if $C_i = 0$ **then** $G_i^4 \leftarrow G_i^3$
 else $G_i^4 \leftarrow G_{l+C_i}^2 + G_i^3$
set $K_{l+G_i^4} \leftarrow K_i$
 (G_i^4 equals the number of keys among K_{l+1}, \dots, K_i that belong to a box B^j with $j \leq C_i$; therefore G_{l+1}^4, \dots, G_u^4 is a permutation of $\{1, \dots, u-l\}$ and after rearranging the array K the elements of B^j are stored in $K_{l+1+G_{l+j}^4}, \dots, K_{l+G_{l+j+1}^4}$)
 13. **in parallel** [$P_i | l+1 \leq i \leq u$]:
if $C_i > 0$ **then** $L_{l+G_i^4} \leftarrow l+1 + G_{l+C_i}^2$
 $U_{l+G_i^4} \leftarrow l + G_{l+1+C_i}^2$
 14. **call in parallel** PPSORT ($l+1, l+G_{l+1}^2$) and for $0 < j \leq w$
 PPSORT ($l+1+G_{l+j}^2, l+G_{l+j+1}^2$)
- end**
end PPSORT

We will not show the correctness of this algorithm because it follows easily from the comments added to the description. Note that if each processor P_i knows its index i , then all processors can be programmed equally.

THEOREM 3. *For any n there is a probabilistic n -processor RAM that sorts n keys and for any input takes no more than $\alpha \log n$ steps for some constant α with probability at least $1 - \exp(-\Omega((\log n)^{5/4}))$.*

Proof. Each of the steps 0-13 in PPSORT ($l+1, u$) takes $O(\log(u-l))$ time. It remains to estimate the size of the boxes B^j into which B is partitioned.

LEMMA 4. *For all $x \in \mathbb{R}, x \geq 1$,*

- i) $\lfloor \sqrt{x} \rfloor \leq \lfloor x / \lfloor \sqrt{x} \rfloor \rfloor \leq \lfloor \sqrt{x} \rfloor + 2$.
- ii) $0 \leq x - \lfloor \sqrt{x} \rfloor \cdot \lfloor x / \lfloor \sqrt{x} \rfloor \rfloor < \lfloor \sqrt{x} \rfloor$.

Proof. Distinguish 4 cases:

| case | $\lfloor \sqrt{x} \rfloor$ | $\lfloor x / \lfloor \sqrt{x} \rfloor \rfloor$ | $\lfloor \sqrt{x} \rfloor \lfloor x / \lfloor \sqrt{x} \rfloor \rfloor$ |
|-----------------------------|----------------------------|--|---|
| $x = y^2$ | y | y | y^2 |
| $y^2 < x < y^2 + y$ | y | y | y^2 |
| $y^2 + y \leq x < y^2 + 2y$ | y | $y+1$ | $y^2 + y$ |
| $y^2 + 2y \leq x < (y+1)^2$ | y | $y+2$ | $y^2 + 2y$ |

□

Let $B = K_{j+1}, \dots, K_u$ be a box of K where the elements of B may be in any order. Let w and v be defined as in step 1 and B^0, \dots, B^w be the boxes into which B is split. For $0 \leq j \leq w$ let b_j be a random variable denoting the size of B^j .

LEMMA 5. *If $u - l \geq 16$ then for any $\beta \geq 0$*

$$P(\text{some } b_j \text{ exceeds } 2\beta) \leq (u - l) \exp\left(1 - \frac{\beta}{\sqrt{u - l}}\right).$$

Proof. In step 2 of the algorithm B is divided into w subsequences $\Gamma^0, \dots, \Gamma^{w-1}$ of size v , from which each one element is selected at random, and a rest Γ^w of size $(u - l) - wv$. By Lemma 4 $(u - l) - wv < w \leq \sqrt{u - l}$. Define a permutation π of $\{1, \dots, u - l\}$ by $K_{l+\pi(1)} < K_{l+\pi(2)} < \dots < K_{l+\pi(u-l)}$.

For a subsequence Δ of 2β consecutive elements of this ordered sequence let $\beta_i, 0 \leq i \leq w$, be the number of keys in Δ that belong to Γ^i . It holds $\sum_{i=0}^w \beta_i = 2\beta$ and $\beta_w < w$. From Lemma 3 follows that the probability not to select any of these elements when choosing one from each of $\Gamma^0, \dots, \Gamma^{w-1}$ is at most

$$\max_{\beta_0, \dots, \beta_w} \sum_{i=0}^{w-1} \left(1 - \frac{\beta_i}{v + 2}\right).$$

Taking the logarithm yields

$$\begin{aligned} \max_{\beta_0, \dots, \beta_w} \sum_{i=0}^{w-1} \ln\left(1 - \frac{\beta_i}{v + 2}\right) &\leq \max_{\beta_0, \dots, \beta_w} \sum_{i=0}^{w-1} -\frac{\beta_i}{v + 2} \\ &\leq \max_{\beta_0, \dots, \beta_w} -\frac{2\beta}{v + 2} + \frac{\beta_w}{v + 2} \leq -\frac{2\beta}{\sqrt{u - l} + 4} + 1 \quad \text{by Lemma 4 and } \beta_w < w \\ &\leq -\frac{\beta}{\sqrt{u - l}} + 1 \quad \text{since } \sqrt{u - l} \geq 4. \end{aligned}$$

This gives

$$\begin{aligned} P(\text{for some } j \ b_j > 2\beta) &\leq P(\text{there is a subsequence } \Delta \text{ of length } 2\beta \text{ of which no element is selected}), \\ &\leq \sum_{\substack{\Delta \\ \text{length of } \Delta = 2\beta}} P(\text{no element of } \Delta \text{ is selected}) \\ &\leq (u - l) \cdot \exp\left(-\frac{\beta}{\sqrt{u - l}} + 1\right). \quad \square \end{aligned}$$

Similar to the proof of Theorem 2 the computation of the parallel RAM on a sequence K of n keys can be represented by a tree, where each node is a box B generated by the recursive partitioning. Sons of a node B are the Boxes B^j into which B is partitioned. The root represents K and leaves are the small boxes.

The runtime obviously depends on how fast any larger box B is completely divided into small boxes. Lemma 5 shows that for each individual box with high probability all its subboxes generated by one partitioning become much smaller, call such an operation successful. But after several recursive steps the algorithm has to handle many still relatively large boxes in parallel, and the more such boxes there are the more likely it is that at least for some of these boxes the partitioning is not successful. Thus

simple repeated application of Lemma 5 to the global tree cannot show that the algorithm finishes fast with high probability.

Instead we do the following: the range of possible sizes of the generated boxes, from maximal n up to minimal 1, is divided into intervals I_0, I_1, \dots . Then for every k the probability is estimated that for any box, of which the size falls into interval I_k , the size of at least one of its subboxes obtained by some number τ_k of repeated partitionings still belongs to this interval.

For suitable intervals and parameters τ_k increasing with k this probability is even globally very small and the claimed bounds follow.

Let $\frac{1}{2} < \gamma < 1$ and $1 < d < 1/\gamma$.

For

$$0 \leq k < M := \left\lceil \frac{\log \log n - \log \log \log n}{\log(1/\gamma)} \right\rceil$$

define

$$\tau_k = d^k, \quad \rho_k = n^{\gamma^k}, \quad \rho_M = 0, \quad \text{interval } I_k =]\rho_{k+1}, \rho_k] \subset \mathbb{R}.$$

Simple arithmetic transformations, using

$$\frac{\log \log n - \log \log \log n}{\log(1/\gamma)} - 1 \leq M - 1 \leq \frac{\log \log n - \log \log \log n}{\log(1/\gamma)},$$

yields

$$(4.1) \quad \begin{aligned} & \tau_{k-1} < \tau_k, \quad \rho_{k-1} > \rho_k \quad \text{for all } 0 \leq k < M, \\ & \frac{1}{d} \left(\frac{\log n}{\log \log n} \right)^{\log d / \log(1/\gamma)} \leq \tau_{M-1} \leq \left(\frac{\log n}{\log \log n} \right)^{\log d / \log(1/\gamma)}, \\ & \rho_{M-1} \geq \log n, \quad I_0 =]n^\gamma, n], \quad [1, \log n] \subset I_{M-1}. \end{aligned}$$

For an internal node B define $i(B)$ by $|B| \in I_{i(B)}$, that is the index of that interval I_k which contains the size of B , and let $s(B)$ be the number of steps 1-13 to split B . We have

$$(4.2) \quad s(B) = O(\log \rho_{i(B)}).$$

If $Q = (B_1, \dots, B_t)$ is a path from the root to a leaf then $0 = i(B_1) \leq \dots \leq i(B_{t-1})$ and $\sum_{j=1}^{t-1} s(B_j) + O(\log n)$ gives the time until subbox B_t is sorted when started with the sequence $B_1 = K$ of length n . Hence the runtime of the algorithm on K can be estimated by

$$(4.3) \quad \max_{B_1, \dots, B_t} \sum_{j=1}^{t-1} s(B_j) + O(\log n)$$

where the max ranges over all paths Q from the root to a leaf. Let E_Q denote the event: for all $0 \leq k \leq M$ the sequence $i(B_1), \dots, i(B_{t-1})$ does not contain k more than τ_k times.

If E_Q holds then the time spent on Q does not exceed

$$(4.4) \quad \sum_{k=0}^{M-1} O(\log n \cdot \gamma^k) \tau_k + O(\log n) = O(\log n) \left[1 + \sum_k \gamma^k \cdot d^k \right] = O(\log n) \quad \text{since } \gamma d < 1.$$

From Lemma 5 it follows for any j and $\lambda < M$ that

$$\begin{aligned} P(i(B_{j+1}) = i(B_j) | i(B_j) = \lambda) & \\ & \cong |B_j| \exp\left(1 - \frac{1}{2} \frac{|B_{j+1}|}{\sqrt{|B_j|}}\right) \\ & \cong \rho_\lambda \exp\left(1 - \frac{1}{2} \frac{\rho_{\lambda+1}}{\sqrt{\rho_\lambda}}\right) \quad \text{since } |B_j| \leq \rho_\lambda \text{ and } |B_{j+1}| \geq \rho_{\lambda+1} \\ & = \exp\left(1 + \ln(n^{\gamma^\lambda}) - \frac{1}{2} \frac{n^{\gamma^{\lambda+1}}}{n^{\gamma^{\lambda/2}}}\right) \\ & = \exp\left(1 + \gamma^\lambda \ln n - \frac{1}{2} \frac{n^{\gamma^{\lambda+1}}}{(n^{\gamma^{\lambda+1}})^{1/2\gamma}}\right) \\ & \leq \exp\left(-\frac{1}{3}(n^{\gamma^{\lambda+1}})^{1-1/2\gamma}\right) \end{aligned}$$

for all n large enough since by (4.1) $n^{\gamma^\lambda} \geq \log n$ and $1/2\gamma < 1$.

This implies

$$\begin{aligned} P(i(B_{j+\tau}) = i(B_{j+\tau-1}) = \dots = i(B_j) | i(B_j) = \lambda) & \\ & = \prod_{l=j+1}^{j+\tau} P(i(B_l) = i(B_{l-1}) | i(B_{l-1}) = \dots = i(B_j) = \lambda) \\ & \leq \exp\left(-\frac{\tau}{3}(n^{\gamma^{\lambda+1}})^{1-1/2\gamma}\right) \end{aligned}$$

and

$P(\text{there exists } 0 \leq k < M \text{ such that the sequence } i(B_1), \dots, i(B_{t-1}) \text{ contains } k \text{ more than } \tau_k \text{ times})$

$$\begin{aligned} (4.5) \quad & \leq \sum_{k=0}^{M-1} \sum_{j=1}^{t-\tau_k} P(i(B_{j+\tau_k}) = \dots = i(B_j) = k) \\ & = \sum_k \sum_j P(i(B_{j+\tau_k}) = \dots = i(B_j) | i(B_j) = k) \cdot P(i(B_j) = k) \\ & \leq \sum_k \frac{n}{\log n} \exp\left(-\frac{\tau_k}{3}(n^{\gamma^{k+1}})^{1-1/2\gamma}\right) \quad \text{since } t \leq n/\log n. \end{aligned}$$

For $k < M$ holds

$$\begin{aligned} \frac{(\tau_k/3)(n^{\gamma^{k+1}})^{1-1/2\gamma}}{(\tau_{k-1}/3)(n^{\gamma^k})^{1-1/2\gamma}} & = d(n^{\gamma^{k+1}-\gamma^k})^{1-1/2\gamma} = d(n^{\gamma^k})^{(\gamma-1)(1-1/2\gamma)} \\ & \leq d(\log n)^\varepsilon \quad \text{with } \varepsilon = (\gamma-1)\left(1 - \frac{1}{2\gamma}\right) < 0. \end{aligned}$$

Thus if n is large enough $(\tau_k/3)(n^{\gamma^{k+1}})^{1-1/2\gamma}$ is monotone decreasing in k and (4.5) can be estimated by

$$\begin{aligned} & \sum_{k=0}^{M-1} \frac{n}{\log n} \exp\left(-\frac{\tau_k}{3}(n^{\gamma^{k+1}})^{1-1/2\gamma}\right) \\ & \leq n \exp\left(-\frac{\tau_{M-1}}{3}(n^{\gamma^{M-1}})^{\gamma-1/2}\right), \end{aligned}$$

$$\begin{aligned} &\cong n \exp \left(1 - \frac{1}{3d} \left(\frac{\log n}{\log \log n} \right)^{\log d / \log (1/\gamma)} (\log n)^{\gamma-1/2} \right) \\ &\cong n \exp \left(-\frac{1}{3d} (\log n)^{(\log d / \log (1/\gamma)) + \gamma - 1/2} (\log \log n)^{-\log d / \log (1/\gamma)} \right). \end{aligned}$$

Since there are fewer than n paths from the root to a leave the probability that for some path Q event E_Q does not happen is less than

$$(4.6) \quad 1 - n^2 \exp \left(-\frac{1}{3d} (\log n)^{(\log d / \log (1/\gamma)) + \gamma - 1/2} (\log \log n)^{-\log d / \log (1/\gamma)} \right).$$

Choosing $\gamma = \frac{5}{6}$ and $d = 1.19$ gives $\log d / \log (1/\gamma) + \gamma - \frac{1}{2} > 1.28$, and bounds (4.6) by

$$1 - \exp \left(2 \ln n - \frac{1}{3d} (\log n)^{1.28} (\log \log n)^{-1} \right) \cong 1 - \exp (-\Omega((\log n)^4)). \quad \square$$

5. New results. The $n \log n$ -processor sorting algorithm in [4] uses an algorithm for merging ordered sequences as a subroutine, but the paper does not contain a solution to the problem of processor assignment for this kind of merging algorithms. A. Borodin and J. Hopcroft show in *Routing, merging and sorting on parallel models of computation*, 14th ACM-STOC, 1982, pp. 338-344 how this problem can be solved efficiently by a parallel RAM implying a deterministic n -processor $\log \log n$ merging algorithm and a deterministic $n \log n$ -processor $\log n$ sorting algorithm.

M. Ajtai, J. Komlós and E. Szemerédi announce to present an $O(n \log n)$ sorting network at the 15th ACM-STOC, April 1983.

REFERENCES

[1] A. AHO, J. HOPCROFT AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
 [2] R. FLOYD AND R. RIVEST, *Expected time bounds for selection*, Comm. ACM, 18 (1975), pp. 165-172.
 [3] P. HÄGGKVIST AND P. HELL, *Parallel sorting with constant time for comparisons*, this Journal, 10 (1981), pp. 465-472.
 [4] F. P. PREPARATA, *New parallel-sorting schemes*, IEEE Trans. Comp. C-27 (1978), pp. 669-773.
 [5] L. G. VALIANT, *Parallelism in comparison problems*, this Journal, 4 (1975), pp. 348-355.

AN "INTERCHANGE LEMMA" FOR CONTEXT-FREE LANGUAGES*

WILLIAM OGDEN[†], ROCKFORD J. ROSS[‡] AND KARL WINKLMANN[§]

Abstract. An "Interchange Lemma" is proven, providing a new necessary condition for languages to be context-free. This "Interchange Lemma" is then used to show that the set of repetitive strings (i.e. strings of the form xyz with nonempty y) over an alphabet of three or more characters is not context-free—an issue which in the past has shown remarkable resilience against standard tools for proving languages not context-free and which has only recently been solved independently in [5] and [10]. The Interchange Lemma presented in this paper is a generalization of the proof technique used in [10].

Key words. context-free languages, interchange lemma, repetitive strings

1. Introduction. Formal language theory provides a variety of tools for proving languages not context-free, e.g. various pumping lemmas and closure results. Still there are problems which do not seem to yield to these classical tools. One such problem is the question of whether or not the set of repetitive strings over an alphabet of three or more characters is context-free (see e.g. [1, pp. 374–375]). Intuitively, the answer is easy: Context-free languages are recognized by pushdown automata, but the first-in-last-out character of a pushdown store makes this storage structure useless for recognizing repetitive strings; therefore the language cannot be context-free. In spite of this clear and easy intuition, the problem was only recently solved [5], [10] after having been open for a long time. This contrast between easy intuition and apparent technical difficulty suggests a deficiency in the theory of context-free languages.

In this paper we attempt to remove this deficiency by generalizing the proof technique used in [10]. This generalization takes the form of an "Interchange Lemma", providing a new necessary condition for languages to be context-free. Unlike pumping lemmas, which predict increasingly *longer* strings to be in a language, the Interchange Lemma points out relationships between strings of the *same length*. Put briefly it says that if a context-free language L contains many strings of some fixed length, then parts of these strings may be interchanged, giving new strings which must also be in L .

In § 2 we state and prove the Interchange Lemma. In § 3 we apply it to show that the set of repetitive strings over an alphabet of three or more characters is not context-free.

2. The Interchange Lemma.

Notation. A context-free grammar (cfg) G is a quadruple (I, E, P, S) . Here I is the set of *internal symbols* (nonterminal symbols, variables), E is the set of *external symbols* (terminal symbols), P is the set of *productions* and S is the *start symbol*. We will use L_n to denote the set of all strings of length n in a language L . Any other notation we use is standard.

* Received by the editors March 16, 1982.

[†] Department of Computing and Information Science, Ohio State University, Columbus, Ohio 43210.

[‡] Department of Electrical Engineering and Computer Science, Montana State University, Bozeman, Montana 59717. This research was conducted while the author was at the Department of Computer Science, Washington State University, Pullman, Washington 99164.

[§] Department of Computer Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1. The research of this author was supported in part by the National Science Foundation under grant MCS 8004128 while he was at the Department of Computer Science, Washington State University, Pullman, Washington 99164.

An informal statement of the Interchange Lemma. The Interchange Lemma concerns properties of L_n . In some applications these sets L_n may be difficult to work with. Therefore, our Interchange Lemma will actually make statements about arbitrary subsets of L_n . These subsets can then be chosen conveniently when applying the Lemma. Specifically, the Interchange Lemma will state that if the cardinality of a subset Q_n of L_n is “large enough”, then there will be strings $z_1 = w_1x_1y_1$ and $z_2 = w_2x_2y_2$ in Q_n with $|w_1| = |w_2|$, $|x_1| = |x_2|$, and $|y_1| = |y_2|$ such that $w_1x_2y_1$ and $w_2x_1y_2$ are in L_n . “Large enough” will mean, roughly, of size n^2 or bigger.

DEFINITION 1. Let $G = (I, E, P, S)$ be a cfg generating L and n be a positive integer. For any subset Q_n of L_n , $A \in I$, and nonnegative integers n_1 and n_2 with $n_1 + n_2 \leq n$, define $Q_n(n_1, A, n_2)$ to be the set of words $z \in Q_n$ which have a derivation of the form

$$(*) \quad S \xRightarrow{*} wAy \xRightarrow{*} wxy = z \text{ with } |w| = n_1 \text{ and } |y| = n_2.$$

(Necessarily $|x| = n - n_1 - n_2$.) \square

Informally, the set $Q_n(n_1, A, n_2)$ thus contains all words in Q_n for which there is a derivation in which the internal symbol A occurs “in the same place”. Specifically, in each such derivation there is an occurrence of A which produces a substring of length $n - n_1 - n_2$ that starts with the $(n_1 + 1)$ st character of the whole string.

Remark 1. If z_1 and z_2 are two words in $Q_n(n_1, A, n_2)$, then by definition there are two derivations,

$$S \xRightarrow{*} w_1Ay_1 \xRightarrow{*} w_1x_1y_1 = z_1$$

and

$$S \xRightarrow{*} w_2Ay_2 \xRightarrow{*} w_2x_2y_2 = z_2,$$

of the form $(*)$ of Definition 1. It then follows that there are also derivations

$$S \xRightarrow{*} w_1Ay_1 \xRightarrow{*} w_1x_2y_1$$

and

$$S \xRightarrow{*} w_2Ay_2 \xRightarrow{*} w_2x_1y_2$$

having the same form $(*)$. Hence $w_1x_2y_1$ and $w_2x_1y_2$ are in L_n . (Since Q_n is arbitrary it is possible that the strings $w_1x_2y_1$ and $w_2x_1y_2$ are not in Q_n ; it is also possible that these strings are not different from the original strings z_1 and z_2 .) \square

It will be useful to correlate the length of the substrings x in derivations of the form $(*)$ with the length n of the entire string. Lemma 1 does this.

LEMMA 1. Let $G = (I, E, P, S)$ be a cfg generating a language L , let $r \geq 2$ be the length of a longest right-hand side of any production in P , and let n and m be integers with $n \geq m \geq r$. Then for each $z \in L_n$ there is a symbol $A \in I$ and a derivation of the form

$$S \xRightarrow{*} wAy \xRightarrow{*} wxy = z$$

with $m \geq |x| > m/r$ and $A \in I$.

Proof. We have to show that a derivation tree for z must contain a subtree with l leaves for some l satisfying $m \geq l > m/r$. The following algorithm finds such a subtree. “leaves under a node X ” is short for “leaves in the subtree of which node X is the root”.

Step 1. Consider the root of the derivation tree as the current node.

Step 2. While there are more than m leaves under the current node, repeat Step 3.

Step 3. Choose from among the children of the current node one with the largest number of leaves under it. (Break ties arbitrarily.) Make that child the new current node.

This algorithm stops because the current node keeps moving down a path in the derivation tree and hence will eventually satisfy the termination condition of Step 2. When the algorithm stops the current node is the root of a subtree with l leaves for some l with $m \cong l > m/r \cong 1$ because

- there are at least m leaves under the root of the derivation tree;
- Step 3 is only applied if there are more than m leaves under the current node;

and

— an application of Step 3 reduces the number of leaves under the current node at most by a factor of r . \square

LEMMA 2. *Let $G = (I, E, P, S)$ be a cfg generating a language L , let $r \cong 2$ be the length of a longest right-hand side of any production in P , and let n and m be integers with $n \cong m \cong r$. Then for any subset Q_n of L_n there are nonnegative integers n_1 and n_2 satisfying $m \cong n - n_1 - n_2 > m/r$ and an internal symbol $A \in I$ such that $\|Q_n(n_1, A, n_2)\| \cong \|Q_n\| / (\|I\|n^2)$.*

Proof. By Lemma 1

$$Q_n = \cup Q_n(n_1, A, n_2),$$

and hence

$$\|Q_n\| = \|\cup Q_n(n_1, A, n_2)\| \cong \sum \|Q_n(n_1, A, n_2)\|$$

where union and summation are over all $A \in I$ and over all nonnegative integers n_1 and n_2 with $m \cong n - n_1 - n_2 > m/r$. Since there are no more than $\|I\|n^2$ terms in the sum at least one of them must equal or exceed $\|Q_n\| / (\|I\|n^2)$. \square

Now we can easily prove the following Interchange Lemma. As before, L_n denotes the set of strings from L which have length n .

INTERCHANGE LEMMA. *Let L be a cfl. Then there is a constant c_L such that for any integer $n \cong 2$, any subset Q_n of L_n , and any integer m with $n \cong m \cong 2$ there are $k \cong \|Q_n\| / (c_L n^2)$ strings z_i in Q_n with the following properties:*

- (i) $z_i = w_i x_i y_i, i = 1, \dots, k$;
- (ii) $|w_1| = |w_2| = \dots = |w_k|$;
- (iii) $|y_1| = |y_2| = \dots = |y_k|$;
- (iv) $m \cong |x_1| = |x_2| = \dots = |x_k| > m/2$; and
- (v) $w_i x_j y_i \in L_n$ for all $i, j \in \{1, \dots, k\}$.

Proof. Let $G = (I, E, P, S)$ be a cfg in Chomsky normal form (see e.g. [8, pp. 92–94]) generating L . Choose $c_L = \|I\|$. Then by Lemma 2 there exist integers n_1 and n_2 satisfying $m \cong n - n_1 - n_2 > m/2$ and an internal symbol $A \in I$ such that $\|Q_n(n_1, A, n_2)\| \cong \|Q_n\| / (c_L n^2)$. By Definition 1 each string $z_i, i = 1, \dots, k$, in this set $Q_n(n_1, A, n_2)$ has a derivation of the form

$$S \xrightarrow{*} w_i A y_i \xrightarrow{*} w_i x_i y_i = z_i \text{ with } |w_i| = n_1 \text{ and } |y_i| = n_2.$$

Such strings w_i, x_i , and $y_i, i = 1, \dots, k$, satisfy claims (i)–(iv). Claim (v) follows from Remark 1. \square

3. An application of the Interchange Lemma. We now apply the Interchange Lemma to repetitive strings. A nonempty string of the form yy is called a *repetition*. A string which contains a repetition as a substring (i.e., a string of the form $xxyz$ with nonempty y) is called a *repetitive string*. Interest in repetitive and nonrepetitive strings dates back to Thue’s 1906 paper [12]. One summary of Thue’s work can be found in [11].

THEOREM [5], [10]. *The set of repetitive strings over a three-letter alphabet is not context-free.*

Proof. The main part of this proof is an application of the Interchange Lemma to prove the following Lemma 3.

LEMMA 3. *The set of repetitive strings over a six-letter alphabet is not context-free.*

Postponing the proof of Lemma 3 for a moment, we observe that the theorem follows from Lemma 3 by the fact that context-free languages are closed under inverse homomorphism (see e.g. [8, pp. 132–133]) and the following result, which is a direct consequence of [2, Thm. 1.7].

COROLLARY TO [2, Thm. 1.7]. *There is an ε -free homomorphism h from a six-letter alphabet to a three-letter alphabet such that for all strings w , w is repetitive if and only if $h(w)$ is repetitive. \square*

We now finish the proof of the theorem by providing a proof of Lemma 3.

Proof of Lemma 3.

Let L be the set of repetitive string over a six-letter alphabet. Assume that L is context-free. In the following we will lead this assumption to a contradiction.

Choose n to be divisible by 4 and large enough to satisfy

$$2^{n/4}/(c_L n^2) > 2^{n/8+1}$$

where c_L is the constant from the Interchange Lemma. (The reason for this choice will become clear.) Throughout this proof we keep n fixed. Therefore we do not always show it explicitly as a subscript.

Choose a nonrepetitive string r' of length $n/4 - 1$ over the three-letter alphabet $\{a, b, c\}$. (The fact that arbitrarily long nonrepetitive strings over a three-letter alphabet do exist was first shown in [12]. Proofs can also be found in [4], [7], [9], [13] and [6, pp. 36–40]. See also [3].) Define $r = \$r'$. The following observation is a straightforward consequence of this choice of r .

Observation 1. The string rr contains only one repetition (rr itself).

For any two strings $u = a_1 a_2 \cdots a_p$ and $v = b_1 b_2 \cdots b_p$ of equal length define *the interleaving $I(u, v)$ of u with v* to be the string $a_1 b_1 a_2 b_2 \cdots a_p b_p$. Now define

$$A_n = \{I(rr, s) : s \in \{0, 1\}^{n/2}\}.$$

Then every string in A_n has length n and typically looks like

$$\$0a0b1c0a1b1a1c0 \cdots 1a0\$1a0b1c0a0b0a1c1 \cdots 1a1.$$

The “ a - b - c -pattern” is the same in both halves of the string and *the same in every string in A_n* . The “0-1-pattern” is entirely arbitrary throughout the string. A simple consequence of this definition of A_n is the following observation. Informally speaking, it states that any change in the “0-1-pattern” (to a new “0-1-pattern”) of a string z_1 in A_n yields a new string, z_2 , which is also in A_n .

Observation 2. Let $z_1 = w_1 x_1 y_1$ and $z_2 = w_2 x_2 y_2$ be strings from A_n with $|w_1| = |w_2|$, $|x_1| = |x_2|$, and $|y_1| = |y_2|$. Then the strings $w_1 x_2 y_1$ and $w_2 x_1 y_2$ are also in A_n .

Observation 3. Let z be a string from A_n . Then z is repetitive (i.e. z contains a repetition) if and only if z is a repetition (i.e. z if of the form $z'z'$). In other words,

the only way in which a string $z = I(rr, s)$ with $s \in \{0, 1\}^{n/2}$ can be repetitive is by having $s = s'$ for some $s' \in \{0, 1\}^{n/4}$.

Observation 3 follows easily from Observation 1 and the Definition of A_n .

The repetitive strings from A_n will form the set Q_n in our application of the Interchange Lemma. Define

$$Q_n = \{I(rr, ss) : s \in \{0, 1\}^{n/4}\}.$$

Observation 4.

$$\|Q_n\| = 2^{n/4}.$$

For strings in A_n we will use terms like “a pair of corresponding positions in the two halves” with the obvious meaning: the i th position of the first half of the string “corresponds” to the i th position of the second half.

Informally speaking, the next observation states that if we take a string from Q_n and change the 0-1-pattern in some substring of length no more than $n/2$ then the resulting string will not be repetitive.

Observation 5. Let $z_1 = w_1x_1y_1 \in Q_n$ and $z_2 = w_2x_2y_2 \in A_n$ with $|w_1| = |w_2|$, $|x_1| = |x_2| \leq n/2$, $|y_1| = |y_2|$, and $x_1 \neq x_2$. Then neither $w_1x_2y_1$ nor $w_2x_1y_2$ is a repetitive string.

Observation 5 is true for the following reason. When x_1 gets replaced by x_2 in the string z_1 some 0 or 1 in z_1 gets changed (since $x_1 \neq x_2$) but since $|x_1| = |x_2| \leq n/2$ the corresponding character in the other half of z_1 remains unchanged. Thus the resulting string $w_1x_2y_1$ is not a repetition and although $w_1x_2y_1$ is in A_n by Observation 2, by Observation 3 it is not repetitive.

Observation 6. Let n_1 and n_2 be two nonnegative integers with $n/2 \geq n - n_1 - n_2 > n/4$ and let S be a subset of Q_n with the property that all strings in S agree on positions $n_1 + 1$ through $n - n_2$ (in other words, two strings in S can only disagree in their first n_1 and their last n_2 characters). Then $\|S\| \leq 2^{n/8+1}$.

Observation 6 holds for the following reason. By assumption there is a stretch of $l = n - n_1 - n_2$ contiguous positions on which all strings in S agree. Because $l \geq n/4$ this stretch includes at least $n/8 - 1$ 0's and 1's but for none of these 0's and 1's does this stretch include the corresponding position in the other half of the string (since $l \leq n/2$). Consequently, from the definitions of Q_n and S all the strings in S agree on at least $n/8 - 1$ pairs of corresponding 0-1-positions. This leaves at most $n/8 + 1$ pairs of corresponding 0-1-positions to be filled arbitrarily.

Applying the Interchange Lemma with $m = n/2$ yields the following observation.

Observation 7 (application of the Interchange Lemma). Define $k = \|Q_n\| / (c_L n^2)$. There is a subset $R = \{z_1, \dots, z_k\}$ of Q_n which satisfies the following properties:

- (i) $z_i = w_i x_i y_i$, $i = 1, \dots, k$;
- (ii) $|w_1| = |w_2| = \dots = |w_k|$;
- (iii) $|y_1| = |y_2| = \dots = |y_k|$;
- (iv) $n/2 \geq |x_1| = |x_2| = \dots = |x_k| > n/4$; and
- (v) $w_i x_j y_i \in L_n$ for all $i, j \in \{1, \dots, k\}$.

To complete the proof, now, we must show that there are $p, q \in \{1, \dots, k\}$ with $x_p \neq x_q$. Let R be as in Observation 7. Then by Observation 4 and our initial choice of n the following observation holds.

Observation 8.

$$\|R\| > 2^{n/8+1}.$$

Let $w_i, x_i,$ and $y_i, i = 1, \dots, k,$ be as in Observation 7.

Observation 9. There are two subscripts $p, q \in \{1, \dots, k\}$ such that $x_p \neq x_q.$

Observation 9 holds for the following reason. Assume for the sake of a contradiction that all $x_i, i = 1, \dots, k,$ are equal. Define n_1 to be the length of the w_i 's and n_2 the length of the y_i 's in Observation 7. Applying Observation 6 to this set R yields

$$\|R\| \leq 2^{n/8+1}$$

contradicting Observation 8.

But now by Observation 5 neither $w_p x_q y_p$ nor $w_q x_p y_q$ are repetitive, contradicting (v) of the Interchange Lemma.

This ends the proof of Lemma 3. \square

4. Summary. The Interchange Lemma provides a new tool for proving languages not context-free. While many other tools exist for studying context-free languages none seems to apply directly to the question of whether or not repetitive strings are context-free. As shown in § 3 the Interchange Lemma allows a fairly simple answer to this question and hence seems to fill a gap in the theory of formal languages.

REFERENCES

- [1] J. M. AUTEBERT, J. BEAUQUIER, L. BOASSON AND M. NIVAT, *Quelques problèmes ouverts en théorie des langages algebriques*, RAIRO Informatique théorique/Theoretical Informatics, 13 (1979) pp. 363-378.
- [2] D. R. BEAN, A. EHRENFEUCHT AND G. F. McNULTY, *Avoidable patterns in strings of symbols*, Pacific J. Math., 85 (1979) pp. 261-294.
- [3] J. BERSTEL, *Sur les mots sans carre définis par un morphisme* in Automata, Languages, and Programming, H. A. Maurer, ed., Lecture Notes in Computer Science, 71, Springer-Verlag, New York, 1979, pp. 16-25.
- [4] C. H. BRAUNHOLTZ, *Solution to Problem 5030*, American Math. Monthly, 70 (1963), pp. 675-676.
- [5] A. EHRENFEUCHT AND G. ROZENBERG, *On the separating power of EOL systems*, RAIRO Informatique théorique/Theoretical Informatics, 17 (1983), pp. 13-22.
- [6] M. HARRISON, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA, 1978.
- [7] G. A. HEDLUND, *Remarks on the work of Axel Thue on sequences*, Nordisk Matematisk Tidsskrift, 15 (1967), pp. 148-150.
- [8] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.
- [9] P. A. B. PLEASANTS, *Nonrepetitive sequences*, Proc. Cambridge Philosophical Society, 68 (1970), pp. 267-274.
- [10] R. ROSS AND K. WINKLMANN, *Repetitive strings are not context-free*, RAIRO Informatique théorique/Theoretical Informatics, 16 (1982), pp. 191-199.
- [11] A. SALOMAA, *Jewels of Formal Language Theory*, Computer Science Press, Potomac, MD, 1981.
- [12] A. THUE, *Über unendliche Zeichenreihen*, Norske Videnskabers Selskabs Skrifter Mat.-Mat. Kl. (Kristiania) 1906, Nr. 7, 00. 1-22.
- [13] ———, *Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen*, Norske Videnskabers Selskabs Skrifter Mat.-Nat. Kl. (Kristiania) 1912, Nr. 1, pp. 1-67.

A STOCHASTIC MODEL OF FRAGMENTATION IN DYNAMIC STORAGE ALLOCATION*

E. G. COFFMAN, JR.†, T. T. KADOTA† AND L. A. SHEPP†

Abstract. We study a model of dynamic storage allocation in which requests for single units of memory arrive in a Poisson stream at rate λ and are accommodated by the first available location found in a linear scan of memory. Immediately after this first-fit assignment, an occupied location commences an exponential delay with rate parameter μ , after which the location again becomes available. The set of occupied locations (identified by their numbers) at time t forms a random subset S_t of $\{1, 2, \dots\}$. The extent of the fragmentation in S_t , i.e. the alternating holes and occupied regions of memory, is measured by $\max(S_t) - |S_t|$. In equilibrium, the number of occupied locations, $|S|$, is known to be Poisson distributed with mean $\rho = \lambda/\mu$. We obtain an explicit formula for the stationary distribution of $\max(S)$, the last occupied location, and by independent arguments we show that $(E \max(S) - E|S|)/E|S| \rightarrow 0$ as the traffic intensity $\rho \rightarrow \infty$. Moreover, we verify numerically that for any ρ the expected number of wasted locations in equilibrium is never more than $\frac{1}{3}$ the expected number of occupied locations.

Our model applies to studies of fragmentation in paged computer systems, and to containerization problems in industrial storage applications. Finally, our model can be regarded as a simple concrete model of interacting particles [Adv. Math., 5(1970), pp. 246–290].

Key words. dynamic storage allocation, checkerboarding, $M/M/\infty$ queue, memory allocation

1. Introduction. Adopting the terminology of queues, suppose customers arrive in a Poisson stream at rate λ to a linear queue of waiting or storage locations numbered $1, 2, \dots$. According to the so-called first-fit policy each customer occupies the lowest numbered location available at his time of arrival. Immediately upon being installed in an available location, a customer commences a delay or residence time having an exponential distribution with parameter μ . At the end of his residence time a customer departs from the queue, thus making available the location he occupied. As locations are occupied and released “holes” build up, so that the *total occupancy*, defined as the highest numbered location occupied by waiting customers, may be substantially greater than the number of customers in the queue. The principal objectives of this paper are an analysis leading to the stationary distribution of this total occupancy, and a characterization of the fraction of wasted space under heavy-traffic conditions, i.e. for large λ/μ .

In queueing parlance our model may be recognized as an $M/M/\infty$ queue on which a first-fit discipline for placement into a linear sequence of servers has been superimposed. The equilibrium distribution for the number in system is well-known for the $M/M/\infty$ system [4, p. 414].

Although we shall make some use of these classical results, we focus on the more difficult analysis of the total occupancy process, an analysis that is clearly more important in the applications noted below. Similar results for an $M/M/1$ queue have been obtained in [2], where conventional methods were found to be adequate. The greater difficulty of our problem stems from the more easily motivated, but combinatorially more complex, first-fit placement rule.

Interpreting customers as requests for single units of storage, our model is an instance of the general problem of dynamic storage allocation in computers. The elements of this subject have been treated by Knuth [6], and a recent survey appears in [1]. In particular, the infinite-server model was introduced in [6, p. 445] in an analysis of the well-known fifty-percent rule.

* Received by the editors September 28, 1982, and in revised form January 30, 1984.

† AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

Paged computer systems are specific applications of our model. Here, single units of storage become pages and locations become page frames, i.e. sets of consecutive memory locations that can accommodate exactly one page and that begin at integral multiples of the fixed page size. As before, the analysis of total occupancy leads to a characterization of the extent of fragmentation that occurs as pages come and go under a first-fit rule. Indeed, by an appropriate substitution of terms, our model applies quite generally to any such first-fit storage/server assignment problem where locations would correspond, for example, to telephone trunks, parking spaces, etc. There are a number of extensions to our model which would broaden its applicability. These, along with their implications for the analysis of stochastic models, are discussed in the last section.

Our analysis starts with the observation that the total occupancy process cannot be formulated as a Markov chain. We then identify a bivariate Markov chain in continuous time from which the stationary state probabilities of this process can be calculated. This calculation amounts primarily to a solution of the partial differential equation governing the generating function for the equilibrium probabilities of the bivariate Markov chain. For this purpose we adopt an apparently novel approach with generating functions, whereby a partial differential equation is replaced by an infinite but solvable system of ordinary differential equations. By an independent argument bounds on the expected total occupancy are derived; asymptotic properties of the total occupancy process are deduced from these bounds.

In the next section the mathematical model is formally defined. The major results, also presented in the next section, are then proved in §§ 3 and 4.

2. Mathematical model. For a fixed traffic intensity $\rho = \lambda/\mu > 0$ consider a continuous time Markov chain M_ρ whose states are the (finite) subsets of $\{1, 2, \dots\}$. A given state S_t is just that collection of numbers corresponding to occupied locations at time t . Transitions occur at rate $\mu > 0$ from any nonempty S to each of the $|S|$ subsets of S obtained by deleting one location from S , and at rate λ from any S to the union of S and the smallest numbered location not in S . Because S diminishes at rate $|S|\mu$, it is easy to see that for any ρ , if $|S_0| < \infty$ then $|S_t| < \infty$ for all $t > 0$ with probability 1 and that M_ρ is a Markov chain with a stationary distribution on the set of finite subsets of $\{1, 2, \dots\}$. It does not seem possible to obtain the stationary distribution of S , but we are able to find this distribution for both $|S|$ and $\max(S)$, the maximum element of S . Note that $\max(S)$ is simply the total occupancy mentioned in the previous section.

Letting $\pi_k = \lim_{t \rightarrow \infty} P(|S_t| = k)$, $k = 0, 1, \dots$, denote the stationary distribution, we have the following standard results from the analysis of an $M/M/\infty$ queue [4]

$$(2.1) \quad \pi_k = \frac{\rho^k}{k!} e^{-\rho}, \quad k = 0, 1, \dots$$

and

$$(2.2) \quad E|S| = \sum_{k \geq 0} k\pi_k = \rho.$$

Next, consider the distribution of $\max(S)$. We will prove in § 3 that

$$(2.3) \quad P(\max(S) > m) = \sum_{n=0}^{\infty} \frac{(-1)^n}{n+1} \frac{1}{\sum_{k=0}^m \binom{m}{k} (n+k)! / \rho^{n+k+1}}, \quad m = 0, 1, \dots.$$

Note that for $m = 0$, $P(\max(S) > 0) = 1 - P(S = \emptyset) = 1 - P(|S| = 0) = 1 - e^{-\rho}$, as seen from (2.1), and (2.3) agrees with this result. For $m = 1$, the right side of (2.3) becomes

an integral and

$$P(\max(S) > 1) = 1 - \int_0^1 e^{-\rho x^{1/\rho}} dx.$$

However, for large m and ρ , calculation of the right side of (2.3) is awkward. For these cases we can make use of the following crude bounds,

$$(2.4) \quad 1 - \sigma_m \leq P(\max(S) > m) \leq \rho \frac{\sigma_m - \sigma_{m-1}}{\sigma_m},$$

where

$$(2.5) \quad \sigma_m = e^{-\rho} \sum_{k=0}^m \frac{\rho^k}{k!}, \quad m = 0, 1, \dots.$$

To prove (2.4), note that $\max(S) \geq |S|$ so that $P(\max(S) > m) \geq P(|S| > m)$; the first inequality thus follows from (2.1) and (2.5). For the second, note first that

$$(2.6) \quad P(\max(S) > m) \leq E|S \cap \{m+1, m+2, \dots\}| = E|S| - E|S \cap \{1, 2, \dots, m\}|.$$

Now observe that $|S \cap \{1, \dots, m\}|$ is an embedded Markov process on states $0, 1, \dots, m$, and that its stationary distribution can be easily obtained as

$$(2.7) \quad P(|S \cap \{1, \dots, m\}| = k) = \frac{\rho^k}{k! \sum_{j=0}^m \rho^j / j!}, \quad k = 0, 1, \dots, m.$$

Using (2.7) in (2.6) gives the second inequality in (2.4).

We can use (2.4) to show that the rate of wastage (the fraction of holes) is asymptotically negligible for $\rho \rightarrow \infty$. First note that, for any integer K ,

$$(2.8) \quad E \max(S) = \sum_{m=0}^{\infty} P(\max(S) > m) \leq K + \sum_{m=K}^{\infty} P(\max(S) > m).$$

Choosing $K = \lfloor (1 + \epsilon)\rho \rfloor$ and using (2.4) and (2.8), we have, since $\sigma_m \uparrow 1$,

$$(2.9) \quad \frac{E \max(S)}{\rho} \leq 1 + \epsilon + \frac{1 - \sigma_{K-1}}{\sigma_K}.$$

Since for every $\epsilon > 0$, $1 - \sigma_{K-1} \rightarrow 0$ as $\rho \rightarrow \infty$, [4, p. 193] it follows from (2.9) that

$$(2.10) \quad \overline{\lim}_{\rho \rightarrow \infty} \frac{E \max(S)}{\rho} \leq 1.$$

On the other hand, $\max(S) \geq |S|$, thus $E \max(S) \geq E|S| = \rho$. Hence

$$(2.11) \quad E(\max(S) - |S|) = o(E|S|) \quad \text{as } \rho \rightarrow \infty.$$

Thus, as $\rho \rightarrow \infty$ the wastage becomes negligible relative to the number of occupied locations.

The method of (2.8) can be used to tighten the bound of (2.9) on $E \max(S)$ by applying (2.8) for $K = \lfloor (\rho \log \rho)^{1/2} \rfloor$ instead of $K = \lfloor (1 + \epsilon)\rho \rfloor$. In fact we show in § 4 that

$$(2.12) \quad E(\max(S) - |S|) \leq c(\rho \log \rho)^{1/2}$$

for some constant c . For an even closer look at asymptotic behavior numerical calculations were worked out by using (2.3) directly in (2.8). A plot of $\log P(\max(S) > m)$

vs. $\log 1/\rho$ revealed a nearly linear graph and thus indicated that, as $\rho \rightarrow \infty$

$$(2.13) \quad E(\max(S) - |S|) \sim c'\rho^\alpha,$$

for constants α and c' . The numerical results showed that $\alpha \approx 0.42$.

In Fig. 1, the wastage rate, defined as $(E \max(S) - E|S|)/E|S|$, is plotted as a function of $1/E|S| = 1/\rho$. It shows that the wastage rate approaches 0 monotonically as $\rho \rightarrow \infty$ and $\rho \rightarrow 0$. What is more interesting, however, is the existence of a maximum clearly less than $1/3$ at $1/\rho = 0.40$ and the strictly monotone behavior on both sides of the maximum. Thus, $E \max(S)$ is at most $\frac{4}{3}$ times $E|S|$.

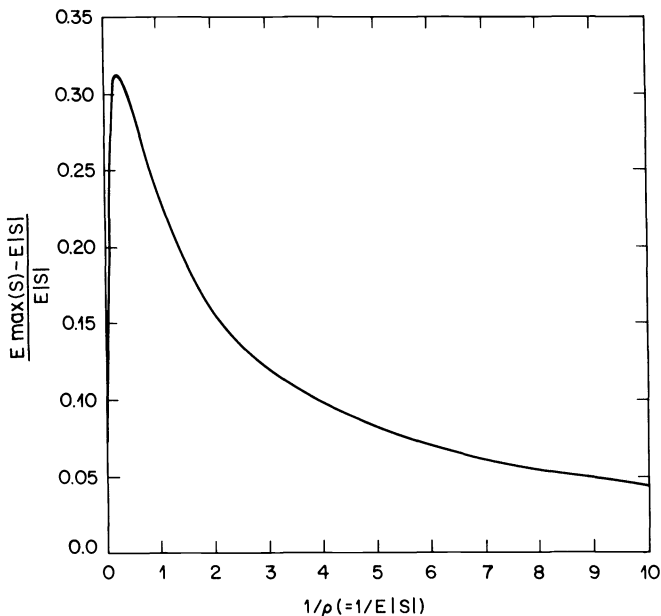


FIG. 1.

3. Derivation of (2.3). The derivation depends first on the use of an embedded two-dimensional Markov process and then on the use of a novel technique with generating functions. The embedded Markov process is $(|S_i \cap \{1, 2, \dots, m\}|, |S_i \cap \{m+1, m+2, \dots\}|)$ which is Markovian on states of integer pairs $(k, r), 0 \leq k \leq m, 0 \leq r$. The stationary distribution, for fixed $m \geq 0$,

$$(3.1) \quad \pi(k, r) = \pi(k, r; m) = P(|S \cap \{1, \dots, m\}| = k, |S \cap \{m+1, m+2, \dots\}| = r)$$

satisfies the equations

$$(3.2) \quad \pi(k, r)(\lambda + (k+r)\mu) = \pi(k-1, r)\lambda + \pi(k+1, r)(k+1)\mu + \pi(k, r+1)(r+1)\mu$$

for $0 \leq k < m, 0 \leq r$ where $\pi(-1, r) = 0$, and

$$(3.3) \quad \pi(m, r)(\lambda + (m+r)\mu) = \pi(m-1, r)\lambda + \pi(m, r-1)\lambda + \pi(m, r+1)(r+1)\mu$$

for $r \geq 0$, where $\pi(m, -1) = 0$. Introducing the generating function

$$(3.4) \quad F(x, y) = \sum_{k=0}^m \sum_{r=0}^{\infty} \pi(k, r)x^k y^r,$$

and $\rho = \lambda/\mu$, we obtain from (3.2) and (3.3)

$$(3.5) \quad \rho(1-x)F(x, y) + (x-1) \frac{\partial F}{\partial x}(x, y) + (y-1) \frac{\partial F}{\partial y}(x, y) = \rho x^m (y-x) f_m(y)$$

where

$$(3.6) \quad f_m(y) = \sum_{r=0}^{\infty} \pi(m, r) y^r = \frac{1}{m!} \frac{\partial^m}{\partial x^m} F(x, y).$$

In view of the fact that

$$(3.7) \quad P(\max(S) \leq m) = P(|S \cap \{m+1, m+2, \dots\}| = 0) = \sum_{k=0}^m \pi(k, 0) = F(1, 0),$$

we want to find $F(1, 0)$. The difficulty with solving (3.5) for F is that it is a partial differential equation (and, moreover, of degree m by (3.6)). However, by alternately setting $x = 1$ and $y = 1$ and repeatedly differentiating, we can reduce (3.5) to an infinite series of ordinary differential equations which we can solve.

First, setting $x = 1$ in (3.5) and noting that $F(1, 1) = 1$, we obtain

$$(3.8) \quad F(1, y) = 1 - \rho \int_y^1 f_m(u) du.$$

Next, setting $y = 1$ in (3.5), we obtain

$$(3.9) \quad F(x, 1) - \frac{1}{\rho} \frac{\partial F}{\partial x}(x, 1) = x^m f_m(1),$$

and solving this (ordinary) differential equation, with $F(1, 1) = 1$,

$$(3.10) \quad F(x, 1) = \rho e^{\rho x} \int_x^{\infty} e^{-\rho u} u^m du f_m(1).$$

Now, for each $n \geq 1$, differentiate (3.5) n times with respect to y and then set $y = 1$, to obtain

$$(3.11) \quad \begin{aligned} \rho(1-x) D_y^n F(x, 1) + (x-1) \frac{\partial}{\partial x} [D_y^n F(x, 1)] + n D_y^n F(x, 1) \\ = \rho x^m (1-x) f_m^{(n)}(1) + n \rho x^m f_m^{(n-1)}(1) \end{aligned}$$

where $D_y^n = \partial^n / \partial y^n$ and

$$(3.12) \quad f_m^{(n)}(y) = D_y^n f_m(y), \quad n = 0, 1, 2, \dots$$

Solving (3.11) for $D_y^n F(x, 1)$ we obtain

$$(3.13) \quad D_y^n F(x, 1) = -\frac{1}{(x-1)^n} \rho e^{\rho x} \int_x^{\infty} du e^{-\rho u} (u-1)^{n-1} [u^m (1-u) f_m^{(n)}(1) + n u^m f_m^{(n-1)}(1)],$$

$$n = 1, 2, \dots$$

Since $F(x, y)$ is a polynomial in x for any y , the right side of (3.13) must be finite at $x = 1$ and hence the integral in (3.13) must vanish at $x = 1$, namely,

$$(3.14) \quad \int_1^{\infty} du e^{-\rho u} (u-1)^{n-1} [u^m (1-u) f_m^{(n)}(1) + n u^m f_m^{(n-1)}(1)] = 0, \quad n \geq 1.$$

Rewriting (3.14),

$$(3.15) \quad \frac{f_m^{(n)}(1)}{n!} \int_1^\infty du e^{-\rho u} (u-1)^n u^m = \frac{f_m^{(n-1)}(1)}{(n-1)!} \int_1^\infty du e^{-\rho u} (u-1)^{n-1} u^m, \quad n \geq 1,$$

we observe that the function described by the left side of (3.15) does not depend on $n = 0, 1, 2, \dots$. Setting $n = 1$ in the right side of (3.15), then using (3.10) and the fact that $F(1, 1) = 1$, we have

$$(3.16) \quad \frac{f_m^{(n)}(1)}{n!} \int_1^\infty du e^{-\rho u} (u-1)^n u^m = \frac{f_m^{(0)}(1)}{0!} \int_1^\infty du e^{-\rho u} u^m = \frac{e^{-\rho}}{\rho}.$$

Now, using Taylor's expansion about $y = 1$,

$$(3.17) \quad f_m(y) = \sum_{n=0}^\infty \frac{f_m^{(n)}(1)(y-1)^n}{n!}.$$

Since the coefficients $\pi(m, r)$ in (3.6) satisfy $\pi(m, r) \leq P(|S| \geq m+r) = O(1/r!)$ by (2.1), f_m is an entire function and the validity of (3.17) follows. Using (3.16) and (3.17), we have

$$(3.18) \quad f_m(y) = \frac{e^{-\rho}}{\rho} \sum_{n=0}^\infty (y-1)^n \frac{1}{\int_1^\infty du e^{-\rho u} (u-1)^n u^m}.$$

Then, from (3.8),

$$(3.19) \quad F(1, y) = 1 - e^{-\rho} \sum_{n=0}^\infty \frac{(y-1)^{n+1}}{(n+1)} \frac{1}{\int_1^\infty du e^{-\rho u} (u-1)^n u^m},$$

and setting $y = 0$ gives (2.3) because of (3.7).

We observe finally that the joint distribution of $\max(S)$ and $|S|$ can be obtained from $F(x, y)$, which is determined from (3.13) and (3.16) explicitly. Indeed

$$(3.20) \quad P(\max(S) \leq m, |S| \leq k) = \sum_{j=0}^k \pi(j, 0; m)$$

and $\pi(j, 0; m) = \pi(j, 0) = \partial^j / \partial x^j F(0, 0)$, which can be obtained from $F(x, y)$. The resulting expression for the joint distribution of $\max(S)$ and $|S|$ does not appear to have a simple form and is therefore omitted.

4. Derivation of (2.12). First, note that

$$(4.1) \quad \sum_{m=K}^\infty \frac{x^m}{m!} \leq \frac{x^K}{K!} \left(1 + \frac{x}{K} + \left(\frac{x}{K}\right)^2 + \dots \right) = \frac{x^K}{K!} \frac{K}{K-x}, \quad x > 0.$$

From (2.8) and (2.4) with $\rho = x$

$$(4.2) \quad E \max(S) \leq K + x \sum_{m=K}^\infty \frac{x^m}{m!} \frac{1}{\sum_{j=0}^m (x^j/j!)}.$$

Since $e^{-x} \sum_{j=0}^{\lfloor x \rfloor} (x^j/j!) \rightarrow \frac{1}{2}$ as $x \rightarrow \infty$ [4, p. 193], by setting $K = \lfloor x + y \rfloor$ we obtain, using (4.1) with Stirling's formula,

$$(4.3) \quad \begin{aligned} E \max(S) &\leq x + y + c_1 x e^{-x} \frac{x^{x+y}}{(x+y)^{1/2} (x+y)^{x+y} e^{-(x+y)}} \frac{x+y}{y} \\ &\leq x + y + \frac{c_1 (x+y)^{3/2} e^y}{y \exp[(x+y) \log(1+y/x)]} \\ &\leq x + y + \frac{c_1 (x+y)^{3/2} e^y}{y \exp(y + y^2/2x)} \end{aligned}$$

for an appropriate constant $c_1 > 0$. Then, setting $x = \rho$ and $y = c_1(\rho \log \rho)^{1/2}$, we have

$$(4.4) \quad E \max(S) \leq \rho + c_1(\rho \log \rho)^{1/2} + \frac{[\rho + c_1(\rho \log \rho)^{1/2}]^{3/2}}{(\rho \log \rho)^{1/2} \rho^{c_1^2/2}}.$$

Thus, by choosing c_1 appropriately, say $c_1 = (2 + \varepsilon)^{1/2}$, the last term of (4.4) can be made to vanish as $\rho \rightarrow \infty$. Finally, by modifying the constant of the second term of (4.4), we obtain (2.12).

5. Discussion. The main results of this paper have been the explicit form in (2.3) for the stationary distribution of $\max(S)$, and the asymptotic behavior of the wastage rate given by (2.11). Two principal conclusions concerning fragmentation are that the stationary expected total occupancy, $E \max(S)$, never exceeds about $\frac{4}{3}$ the minimum possible expectation, $E|S|$, and that the difference between the two is asymptotically $o(E|S|)$; i.e. for sufficiently heavy traffic the effects of fragmentation become unimportant. Although proved only for the first-fit rule, one would expect this asymptotic behavior of any rule that places newcomers, whenever possible, in an unoccupied location less than $\max(S_i)$.

We have considered several extensions to the basic model with the same objectives in each case: an explicit form for $P(\max(S) = m)$ in equilibrium and a characterization of the asymptotic wastage rate. While the first objective appears unattainable (and hence not treated in detail here), these extensions are given below as open problems, since the prospects for asymptotic results or easily computed numerical results may be much better.

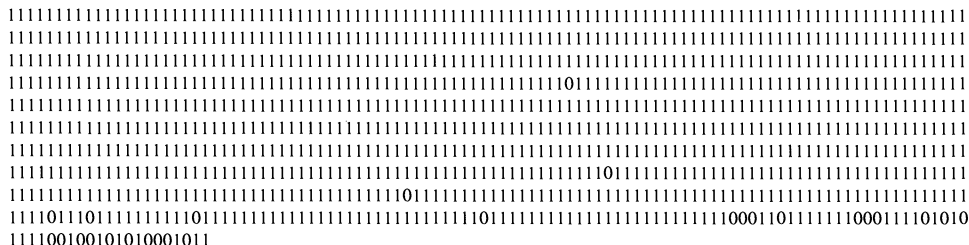
1. *Extensions to the delay ($M/M/\infty$) queue.* Suppose there is a fixed bound, M , on the number of locations: arrivals at times t when $|S_t| = \max(S_i) = M$ are simply lost. Our basic process $[S_t \cap \{1, 2, \dots, m\}, S_t \cap \{m+1, \dots, M\}]$ is now a bivariate finite Markov chain. Although explicit forms are apparently limited to matrix equations, conventional numerical methods can be employed to study the influence of the boundary. The results of preceding sections serve as descriptions of the asymptotic behavior, $M \rightarrow \infty$.

As another extension to the original model, suppose all arrivals now request a fixed number $l > 1$ of consecutive locations, and are accommodated by a first-fit policy; departures occur individually as before. It is easy to verify for this bulk-arrival queue that the process $[S_t \cap \{1, 2, \dots, m\}, S_t \cap \{m+1, m+2, \dots\}]$ no longer has the Markov property. Indeed, the types of fragmentation now possible are such that any Markov chain from which $P(\max(S_i) = m)$ could be calculated would have a much larger state space (essentially, it is now necessary to consider ordered subsets).

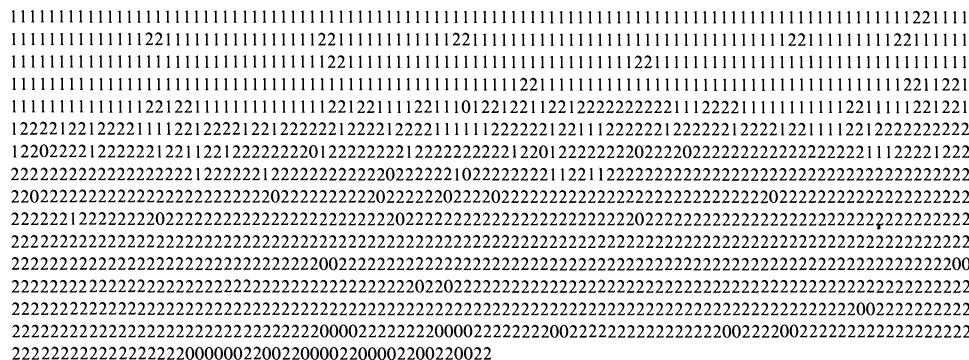
It is also interesting to question whether the wastage rate is still asymptotically negligible in this model. In contrast to the $l = 1$ case it is now possible for $\max(S_i)$ to increase even though holes exist. Equivalently, $\max(S_i) \leq \max_{\tau \leq i} |S_\tau|$ no longer holds for $l > 1$. It would be especially interesting to resolve this issue analytically in the general model where l is a random variable with a stationary distribution, $\{f_i\}$, and all locations of a given request are made available (i.e. depart) at the same instant. For if $E \max(S) - E|S|$ were not asymptotically $o(E|S|)$ as $\rho \rightarrow \infty$ then this would be further strong support for the concept of paging (or containerization in a general industrial setting).

To acquire further insight into this general problem, a number of simulations were run on first-order extensions of the basic model. First, request sizes were allowed to be either 1 or 2 locations and second, either 2 or 3 locations. In addition to the parameters λ and μ we have the parameter p denoting the stationary probability that a given request is for the smaller of the two sizes.

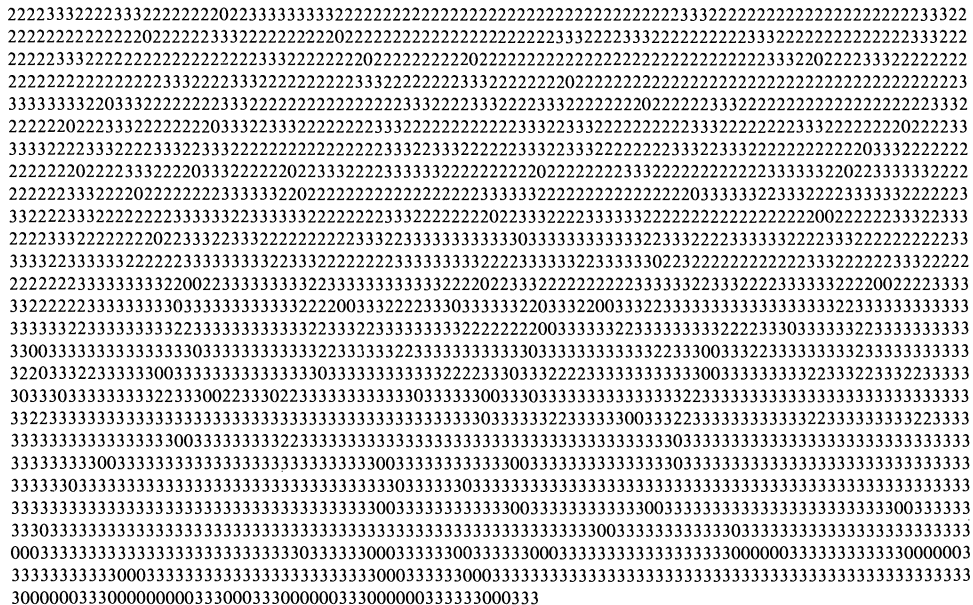
Typical results of these simulations are shown in the checkerboard patterns of Fig. 2, where locations 1-100 are in row 1 of the tableaux, locations 101-200 in row 2, etc. An integer k in some location of a tableau signifies that this location is one of a sequence of k allocated to a single request ($k=0$ denotes an available location). For



a. $\lambda = 1, \mu = 0.001$



b. $\lambda = 1, \mu = 0.001, p = 1/2$



c. $\lambda = 1, \mu = 0.001, p = 1/2$

FIG. 2. Storage patterns under heavy loading.

comparison with our earlier analysis we have also shown results for the basic model having requests of size 1 only.

Figure 2a is consistent with our earlier asymptotic analysis in that the number of available locations is small relative to the number of occupied locations when ρ is large ($\rho = \lambda/\mu = 1000$ in the figure). As expected from a first-fit rule the available locations tend to be higher numbered ones.

In Fig. 2b both of these effects are also illustrated for the system with two request sizes, 1 and 2, in spite of the greater fragmentation stemming from the fact that holes with a single location can be used only by requests for a single location. Accordingly, the first-fit rule justifies another obvious feature of the pattern, viz. that requests of size 1 tend to be concentrated in the smaller numbered locations.

The results from which Fig. 2c was selected also indicate that available locations tend to be higher numbered and smaller allocations tend toward the smaller numbers when request sizes are limited to 2 and 3. On the other hand, fragmentation is somewhat worse in this system because of the occurrence of poor fits, e.g. a request of size 2 allocated the first two locations of a hole consisting of three available locations creates an unusable hole of one location. Thus, the above effects are somewhat less pronounced than in Fig. 2b.

Motivated by these limited simulations, the authors quite recently conducted much more extensive tests of asymptotic behavior. These simulation results (to appear in [2]) suggest strongly that our earlier conjecture in fact holds, i.e. first-fit is asymptotically optimal in the sense that $E \max(S) - E|S| = o(E|S|)$ as $\rho \rightarrow \infty$ for any distribution $\{f_i\}$. Another recent extension [5] of the present paper stems from the apparent difficulty in proving this conjecture, and is based on a modified first-fit rule. Although the new rule is unlikely to be selected in favor of the simple first-fit rule in practice, it is proved rigorously in [5], with the help of the results in this paper, that the new rule is asymptotically optimal.

2. *Single server disciplines.* Suppose we have the original first-fit model except that only one occupied location can be served (at rate μ) at a time. In this case, of course, we need $\lambda < \mu$ for a stable system, i.e. for the existence of a stationary distribution. The preemptive, last-in-first-out service discipline, where the location being served is always $\max(S_t)$, is trivially solved, for in this case $|S_t|$ and $\max(S_t)$ are identical and no fragmentation occurs. The standard solution for the stationary $M/M/1$ queue applies.

Unfortunately, the more interesting first-in-first-out discipline suffers from the same general difficulties as the bulk-arrival $M/M/\infty$ queue described earlier. However, suppose we consider the *processor-sharing* discipline: when $|S_t| > 0$ every occupied location is being served, but at a rate $\mu/|S_t|$, so that the overall departure rate from a nonempty system is still μ . In this case, $[S_t \cap \{1, 2, \dots, m\}, S_t \cap \{m+1, m+2, \dots\}]$ retains the Markov property, but unfortunately, the partial differential equation for the generating function appears to be quite intractable. It is again apparent that one must resort to a numerical study.

Acknowledgments. We are grateful to J. C. Lagarias for assistance in obtaining the bound (2.12), and J. A. Morrison for interest and discussions on the asymptotics of the formula (2.3) as $\rho \rightarrow \infty$.

REFERENCES

- [1] E. G. COFFMAN, JR., *An introduction to combinatorial models of dynamic storage allocation*, SIAM Rev., 25 (1983), pp. 311-325.

- [2] E. G. COFFMAN, JR., T. T. KADOTA AND L. A. SHEPP, *On the asymptotic optimality of first-fit storage allocation*, IEEE Trans. Software Engineering, to appear.
- [3] E. G. COFFMAN, JR. AND A. C. MCKELLAR, *On the motion of an unbounded, Markov queue in random access storage*, IEEE Trans. Comput., June, 1968.
- [4] W. FELLER, *An Introduction to Probability Theory and Its Applications*, Vol. 1, 2nd Edition, John Wiley, New York, 1950.
- [5] T. T. KADOTA, L. A. SHEPP AND J. ZIV, *Asymptotic efficiency of dynamic storage systems*, to appear.
- [6] D. E. KNUTH, *Fundamental Algorithms*, Vol. 1, 2nd edition, Addison-Wesley, 1973, Reading, MA, Section 2.5.
- [7] F. SPITZER, *Interaction of Markov processes*, Advances in Math, (1970) pp. 246–290.

SEQUENTIAL MACHINE CHARACTERIZATIONS OF TRELIS AND CELLULAR AUTOMATA AND APPLICATIONS*

OSCAR H. IBARRA†, SAM M. KIM‡ AND SHLOMO MORAN§

Abstract. We look at a simple, but general model, of a systolic system called a trellis automaton (TA). A TA is equivalent in computational power to a one-dimensional unbounded cellular automaton (CA), a model of parallel computation which has been studied extensively in the literature. Different varieties of TA's are equivalent to corresponding variations of CA's. We present, for the first time, sequential machine characterizations of TA's (CA's). The sequential machines are useful and powerful tools for investigating properties of TA's (CA's). They are easy to program because, unlike the parallel models, one does not have to deal with the problem of synchronization. Several applications are given. In particular, we prove a new speed-up theorem which is stronger than what has previously been shown.

Key words. sequential machine, Turing machine, systolic system, trellis automaton, cellular automaton

1. Introduction. In recent years, computer science has devoted much attention to problems related to highly structured computing systems and their potential applications. Current VLSI technology requires uniformity and regularity in both the processing elements and their integration on a given chip. These requirements naturally lead one to conceive of a multiprocessor system with a large number of relatively simple and uniform processors interconnected in a regular pattern. A simple model of such a system is the systolic array automaton. It is known that homogeneous systolic arrays with simple interconnection networks (e.g., tree-structured, linear, orthogonal and hexagonally-interconnected) can efficiently solve nontrivial problems such as matrix manipulations, sorting, searching and language recognition problems [3], [5]-[7], [13], [16]-[19]. Analyzing the power of a systolic system is not so easy; it requires us to think about a large number of synchronized processes. Not only is it usually hard to find an algorithm to solve a problem on such a system, but showing the correctness of a given algorithm is also quite difficult. Hence, it is useful to have characterizations of these systems in terms of sequential machines where we do not have to deal with the problem of synchronization, thus making programming easier.

In [6], [7] a very simple model of a systolic trellis automaton, called a triangular trellis automaton, was introduced and studied (see also [5]). This model was characterized in terms of a simple type of Turing machine in [13]. This characterization turned out to be a very useful tool for investigating the properties of triangular trellis automata. The characterization was used to prove new results as well as give simpler proofs of known results concerning the trellis model.

In this paper we look at a more general model of a trellis automaton which we call a systolic trellis automaton (TA). A TA (see Fig. 1) is an infinite planar array of simple processors of combinational logic with unit propagation delay between neighboring processors (nodes). The TA is used as a language recognizer. An input string $a_1 a_2 \cdots a_n$ applied to n consecutive terminals is accepted if and only if some node on the accepting column generates an accepting symbol. The least number of rows needed to accept the string is the time complexity of the TA on the given string (see Fig. 1).

* Received by the editors October 4, 1983, and in final form February 17, 1984. This research was supported in part by the National Science Foundation under Grants MCS81-02853 and MCS83-04756.

† Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455.

‡ Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12181.

§ Department of Computer Science, Technion, Haifa, Israel.

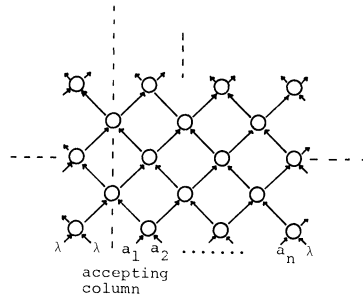


FIG. 1

A TA is equivalent to a one-dimensional unbounded cellular automaton (CA) [1], [8], [9], [10], [15], [22], [23]. Moreover, variations of a TA are equivalent to corresponding variations of a CA (e.g., bounded, one-way, real-time, etc.) [1], [2], [8], [9], [10], [15], [22], [23].

In this paper we characterize TA's and, hence, also CA's in terms of full scan Turing machines (STM's). An STM is a restricted type of on-line Turing machine with a single worktape. Using the characterization, we prove the following speed-up theorem for TA's (CA's): Let $R(n)$ be any function. Then any TA (CA) with time complexity $n + R(n)$ can be converted to one with time complexity $n + R(n)/k$ for any positive integer k . This result is stronger than what has previously been shown. For example, the speed-up for CA's given in [22] is from $n + R(n)$ to $n + (n + R(n))/k$. This is because the speed-up in [22] uses the standard technique of "packing" the input symbols and simulation. Packing involves synchronization (using the firing squad algorithm [24]) which takes extra time. Thus, the speed-up in [22] is not useful when $R(n)$ is a slow growing function. Our result shows that it is possible to have a speed-up from $n + R(n)$ to $n + R(n)/k$ for any $R(n)$. This strong speed-up result is useful because there are languages accepted by CA's (TA's) with time complexity close to real-time, but which do not seem to be recognizable in real-time. For example, we can show, using the characterization, that the language $L_1 = \{x | x \text{ in } \{0, 1\}^+, \text{ number of 1's in } x = \text{number of 1's in the binary representation of the length of } x\}$ can be accepted by a TA(CA) in time $n + \log(n)/k$ for any $k \geq 1$. However, it seems that L_1 cannot be recognized in less time. Actually, we are able to show that there is a language accepted by a TA(CA) with time complexity quite close to real-time, but does not seem to be recognizable in real-time. The language is $L_2 = \{c^{\lceil \log_2 |x_i| \rceil} x_1 c x_2 \# \dots \# x_k | x_i \text{ in } \{0, 1\}^+, |x_{i+1}| = \lceil \log_2 |x_i| \rceil, |x_k| = 1, \text{ the number of 1's in the binary expansion of } k \text{ is equal to the number of 1's in } x_1\}$. L_2 can be accepted by a TA (CA) in time $n + \log^*(n)$, where $\log^*(n)$ is the number of times we must take logarithms base 2 of n to get to 1 or below. Direct construction of the CA's (TA's) to accept languages L_1 and L_2 is not very easy. In fact, there is no example in the literature of a language which can be accepted by a CA in close to real-time, but which does not appear to be recognizable in real-time.

The STM characterization is also useful in establishing hierarchies of TA (CA) time complexity classes. For example, we show that there is a dense time hierarchy of the TA (CA) complexity classes in the polynomial range.

TA's can simulate arbitrary TM's. Hence, it would seem that all nontrivial decision problems are undecidable. For example, it follows that the problem, "Given a TA, A , an input, w , and a symbol, γ , will any processor in A output γ ?" is undecidable. The problem remains undecidable even when restricted to the processors on a given column. On the other hand, the problem is clearly decidable if we restrict it to the processors

of a given row (simply simulate A on input w until the given row is reached). Interestingly, we can show that the problem is decidable when restricted to the processors on a given diagonal. The proof consists of showing that for any diagonal d , the infinite sequence of symbols generated by the processors on the given diagonal is ultimately periodic. The lengths of the initial segment and the period are functions of d and the cardinality of the alphabet of the TA. There are, in fact, examples where the period length is a monotonically increasing function of d , even when the TA and the input string w are fixed.

The paper consists of 4 sections in addition to this section. Section 2 formally defines a TA and shows its equivalence to CA. STM is also defined in this section. Section 3 gives the characterization of TA's (CA's) in terms of STM's. The strong speed-up theorem is proved in this section, and some examples of STM constructions are given. Section 4 proves a dense time hierarchy of TA(CA) time complexity classes in the polynomial range. Finally, § 5 looks at some variations of TA's (CA's) and their characterizations.

2. TA's, CA's and STM's. We begin with a definition of a simple model of a systolic system called a systolic trellis automaton (TA). A TA is a generalization of the triangular trellis automaton studied in [6], [7], [13]. We shall see that TA's are equivalent (in some precise sense) to one-dimensional unbounded cellular automata [1], [8], [9], [10], [15], [22], [23].

A TA is a system $A = \langle \Gamma, \Sigma, \Delta, f \rangle$ consisting of an infinite planar array of identical processors (i.e., combinational logics) with unit time propagation delay between the processors as depicted in Fig. 2(a). In that figure, each node represents a processor which computes a partial function $f: \Gamma \times \Gamma \rightarrow \Gamma$, where Γ (which contains the blank symbol λ) is a finite operational alphabet. The function f has the property that $f(\lambda, \lambda) = \lambda$. Each processor computes f as shown in Fig. 2(b). An input to the TA is

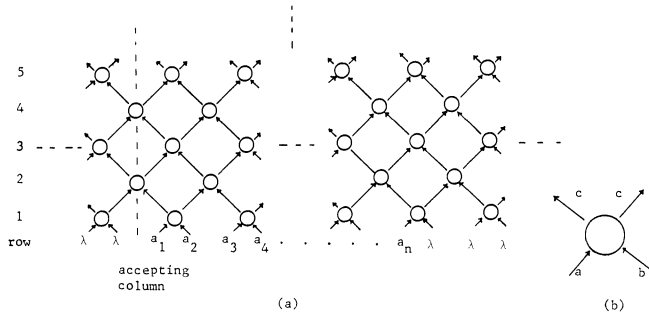


FIG. 2. (a) A TA. (b) A processor(node): $f(a, b) = c$.

a string $w = a_1 a_2 \dots a_n$, where a_i is in the input alphabet $\Sigma \subseteq \Gamma - \{\lambda\}$. The symbols a_1, a_2, \dots, a_n are applied to the input terminals of some $\lceil n/2 \rceil$ consecutive nodes of the base row (see Fig. 2(a)). All other input terminals of the base row get λ 's. (Note that if n is odd, then the right input terminal of the $\lceil n/2 \rceil$ th node gets λ .) The TA accepts w in time t if the processor in row t of the accepting column (the dotted column in Fig. 2(a)) outputs a symbol from the set of accepting symbols, $\Delta \subseteq \Gamma - \{\lambda\}$. $L(A) = \{x \mid x \text{ is accepted by } A \text{ in time } t \text{ for some } t\}$ is the language (or set) accepted by A . $L \subseteq \Sigma^+$ is a TA-language if there exists a TA, A , such that $L = L(A)$.

A TA has time complexity $T(n)$ if every input of length n that is accepted can be accepted in time less than or equal to $T(n)$. Clearly, $T(n) \geq 2 * \lceil n/2 \rceil$. (If $T(n) < n$, then some suffix of the input cannot affect the outcome of the computation.)

We can define a different input mode for the TA's such that the input string is applied on n consecutive base nodes as shown in Fig. 3. Call this model MTA. One can show that a language L is accepted by a TA in time $2 * \lceil n/2 \rceil + R(n)$, $R(n) \geq 0$, if and only if it can be accepted by an MTA in time $2(n + R(n))$. An MTA is essentially a time-space diagram for a one-dimensional unbounded cellular automaton (CA, for

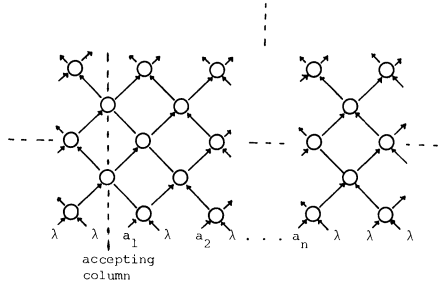


FIG. 3. An MTA.

short) [1], [8], [9], [10], [15], [22], [23]. A CA is shown in Fig. 4(a). Each node represents the same sequential machine. The inputs a_1, a_2, \dots, a_n in Σ are applied in parallel at time 0 to the input terminals of some n consecutive nodes. The input terminals of all other nodes, those that do not receive an input symbol from Σ , get λ 's. After time 0, the input terminals of all the nodes get λ 's. At each time unit, a node sends a symbol (signal) to its left and right neighbor (see Fig. 4(b)). The sequential machine representing a node is specified by a transition function $\delta: \Gamma \times (\Sigma \cup \{\lambda\}) \times Q \times \Gamma \rightarrow \Gamma \times Q \times \Gamma$, where Γ , which contains λ , is the communication alphabet, Σ is the input alphabet, and Q is the state set. The transition $\delta(Z_L, a, q, Z_R) = (Z'_L, q', Z'_R)$ (see Fig. 4(c)) means that if a node receives symbols Z_L and Z_R at time t from its left and right neighbors, respectively, and it is in state q with symbol a applied to its external input terminal, then, at time $t + 1$, the node enters state q' and sends symbols Z'_L and Z'_R to its left and right neighbors, respectively. The transition function δ satisfies the condition: $\delta(\lambda, \lambda, q_0, \lambda) = (\lambda, q_0, \lambda)$. The input $a_1 \dots a_n$ is accepted by the CA within time $T(n)$ if the node which gets input symbol a_1 (circled twice in Fig. 4(a)) enters an accepting state after at most $T(n)$ steps.

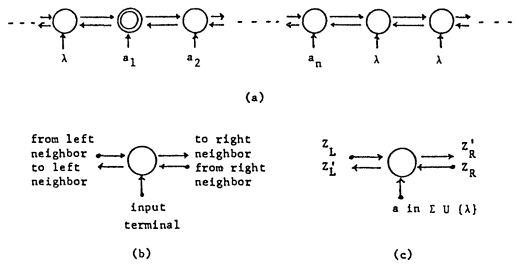


FIG. 4. (a) CA. (b) The input/output terminals. (c) Signal notation.

It can easily be shown that a language L is accepted by an MTA in time $2(n + R(n))$, $R(n) \geq 0$, if and only if it can be accepted by a CA in time $n + R(n)$. Hence, we have PROPOSITION 1. Let $R(n) \geq 0$. A language L is accepted by a TA in time $2 * \lceil n/2 \rceil + R(n)$ if and only if it can be accepted by a CA in time $n + R(n)$.

It can be shown that the TA's and CA's are time-wise equivalent to the "one-sided" TA and CA shown in Figs 5(a) and 5(b), respectively.

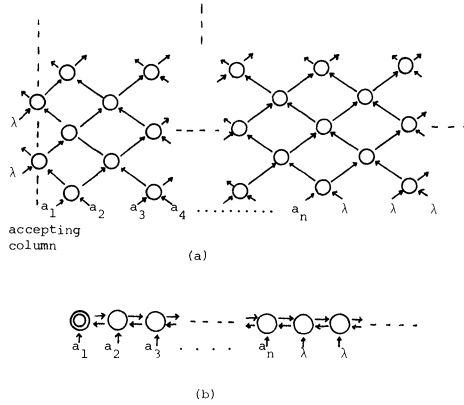


FIG. 5. Equivalent forms of TA and CA.

Important convention. For notational convenience, when dealing with TA's, we sometimes write $2 * \lceil n/2 \rceil + R(n)$ as $n + R(n)$.

The TA(CA) can be characterized by a restricted type of on-line Turing machine with a single worktape called a full scan Turing machine (STM). An STM is a device $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \Delta \rangle$, where Q, Σ, Γ are the set of states, input alphabet and tape alphabet, respectively (see Fig. 6). Γ contains two special symbols $\$$ and λ (for blank).

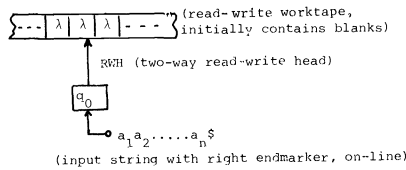


FIG. 6. An STM.

$\$$ which is not in Σ is used as an end marker for the input. q_0 is the start state, $\Delta \subseteq \Gamma - \{\$, \lambda\}$ is the accepting tape alphabet, and δ is a partial function $\delta: Q \times (\Sigma \cup \{\$, \epsilon\}) \times \Gamma \rightarrow Q \times (\Gamma - \{\lambda\}) \times \{-1, +1\}$. Thus M does not write blanks. δ is restricted as follows: Suppose $\delta(q, a, Z_1) = (p, Z_2, d)$. Then

- (1) If $q = q_0$ and $Z_1 = \lambda$, then a is in $\Sigma \cup \{\$\}$, $p \neq q_0$, $d = -1$ and if $a = \$$ then $Z_2 = \$$ else $Z_2 \neq \$$.
- (2) If $q \neq q_0$ and $Z_1 \neq \lambda$, then $a = \epsilon$, $p \neq q_0$, $Z_2 \neq \$$, $d = -1$.
- (3) If $q \neq q_0$ and $Z_1 = \lambda$, then $a = \epsilon$, $p = q_0$, $Z_2 \neq \$$, $d = +1$.
- (4) If $q = q_0$, $Z_1 \neq \lambda$ and $Z_1 \neq \$$, then $a = \epsilon$, $p = q_0$, $Z_2 = Z_1$, $d = +1$.
- (5) If $q = q_0$ and $Z_1 = \$$, then $a = \epsilon$, $p \neq q_0$, $Z_2 = \$$, $d = -1$.

Restrictions (1)-(5) mean that M 's read-write head (RWH) operates as follows: The RWH makes alternative sweeps on the worktape (right-to-left and left-to-right between λ 's). During the left-to-right sweep, the machine remains in a distinguished state q_0 and does not change the tape contents. However, during the right-to-left sweep, the machine can change state (except into q_0) and can change the tape contents. (The RWH does not write λ 's.) An input symbol is read if and only if the RWH is on λ and the machine is in state q_0 .

The machine starts by making a right-to-left sweep (see Fig. 7). Thus M , in state q_0 with λ on the worktape reads a_1 , and rewrites λ by a tape symbol, say $Z_1 (\neq \lambda)$, and moves left in some state $q (\neq q_0)$. Now in state q , reading λ on the worktape (on

ϵ input), the machine rewrites λ by a tape symbol, say $Z_2 (\neq \lambda)$, and starts a left-to-right sweep in state q_0 until the RWH reads λ . Repeating the procedure, the machine reads a_2 , replaces λ with a tape symbol Z_3 and begins the next right-to-left sweep. When the machine reads the input end marker “\$”, it replaces λ with the distinguished symbol “\$” and the range of the sweep can no longer be extended to the right. It can, however, still be extended towards the left by one cell each time the machine takes a right-to-left sweep. Figure 7 shows a worktape profile of an STM for 3 complete (right-to-left and left-to-right) sweeps. An input string $w = a_1 a_2 \cdots a_n$, $n \geq 1$, each a_i in Σ , is accepted if M , when given the string $a_1 a_2 \cdots a_n \$$, writes an accepting symbol (i.e., a symbol in Δ) at any time after reading \$. $L(M)$ denotes the language (or set) accepted by M . $L \subseteq \Sigma^+$ is an STM-language if there exists an STM M such that $L = (M)$.

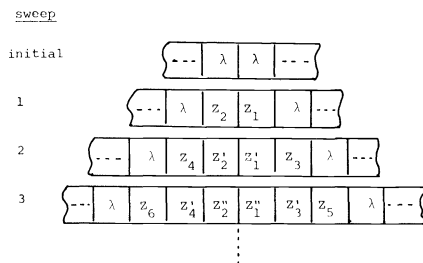


FIG. 7. The worktape profile from 3 full sweeps of an STM.

The sweep complexity $S(n)$ of an STM on an input $a_1 a_2 \cdots a_n$ is the least number of complete sweeps needed to accept the input, where a complete sweep consists of a right-to-left and a left-to-right sweep. Clearly, $S(n) \geq n + 1$.

To simplify the proofs of the characterizations, it is convenient to introduce a generalization of an STM, called an r -STM. For a positive integer $r \geq 1$, an r -STM is an STM which reads r input symbols before it starts a right-to-left sweep. If the machine encounters \$ (the input end marker) while trying to read r input symbols, the machine writes \$ on the worktape and makes a right-to-left sweep. After the machine writes \$, it operates like an STM (i.e., repeats the sweeps extending the range only to the left). By definition an STM is a 1-STM. The sweep complexity $S(n)$ of an r -STM is defined in the same way as in an STM. Clearly, $S(n) \geq \lfloor n/r \rfloor + 1$.

The models defined above can be made nondeterministic in the obvious way. For example, a TA, $A = \langle \Gamma, \Sigma, \Delta, f \rangle$, can be made nondeterministic by making the function f multivalued, i.e., $f: \Gamma \times \Gamma \rightarrow 2^\Gamma$. To simplify notation, the results presented in this paper are stated in terms of the deterministic models. However, the results easily extend to the nondeterministic models.

The relationship between STM's and r -STM's is formalized by the following theorem.

THEOREM 1. *A language L is accepted by an r -STM with sweep complexity $S_1(n) = \lfloor n/r \rfloor + R(n)$, $R(n) \geq 1$, if and only if it can be accepted by an STM with sweep complexity $S_2(n) = n + R(n)$.*

Proof. Suppose M_1 is an STM with sweep complexity $S_1(n) = n + R(n)$, $R(n) \geq 1$. We construct an equivalent r -STM M_2 with sweep complexity $S_2(n) = \lfloor n/r \rfloor + R(n)$ as follows. M_2 has a worktape with each cell divided into r subcells. Reading r input symbols at a time (except possibly the last read sequence), M_2 simulates r right-to-left sweeps of M_1 in one sweep until all the input symbols are read. Then each subsequent sweep is simulated by M_2 in one sweep. Notice that M_1 takes $n + 1$ sweeps to read the

whole input string (including $\$$), while M_2 takes $\lceil (n+1)/r \rceil$ sweeps. Hence, M_2 takes $\lceil (n+1)/r \rceil + (R(n) - 1) = \lfloor n/r \rfloor + R(n)$ sweeps for the simulation.

Now suppose M_1 is an r -STM with sweep complexity $S_1(n) = \lfloor n/r \rfloor + R(n)$, $R(n) \geq 1$. An equivalent STM, M_2 , with sweep complexity $n + R(n)$ can be constructed as follows. The left neighbor of the cell where M_2 's work head is initially positioned has two subcells, one of which is used for packing r input symbols (see Fig. 9 for the case $r=2$). In each right-to-left sweep, M_2 moves the input symbol read onto the packing cell and packs it. When M_2 sees that r input symbols are packed in the packing cell, the machine simulates M_1 's right-to-left sweep. After processing all of the input symbols, M_2 simulates each of the subsequent right-to-left sweeps of M_1 in one sweep. As an example for the case $r=2$, Fig. 9 shows M_2 's worktape profile when simulating M_1 's computation which is shown in Fig. 8. In Fig. 9, $\#$'s are written for blanks. (Recall that, by definition, an STM does not write λ .) Clearly, M_2 takes $n + R(n)$ sweeps if M_1 makes $\lfloor n/r \rfloor + R(n)$ sweeps. \square

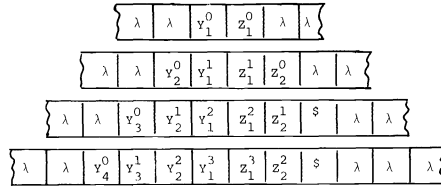


FIG. 8. A computation profile of M_1 ($r=2$) on input $a_1a_2a_3a_4a_5\$$.

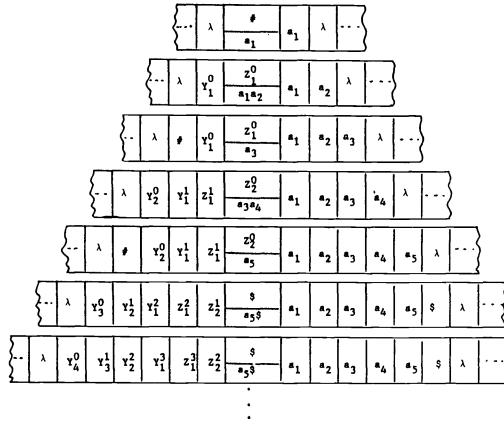


FIG. 9. The simulation profile of M_2 .

In § 3 we shall characterize TA's (CA's) in terms of STM's. Some well-known variations of TA's (CA's) [1], [2], [8], [9], [10], [15], [22], [23] can also be characterized in terms of corresponding variations of STM's. We give a few examples in § 5.

3. Speed-up and characterization theorems. In this section, we prove that TA's (CA's) and STM's are equivalent in computational power. Before we present the characterization, we prove a speed-up theorem for STM's. The speed-up theorem can be used to prove a similar speed-up theorem for TA's (CA's) which is stronger than what has been shown previously.

THEOREM 2. *Let M_1 be an STM with sweep complexity $S_1(n) = (n+1) + R(n)$, $R(n) \geq 0$. Then for any positive integer k , we can construct an equivalent STM M_2 with sweep complexity $S_2(n) = (n+1) + R(n)/k$.¹*

¹ $R(n)/k$ means $\lfloor R(n)/k \rfloor$.

Proof. Clearly, it is sufficient to prove the case $k = 2$. M_2 has a 2-track worktape. M_2 simulates M_1 faithfully (i.e., sweep by sweep) until it reads the input end marker $\$$. Then M_2 simulates 2 sweeps of M_1 in one sweep. For example, suppose M_1 on input $a_1a_2\$$, has the computation profile shown in Fig. 10. Then M_2 's profile will be as shown in Fig. 11, where the circled symbols are written on one cell. \square

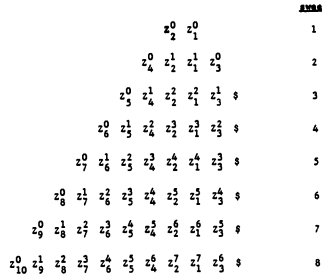


FIG. 10. The computation profile of M_1 on input $a_1a_2\$$.

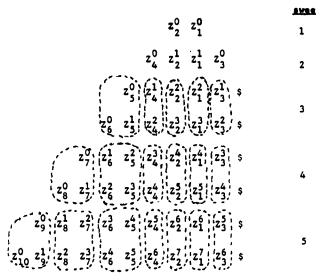


FIG. 11. The simulation profile of M_2 .

COROLLARY 1. Any STM with sweep complexity $n + c$, where c is some fixed constant, can be converted to one with sweep complexity $n + 1$.

Remark 1. If an STM has sweep complexity $n + 1$, then we say it is real-time. In this case, we do not really need the input end marker $\$$. (When making a right-to-left sweep after reading an input symbol a , the machine can also simulate, in parallel, the computation that would take place if the symbol to the right of a is $\$$.) Thus, a real-time STM with end marker $\$$ having sweep complexity $n + 1$ is equivalent to an STM without end marker having sweep complexity n .

Next, we characterize TA's (CA's) in terms of STM's. We shall see that the characterization is very useful in studying the properties of TA's (CA's).

THEOREM 3. Let $R(n) \geq 0$. A language L is accepted by a TA (CA) in time $T(n) = n + R(n)$ if and only if it can be accepted by an STM with sweep complexity $S(n) = (n + 1) + R(n)$.

Proof. By Proposition 1, we need only prove the equivalence of TA's and STM's. The constructions are given in Lemmas 1 and 2 below. \square

LEMMA 1. Let A be a TA with time complexity $T(n) = 2 * \lceil n/2 \rceil + R(n)$, $R(n) \geq 0$. We can construct an equivalent STM M with sweep complexity $S(n) = (n + 1) + R(n)$.

Proof. We shall only illustrate the construction by example. The formal construction is left to the reader.

Consider a computation of the TA, A , on an input $a_1a_2a_3a_4a_5$ as shown in Fig. 12, where we assume that the nodes output some results of their computations (the outputs are not shown in the figure to simplify the illustration). Construct a 2-STM, M' , which simulates A as shown in Fig. 13. Each cell in the tape profile lists the

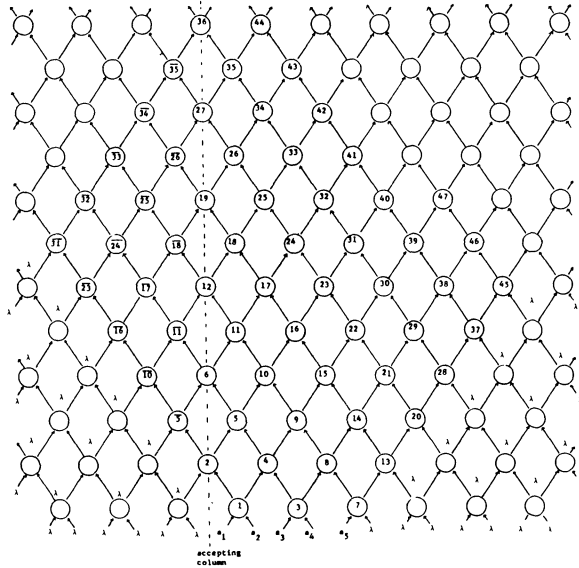


FIG. 12. The computation profile of A .

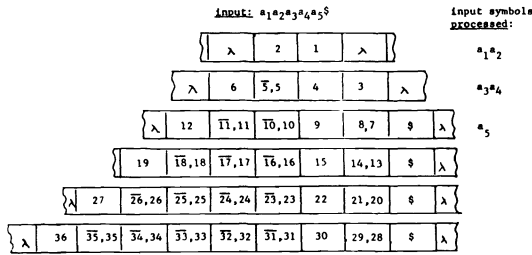


FIG. 13. The simulation profile of M' .

identification numbers of the node or nodes that it simulates. (Actually, the node number would be replaced by the output of the node.) In this figure, M' simulates A node by node in the order 1, 2, 3, 4, (5, 5), 6, 7, 8, 9, (10, 10), (11, 11), 12, \dots . Notice that nodes \bar{n} and n are located at symmetric points with respect to the accepting column, and M' simulates them at the same time. If the length of the input is odd, the last input symbol, a_m , will be read with \$. In this case, the machine writes \$ as defined, carries a_n to the left neighboring cell, creates a new subcell and writes the value $f(a_m, \lambda)$ on it. This operation corresponds to the simulation of node 7 in Fig. 12 (see (8, 7) from the third sweep in Fig. 13). From now on, this subcell is considered the first cell to the left of the \$ that was written when the input end marker was read. M' simulates the accepting node on row i at the end of i th sweep (in the order of the accepting nodes 2, 6, 12, 19, \dots in Fig. 12). Clearly, M' can simulate A in $\frac{1}{2}(2 * \lfloor n/2 \rfloor + R(n)) + 1 = \lfloor n/2 \rfloor + R(n)/2 + 1$ sweeps. By Theorem 1, we can construct an STM, M , which simulates M' in $n + R(n)/2 + 1 \leq (n + 1) + R(n)$ sweeps. \square

LEMMA 2. Let M be an STM with sweep complexity $S(n) = (n + 1) + R(n)$, $R(n) \geq 0$. We can construct an equivalent TA, A , with time complexity $T(n) = 2 * \lfloor n/2 \rfloor + R(n)$.

Proof. Using the speed-up theorem and Theorem 1, we can construct a 2-STM, M' , which simulates M in $\lfloor n/2 \rfloor + R(n)/2$ sweeps. Assume, without loss of generality,

that M' writes an accepting symbol only on λ at the end of the right-to-left sweep. (Given a 2-STM, M' , we can easily convert M' to a 2-STM, M'' , which has this property. Whenever M' writes an accepting symbol, M'' carries this information in its finite control to the end of the sweep and writes an accepting symbol.) We also assume that $\delta(q_0, \$, \lambda) = \delta(q_0, \epsilon, \$) = (p, \$, -1)$ for some fixed state p . For suppose M' has $\delta(q_0, \$, \lambda) = (s, \$, -1)$ and $\delta(q_0, \epsilon, \$) = (t, \$, -1)$. Then we can modify M' so that it enters a unique state p in both cases. When M' reads $\$,$ it enters p , and makes a right-to-left sweep. Entering the left neighboring cell in state p , M' marks the cell if it is not marked, and does the operation with state s . If the cell is marked, the machine does the operation with state t .

As with Lemma 1, we illustrate the construction of a TA from M' by an example. Suppose that M' with the input $a_1 a_2 a_3 a_4 a_5 a_6 a_7 \$$, has the computation profile shown in Fig. 14. Then the simulation profile of a TA, A , will be represented by Fig. 15 which shows only the information generated on the tape profile. To simplify notation we use states q_0, p and q , where q can be any state. Notice that the sequence of nodes $1, 2, 3, 4, \dots$ on the diagonal lines in Fig. 15 simulates the right-to-left sweeps of M' . It is easy to check that A has time complexity $2 * \lceil n/2 \rceil + R(n)$. \square

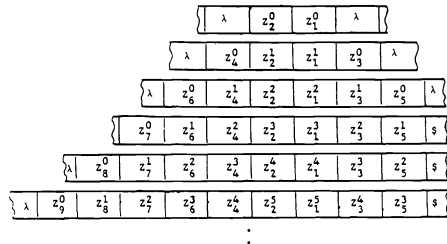


FIG. 14. The computation profile of M' on input $a_1 a_2 a_3 a_4 a_5 a_6 a_7 \$$.

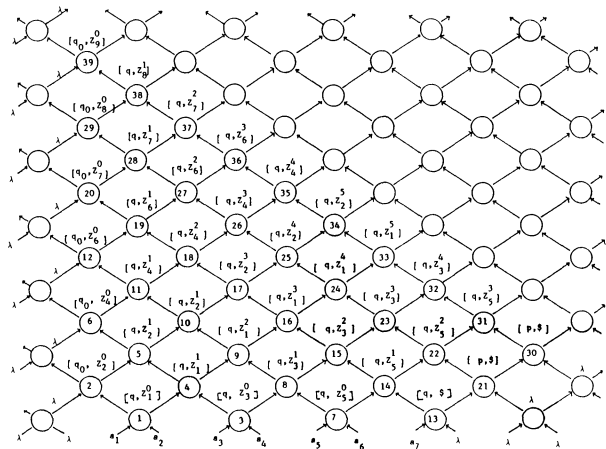


FIG. 15. The simulation profile of A .

From Theorems 2 and 3, we get the following corollary:

COROLLARY 2. Let A_1 be a TA (CA) with time complexity $T_1(n) = n + R(n)$, $R(n) \geq 0$. Then for any positive integer k , we can construct an equivalent TA (CA) A_2 with time complexity $T_2(n) = n + R(n)/k$. (Note that, since a TA accepts an input only on an even numbered row, $R(n)/k$ should be even.)

A TA (CA) with time complexity $T(n) = n$ is called a real-time TA (CA). From Theorem 3 and Corollary 1, we get the following result which has essentially been shown in [8] using a different proof technique.

COROLLARY 3. *Any TA (CA) with time complexity $n + c$, where c is some fixed constant, can be converted to a real-time TA (CA).*

Corollary 2 is stronger than what has been shown before. For example, the speed-up for CA's given in [22] is from $n + R(n)$ to $n + (n + R(n))/k$. This is because the speed-up in [22] uses the standard technique of "packing" the input symbols and simulation. Packing involves synchronization (using the firing squad algorithm [24]) which takes extra time. From Corollary 2, it is possible to have a speed-up for CA's from $n + R(n)$ to $n + R(n)/k$. This strong speed-up result is useful because there are languages accepted by CA's (TA's) with time complexity close to real-time, but which do not seem to be recognizable in real-time. For example, using the characterization and Corollary 2, we can conclude that, for any positive integer d , the languages in Examples 1 and 2 below are recognizable by CA's (TA's) in time

$$n + \overbrace{\log \cdots \log (n)}^k / d \quad \text{and} \quad n + \log^*(n) / d,$$

respectively, where $\log^*(n)$ is the number of times we must take logarithms base 2 of n to get to 1 or below.

Since an STM is a sequential device, it is relatively easy to program. The characterization theorem (Theorem 3) can be used to provide simple solutions to many language recognition problems concerning CA's (TA's). In fact, since we can easily convert an STM into an equivalent CA (TA), STM's can be used as programming tools when designing CA's (TA's). We look at two examples.

Example 1. Let k be any positive integer. Then the language $L_k = \{c^{\lceil \log_2 |x_1| \rceil} x_1 c x_2 \# \cdots \# x_k \mid x_i \in \{0, 1\}^+, |x_{i+1}| = \lceil \log_2 |x_i| \rceil, \text{the number of 1's in the binary expansion of } |x_k| = \text{the number of 1's in } x_k\}$ is in an STM-language with sweep complexity

$$S(n) = n + \overbrace{\log \log \cdots \log (n)}^k.$$

L_k can easily be accepted by an STM, M , operating as follows. While copying the input on its worktape, M does the following:

- (1) Reading x_i , $1 \leq i \leq k$, M builds a counter to the left of the separator to count the length of x_i . At the same time, for $2 \leq i \leq k$, the machine checks that the length of x_i is equal to the length of the binary representation of the length of x_{i-1} .
- (2) After reading $\$$ (input end marker), M checks that the number of 1's in the binary representation of $|x_k|$ is equal to the number of 1's in x_k .

All the required counting and length checking can be done during the first n sweeps, where n is the length of the input. To check that the number of 1's in the binary representation of x_k is equal to the number of 1's in x_k , M needs an additional $\log(|x_k|)$ sweeps. So M takes

$$n + \log(|x_k|) = n + \overbrace{\log \log \cdots \log (|x_1|)}^k \leq n + \overbrace{\log \log \cdots \log (n)}^k$$

sweeps.

Example 2. $L = \{c^{\lceil \log_2 |x_1| \rceil} x_1 c x_2 \# \cdots \# x_k \mid x_i \in \{0, 1\}^+, |x_{i+1}| = \lceil \log_2 |x_i| \rceil, |x_k| = 1, \text{the number of 1's in the binary expansion of } k \text{ is equal to the number of 1's in } x_1\}$ is in an STM-language with sweep complexity $S(n) = n + 2 \log^*(n)$.

The algorithm is basically the same as that for Example 1. After reading the input end marker $\$$, the STM, counting the number of separators (c and $\#$'s), constructs

the binary representation of k to the left of the separator c . Then the machine compares the number of 1's in x_1 and the number of 1's in the binary representation of k . If $|x_1| = m$, the machine needs $\log^*(m)$ sweeps to count k . The machine takes at most $\log(\log^*(m))$ sweeps (the length of the binary representation of k) to compare the number of 1's. Hence, the machine needs $n + \log^*(m) + \log(\log^*(m)) \leq n + 2 \log^*(n)$ sweeps.

The reader will observe that direct construction of CA's (TA's) to accept the languages in Examples 1 and 2 is not as easy. In fact, there is no example in the literature of a language which can be accepted by a CA in close to real-time, but which does not seem to be recognizable in real-time.

The next theorem shows the relationship between STM's and TM's. The first part has already been observed in [7].

THEOREM 4. (1) *If L is accepted by an STM with sweep complexity $S(n)$, then L can be accepted by a single-tape TM in time $O(S^2(n))$.*

(2) *If L is accepted by a single-tape TM in time $T(n)$, then L can be accepted by an STM with sweep complexity $O(T(n))$.*

Proof. Direct simulation of an STM proves (1). To prove (2), we construct an STM M_2 which simulates a single-tape TM, M_1 , by constructing the *ID*'s of M_1 . If we assume that M_1 begins its computation with its read-write head on the blank to the left of the input string, we can construct an STM, M_2 , which simulates M_1 in the following way: Given an input $a_1 a_2 \cdots a_n \$$, M_2 copies the input on its worktape (see Fig. 16(a)), and constructs M_1 's initial *ID* as shown in Fig. 16(b) by shifting the symbols in Fig. 16(a) to the left. From this point on, each time M_2 constructs M_1 's next *ID*, it shifts the whole *ID* one cell to the left, inserts a blank in the vacant cell, and moves the next state according to the M_1 's direction. This process is illustrated by the sequence of moves represented by Figs. 16(c)-(f). (In the figures the B 's represents the blanks of the TM tape. Note that, by definition, an STM does not write λ .) Obviously, M_2 takes $O(T(n))$ sweeps. \square

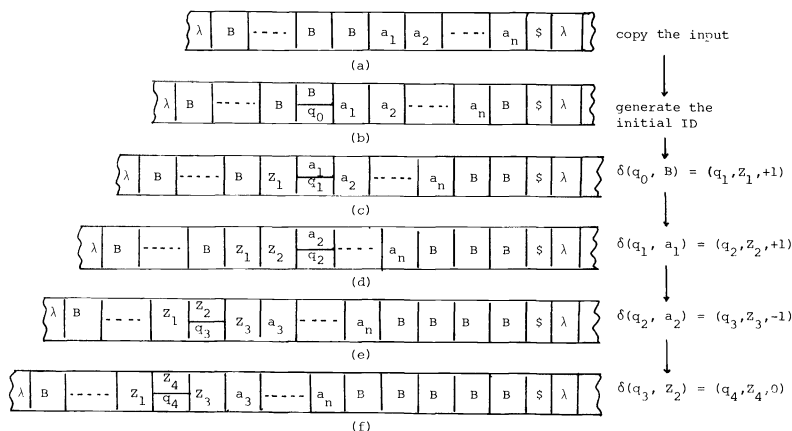


FIG. 16. The worktape of M_2 simulating M_1 .

It follows from Theorem 3 that Theorem 4 also holds when "STM" is replaced by "TA" ("CA"), and "sweep" by "time".

From Theorems 3 and 4 we see that TA's can simulate arbitrary single-tape TM's. Hence, it would seem that all nontrivial decision problems are undecidable. For example, it follows from the undecidability of the halting problem that the problem:

“Given a TA, A , a string, w , and a letter, Z_0 , in Γ , does any processor in A output Z_0 on input w ?” is undecidable. The problem becomes decidable, however, if we restrict it to the processors in a given row in A (this can be done by simulating A on input w until it reaches the appropriate row). On the other hand, the problem remains undecidable when restricted to the processors in a given column of A . (Recall that A accepts w if and only if some processor in column 0 outputs a symbol in Δ , see Fig. 17.) Interestingly, the problem is decidable when restricted to the processors in a given diagonal. This follows from an interesting property of the diagonals of a TA.

Notation. Rows, columns and diagonals are defined as in Fig. 17. The rows are numbered from 1 to ∞ , the bottom row being the first. The columns are numbered symmetrically from $-\infty$ to $+\infty$, on either side of the accepting column which is numbered 0. The diagonals (directed from the lower right to upper left) are numbered from $-\infty$ to $+\infty$ on either side of the diagonal containing the processor activated by the input symbol a_1 which is numbered 1.

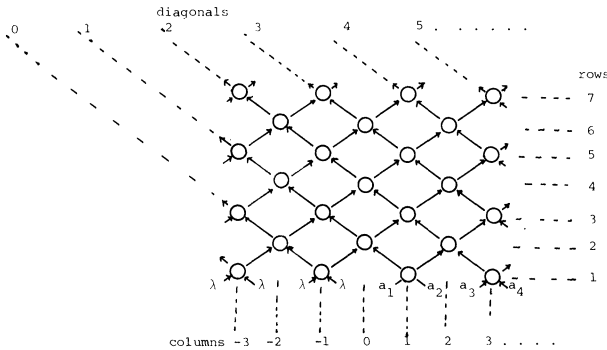


FIG. 17. Numbering of the rows, columns and diagonals of a TA.

In the rest of this section, A is some fixed TA, and $w = a_1 \cdots a_n$ is some fixed input string over Σ . Let i and j be given integers ($i \geq 1$). Then, $d_{j,i}$ is defined as the output of the processor on the i th row and j th diagonal of A on input w . $D_j = (d_{j,1}, d_{j,2}, \dots, d_{j,i}, \dots)$ is the sequence of outputs of the processors on the j th diagonal (if $j \leq 0$, then $D_j = (\lambda, \lambda, \dots)$).

DEFINITION. A sequence $(a_1, a_2, \dots, a_n, \dots)$ is *ultimately periodic with initial index i_0 and period p* if for some positive integers i_0 and p , $a_i = a_{i+p}$ for all $i \geq i_0$, and i_0 and p are the smallest positive integers which have this property.

The following lemma is easily shown by induction on j .

LEMMA 3. Let Γ , the alphabet of A , have k letters. Then, for all $j \geq 0$, D_j is ultimately periodic. The initial index of D_j is at most $k^j + 1$ and the period of D_j is at most k^j .

We can now use Lemma 3 to prove the following theorem.

THEOREM 5. The problem: “Given a TA, A , a string, w , a letter, Z_0 and a number, j , does any processor on the j th diagonal of A output Z_0 on input w ?” is decidable.

Proof. By Lemma 3, Z_0 is produced by one of the processors at the j th diagonal if and only if it is produced by a processor in the j th diagonal and i th row for some $i \leq 2k^j + 1$. Hence, we need only check the outputs of the first $2k^j + 1$ rows to decide the problem, and this can be done by a straightforward simulation. \square

Next, we give an example which shows that the periods of the D_i 's become arbitrarily long even for a very simple TA. Define the following TA:

$$\Gamma = \{0, 1, \lambda\}, \text{ for } a, b \text{ in } \{0, 1\}: f(a, b) = a + b \pmod{2}; f(a, \lambda) = f(a, 0); f(\lambda, b) = f(0, b).$$

Let the input be the one letter word “1”. Figure 18 shows part of the output from the active portion of the TA on this input. The labels (0’s and 1’s) on the nodes show the outputs from the nodes. This output can be constructed from the Pascal triangle by replacing the odd numbers in it by 1 and the even numbers by 0. Since the $(i + 1)$ st “diagonal” in the Pascal triangle is the infinite sequence $((i), \binom{i+1}{i}, \dots)$, the sequence D_i of the TA on input “1” has the following form:

$$D_i = \left(\overbrace{\lambda, \lambda, \dots, \lambda}^i, 1, \binom{i+1}{i} \bmod 2, \binom{i+2}{i} \bmod 2, \dots \right).$$

If we set $k = 2^i$, the length of the period of D_k is $2k$, as the following proposition shows.

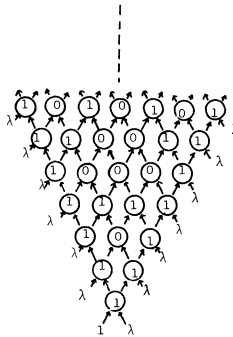


FIG. 18. A TA.

PROPOSITION 2. *Let $m \geq 0$ be given, and let $m_k = (m + 1)(m + 2) \cdots (m + k)$ (in particular, $0_k = k!$). Then for $k = 2^i$, 2^{k-1} divides m_k , and 2^k divides m_k if and only if $k \equiv m \pmod{2k} \leq 2k - 1$.*

Proof. Let k and m be given. Then among the numbers $m + 1, m + 2, \dots, m + k$, exactly $k/2$ are divisible by 2, exactly $k/4$ are divisible by 4, and, in general, $k/2^j$ are divisible by 2^j for $1 \leq j \leq i$. This implies that:

- (a) the largest d such that 2^d divides m_k is at least $k/2 + k/4 + \dots + 1 = k - 1$;
- (b) d is greater than $k - 1$ if and only if one of the numbers $m + 1, m + 2, \dots, m + k$ is divisible by $2^{i+1} = 2k$. (There may be at most one such number.)

The proposition follows. \square

To see that the length of the period of D_k is indeed $2k$, recall that the $(k + m)$ th element in D_k is $(m_k/k!) \pmod{2}$. From Proposition 2 we know that this number is 1 if $0 \leq m \pmod{2k} \leq k - 1$, and that it is 0 otherwise. Hence, the period of the k th diagonal consists of k 1’s followed by k 0’s.

4. Two hierarchy theorems. Let $STM(T(n))$ denote the class of languages accepted by STM’s with sweep complexity $T(n)$. Let $TA(T(n))$ and $CA(T(n))$ denote the corresponding classes for TA’s and CA’s. We shall prove two hierarchy theorems concerning $STM(T(n))$. The first follows rather easily from a known hierarchy result concerning TM time complexity classes and Theorem 4. The second result shows how a stronger theorem (dense hierarchy) can be obtained using translation. By Theorem 3 the results hold also for $TA(T(n))$ and $CA(T(n))$.

THEOREM 6. *$STM(T_2(n)) - STM(T_1(n)) \neq \phi$ if $T_2(n)$ is time constructible on a single-tape TM and*

$$\inf_{n \rightarrow \infty} \frac{T_1^2(n) \log^* T_1(n)}{T_2(n)} = 0,$$

where $\log^*(m)$ is the number of times we must take logarithms base 2 of m to get to 1 or below.

Proof. It is known [20] that if $T_2(n)$ is time constructible on a k -tape TM ($k \geq 1$) and $\inf_{n \rightarrow \infty} (T_1(n) \log^* T_1(n) / T_2(n)) = 0$, then there is a language accepted by a k -tape $T_2(n)$ time-bounded TM but by no k -tape $T_1(n)$ time-bounded TM. The theorem now follows from this result and Theorem 4. (A $T_1(n)$ sweep-bounded STM can be simulated by a 1-tape $T_1^2(n)$ time bounded TM; a 1-tape $T_1(n)$ -time bounded TM can be simulated by a $T_1(n)$ -sweep bounded STM.) \square

Next, we show how we can improve Theorem 6 using translation (or padding). Translational techniques have been used before in the study of complexity classes of TM's, formal languages, multihead automata, etc. (see e.g. [4], [11], [12], [21]). Our contribution here is the technique employed to prove a translation lemma. To illustrate the idea, we consider the polynomial-time complexity class, i.e., the class $\cup_{k \geq 1} \text{STM}(n^k)$. The translation lemma we need is:

LEMMA 4. *If $\text{STM}(T_1(n)) \subseteq \text{STM}(T_2(n))$, then $\text{STM}(T_1(2^{kn})) \subseteq \text{STM}(T_2(2^{kn}))$ for every positive integer k .*

Before we prove the lemma, we prove the following.

THEOREM 7. *If $r \geq 1$ and $\epsilon > 0$ then $\text{STM}(n^r) \subset \text{STM}(n^{r+\epsilon})$.*

Proof. The use of the translation lemma to prove this result is similar to the one described in [11].

Clearly, there are positive integers s and t such that $\text{STM}(n^r) \subseteq \text{STM}(n^{s/t})$ and $\text{STM}(n^{(s+1)/t}) \subseteq \text{STM}(n^{r+\epsilon})$. Thus, it suffices to show that $\text{STM}(n^{s/t}) \subset \text{STM}(n^{(s+1)/t})$. Suppose not. Then $\text{STM}(n^{(s+1)/t}) \subseteq \text{STM}(n^{s/t})$. By the translation lemma, with $k = (s+i)t$, $i = 0, 1, \dots, s$ we have $\text{STM}(2^{(s+1)(s+i)n}) \subseteq \text{STM}(2^{s(s+i)n})$. Now for $i = 1, 2, \dots, s$, $(s+1)(s+i-1) \geq s(s+i)$. Hence $\text{STM}(2^{s(s+i)n}) \subseteq \text{STM}(2^{(s+1)(s+i-1)n})$, for $i = 1, 2, \dots, s$. It follows that $\text{STM}(2^{(s+1)(s+s)n}) \subseteq \text{STM}(2^{s^2n})$, i.e., $\text{STM}(2^{(2s^2+2s)n}) \subseteq \text{STM}(2^{s^2n})$. But by Theorem 6, $\text{STM}(2^{s^2n}) \subset \text{STM}(2^{2s^2n}) \subseteq \text{STM}(2^{(2s^2+2s)n})$. We get a contradiction, and so the assumption that $\text{STM}(n^{(s+1)/t}) \subseteq \text{STM}(n^{s/t})$ is false. \square

We now give the proof of Lemma 4.

Proof of Lemma 4. We only prove the case when $k = 1$. The constructions generalize easily to arbitrary k . Let M_1 be a $T_1(2^n)$ sweep-bounded STM accepting a language L_1 . Let $\#$ be a new symbol. Define the language $L_2 = \{x\#^{2^n-n} \mid x \text{ in } L_1, |x| = n\}$. We claim that L_2 can be accepted by an STM, M_2 , which is $T_1(n)$ sweep-bounded. M_2 operates as follows when given input

$$a_1 a_2 \cdots a_n \overbrace{\# \cdots \#}^m \$:$$

M_2 sets up a binary counter to the left of the initial position of the RWH on a separate track of the worktape. M_2 reads the input and increments the counter for every symbol read while simultaneously simulating the actions of M_1 on $a_1 \cdots a_n \$$, $\#$ being interpreted as \$.

After reading

$$a_1 a_2 \cdots a_n \overbrace{\# \cdots \#}^m \$,$$

the worktape will look like that shown in Fig. 19 if and only if $m = 2^n - n$. If this is

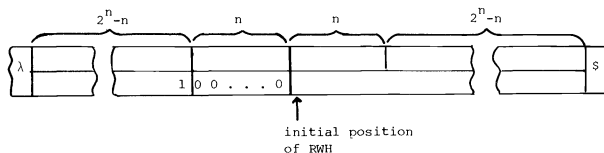


FIG. 19

the case, M_2 continues the simulation of M_1 , and accepts if and only if M_1 accepts. It is clear that L_2 is accepted by M_2 , and M_2 is $T_1(n)$ sweep-bounded. Now by hypothesis, $STM(T_1(n)) \subseteq STM(T_2(n))$. Hence, L_2 is accepted by some $T_2(n)$ sweep-bounded STM, M_3 . We now describe a $T_2(2^n)$ sweep-bounded STM, M_4 , which will accept the original language L_1 . M_4 when given $a_1 a_2 \cdots a_n \$$ simulates the computation of M_3 on input $a_1 a_2 \cdots a_n \#^{2^n-n} \$$ as follows: M_4 first reads $a_1 \cdots a_n$ and sets up its worktape as shown in Fig. 20. When $\$$ is read, the RWH writes $\$$, moves left and replaces a_n by \oplus , adds 1 to the binary counter on the second track (which is initially blank), and shifts $a_1 \cdots a_n$ one cell to the left. When a_1 crosses the boundary, it is processed, i.e., M_4 simulates what M_3 would have done on input a_1 . For example, if M_3 with input a_1 writes $Y_1 Z_1$ on its worktape, then M_4 's worktape would look like Fig. 21. Then M_4 makes another right-to-left sweep to modify the worktape to that

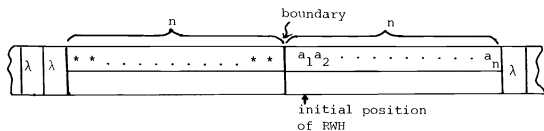


FIG. 20

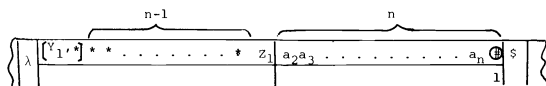


FIG. 21

shown in Fig 22. The next right-to-left sweep shifts $a_2 \cdots a_n \#$ to the left and when a_2 crosses the boundary it is processed. The worktape after this sweep would look like Fig. 23. And after another right-to-left sweep would be changed as shown in Fig. 24.

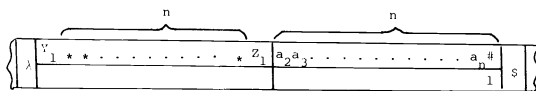


FIG. 22

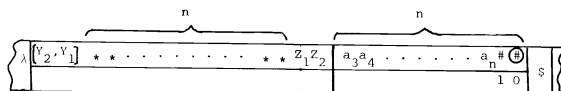


FIG. 23

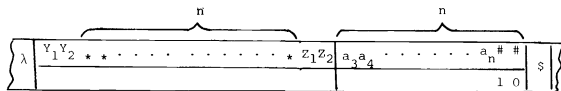


FIG. 24

(For notational convenience, in Fig. 24, Z_1 and Y_1 were not changed, although, in general, they would be different after a_2 was processed.) The computation described above (2 sweeps for every symbol crossing the boundary) continues until the counter on the second track reaches the value shown in Fig. 25. When this happens, the machine computes just like before, but when the RWH sees a "1" to the left of the boundary, it knows that it has already processed exactly $(2^n - n) \#$'s, and the symbols to the left

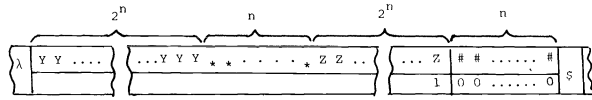


FIG. 25

of the boundary are no longer shifted. Note, however, that the machine needs to write a “dummy” symbol on the left because it still has to make 2 sweeps (the second sweep rewrites \otimes by $\#$). Note also that the contents of the n -bit counter to the right of the boundary will keep on changing, but will have no effect in subsequent computations. Clearly, M_4 has sweep complexity $2T_2(2^n)$. By the speed-up theorem, M_4 can be converted to a STM with sweep complexity $T_2(2^n)$. \square

Theorems 6 and 7, of course, translate directly to TA ($T(n)$) and CA ($T(n)$) by Theorem 3. The reader should note that a direct proof of Lemma 4 for TA’s or CA’s would be much more difficult.

5. Variations of TA’s (CA’s) and STM’s. There are several variations of an MTA (CA) that can be characterized in terms of corresponding variations of an STM. We look at four examples here. Other examples can be found in [14].

First we consider the case when the MTA (CA) is bounded. Figure 26 shows a bounded MTA (BMTA, for short). Thus, a BMTA is an MTA in which the “width” (space) of the computation is bounded by the length of the input (dotted lines in the figure). A BMTA is essentially a time-space diagram of a bounded cellular automaton (BCA) [1], [8], [9], [10], [15], [22], [23] like that shown in Fig. 27.

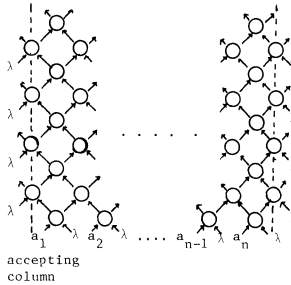


FIG. 26. A BMTA.

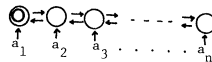


FIG. 27. A BCA.

One can easily show that a language is accepted by a BMTA in time $2(n + R(n))$, $R(n) \geq 0$, if and only if it can be accepted by a BCA in time $n + R(n)$. A BMTA (BCA) can be characterized by a bounded version of an STM, called a BSTM. A BSTM operates like an STM except that when it reads the input end marker “\$”, it writes \$ on the worktape which is propagated to the left by one cell each time the machine takes a right-to-left sweep. The computation profile of a BSTM on an input $a_1 a_2 a_3 \dots a_n \$$ is shown in Fig. 28. As in Theorem 3, we can show that L is accepted by a BCA in time $n + R(n)$, $R(n) \geq 0$, if and only if it can be accepted by a BSTM with sweep complexity $(n + 1) + R(n)$. Theorem 2 and Corollaries 2 and 3 (speed-up’s) also apply to BSTM’s and BCA’s.

Example 3. $L = \{xx \mid x \text{ in } \Sigma^+\}$ can be accepted by a BSTM with sweep complexity $S(n) = n + 1$. Hence L can be accepted by a BCA in real-time.

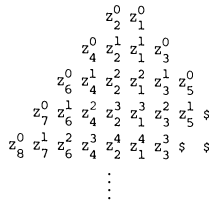


FIG. 28. Computation profile of a BSTM on input $a_1 a_2 a_3 \$.$

We can construct a BSTM, M , which accepts L as follows. Copying the input on its two-track worktape, M folds the input string into halves and compares them. Figure 29 illustrates this operation, where the B 's denote blanks. It follows from the characterization theorem that L can be accepted in real time by a BCA. Again, direct construction of a BCA is not as easy.

Next, consider the triangularly shaped MTA's shown in Fig. 30. In that figure, TA1 and TA2 get the input symbols a_1, a_2, \dots, a_n at time 0 and TA3 and TA4 get the input symbol a_i at time $i - 1$. All the triangular trellises accept the input if and only

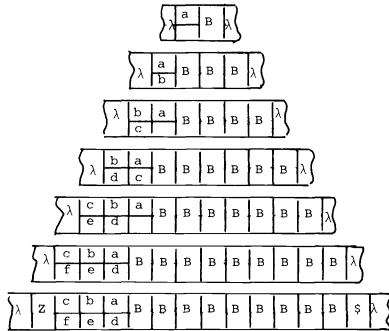


FIG. 29. The worktape profile of M .

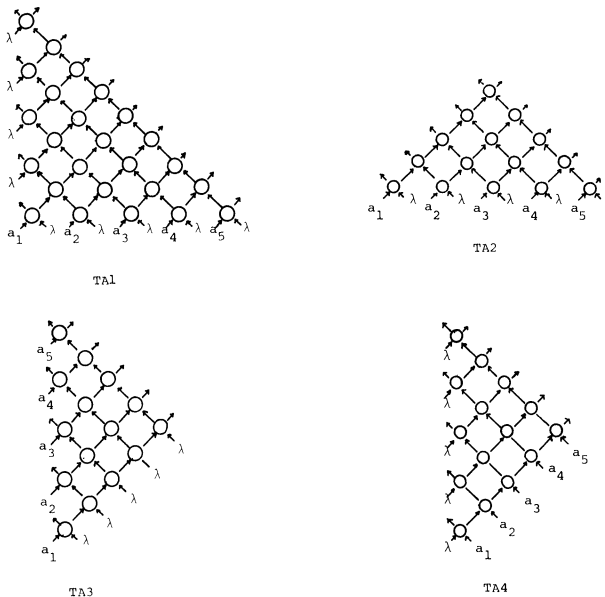


FIG. 30. Triangular systolic trellises.

if the topmost node generates an accepting symbol. Note that TA2 has time complexity n , and the others have time complexity $2n - 1$. The model TA2 is the triangular trellis studied in [6], [7], [13]. It can be shown (see [5], [8], [14]) that TA1, TA2 and TA3 are equivalent (in recognition power) to a real-time CA [1], [8], [10], [15], [22], [23], a real-time one-way CA (OCA) [1], [8], [9], [10], [23] and a real-time iterative automaton (IA) [2], respectively. See Fig. 31, column 2. TA4 is equivalent to the one-way iterative automaton (OIA) operating in time $2n - 1$. In the figure, the nodes represent the same (Mealy type) sequential machine. In CA and OCA, the input symbols a_1, a_2, \dots, a_n are fed in parallel at time 0. In IA and OIA, the input symbols a_1, \dots, a_n are fed serially at time $0, 1, \dots, n - 1$. The string $w = a_1 a_2 \dots a_n$ is accepted (in the cases of CA, OCA and IA) if the leftmost node enters an accepting state at time n . In the case of OIA, the leftmost node must enter an accepting state at time $2n - 1$.

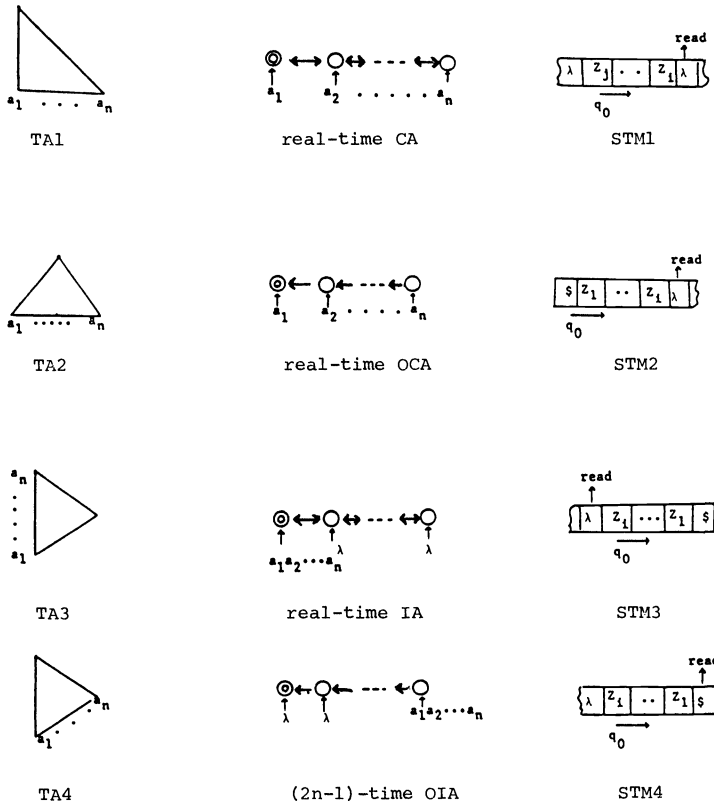


FIG. 31. Characterizations of triangular trellises.

We illustrate that TA4 and an OIA are equivalent with a simple example. Let M be an OIA with input $a_1 a_2 a_3 a_4$ as shown in Fig. 32(a). By expanding M in time and space, we get the trellis form shown in Fig. 32(b). Figure 32(c) shows the active area in Fig. 32(b) during the first $2n - 1$ steps. Finally, by folding the trellis in Fig. 32(c) along the dotted line, we get the triangular trellis shown in Fig. 32(d). Conversely, given a TA4, it is easy to transform it back to an OIA.

Column 3 of Fig. 31 shows the STM variations which characterize the triangular trellises. In that figure all the STM variations operate in real-time, i.e., take $n + 1$ sweeps. STM1 is the real-time STM. For STM2, STM3 and STM4, initially the worktape

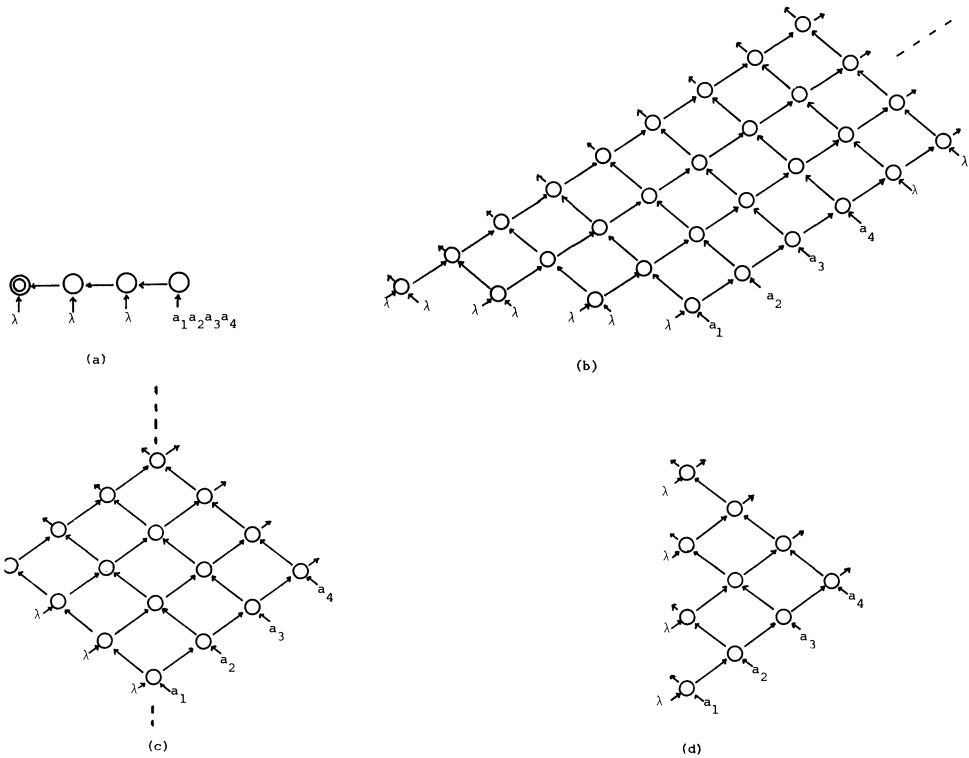


FIG. 32. Transformation of an OIA into a TA4.

contains blanks (λ 's) except for the symbol $\$$. The restrictions are the same as in an STM except the following:

- (1) Initial position of the RWH:
 - STM2: the blank cell to the right of $\$$.
 - STM3, STM4: the cell where $\$$ is written.
- (2) Condition for reading the input (see the arrow labeled "read" in Fig. 31):
 - STM2, STM3: when the RWH reads λ .
 - STM4: when the RWH reads $\$$.
- (3) Sweep range:
 - Between $\$$ and λ .
- (4) Accepting condition:
 - STM3, STM4: writes an accepting tape symbol at the end of the right-to-left sweep after reading the input end marker.
 - STM2: enters an accepting state at the end of the right-to-left sweep after reading the input end marker.

Figure 33 illustrates the operations of STM2, STM3 and STM4. Note that the sweep patterns for STM3 and STM4 are the same. The difference is the location of the RWH when the machines read the input.

The sequential machine characterizations can be used to show, rather easily, that certain languages are recognizable by the parallel devices. We give two examples below.

Example 4. $L = \{0^{2^n} \mid n \geq 1\}$ can be accepted by an STM3 which increments a binary counter (on the worktape to the left of $\$$) every time it makes a right-to-left sweep. The input is accepted if the counter has value $2^n + 1$ when the machine sees the input end marker $\$$. Hence, L can be accepted by a TA3 or a real-time IA. Direct construction of a real-time IA is not as simple (see, e.g. [8]).

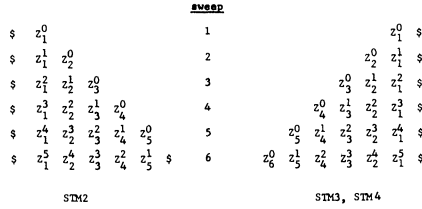


FIG. 33. Computation profiles of STM2, STM3 and STM4 on input $a_1a_2a_3a_4a_5\$$.

Example 5. Let Σ be an alphabet. For $i = 1, 2$, let $\Sigma_i = \{a_i \mid a \text{ in } \Sigma\}$ be a distinct alphabet. Let $\Delta = \Sigma_1 \cup \Sigma_2$. Define the homomorphisms $h_i: \Delta^* \rightarrow \Sigma^*$ by $h_i(a_j) = a$ if $i = j$ and ε otherwise. Let $\#$ be a new symbol. For $r \geq 0$, define the language $L_r = \{\#^m x \#^m \mid n > 0, x \text{ in } \Delta^+, |x| = 2n, \text{ and } h_1(x) = h_2(x)\}$. (Note that the strings in L_0 have no $\#$'s.) L_0 is called the twin shuffle over Σ [7]. It was conjectured in [7] that L_0 cannot be accepted by a TA2 (real-time OCA). However, using a nontrivial algorithm, L_2 was shown to be recognizable by a TA2 (real-time OCA) [7]. Here, we show by a simple construction that, in fact, L_r and the languages $L'_r = \{\#^m x \mid \text{same condition as in } L_r\}$ and $L''_r = \{x \#^m \mid \text{same condition as in } L_r\}$ (which have only one-sided paddings) can be accepted by STM2's for any $r \geq 1$. Thus, by the characterization in Fig. 31 these languages can be accepted by TA2's (real-time OCA's).

We describe an STM2 M' accepting L'_r . The STM2's for L_r and L''_r are constructed similarly. The worktape has two tracks. Let $y\$$ be an input to M' . The i th track in the padded segment (i.e., the $\#$'s) is used to construct the string $h_i(y)$ from the alphabet Σ_i with the symbols of $h_i(y)$ spaced r cells apart. Figure 34 shows an accepting profile for the case $r = 1$, $\Sigma = \{a, b, c\}$, $\Sigma_1 = \{a_1, b_1, c_1\}$, $\Sigma_2 = \{a_2, b_2, c_2\}$.

It does not seem that $L_0 = L'_0 = L''_0$ can be accepted by an STM2. However, it is obvious that an STM1 can accept L_0 . Hence, L_0 is recognizable by a TA1 (real-time CA).

Let $\mathcal{L}(\text{TA}i)$ denote the class of languages accepted by triangular trellises of type $\text{TA}i$. One can easily show that an STM1 can be simulated by an STM4, and conversely. Hence, $\mathcal{L}(\text{TA}1) = \mathcal{L}(\text{TA}4)$. Also, from [8], we know that $\mathcal{L}(\text{TA}2) \subset \mathcal{L}(\text{TA}1)$, $\mathcal{L}(\text{TA}3) \subset \mathcal{L}(\text{TA}1)$, $\mathcal{L}(\text{TA}2) - \mathcal{L}(\text{TA}3) \neq \emptyset$, and $\mathcal{L}(\text{TA}3) - \mathcal{L}(\text{TA}2) \neq \emptyset$.

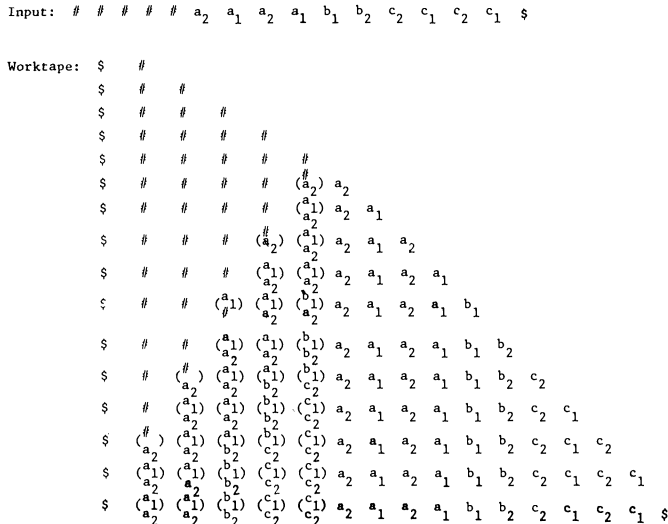


FIG. 34. The accepting profile of M' .

REFERENCES

- [1] W. BUCHER AND K. CULIK II, *On real time and linear time cellular automata*, RAIRO Inform. Théoret., to appear.
- [2] S. COLE, *Real-time computation by n-dimensional iterative arrays of finite-state machines*, IEEE Trans. Comput., C-18 (1969), pp. 349-365.
- [3] L. CONWAY AND C. MEAD, *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA, 1980.
- [4] S. COOK, *A hierarchy for nondeterministic time complexity*, JCSS, 7 (1973), pp. 343-353.
- [5] K. CULIK II AND J. PACHL, *Folding and unrolling systolic arrays*, ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Ottawa, August 1982.
- [6] K. CULIK II, J. GRUSKA AND A. SALOMAA, *Systolic trellis automata (for VLSI)*, Res. Rep. CS-81-34, Dept. Computer Science, Univ. Waterloo, Waterloo, Ontario, Canada, 1981.
- [7] ———, *Systolic trellis automata: stability, decidability and complexity*, Res. Rep. CS-82-04, Dept. Computer Science, Univ. Waterloo, Waterloo, Ontario, Canada, 1982.
- [8] C. CHOFFRUT AND K. CULIK II, *On real-time cellular automata and trellis automata*, Bericht F114, Institute für Informationsverarbeitung, Universität Graz, 1983.
- [9] C. DYER, *One-way bounded cellular automata*, Inform. Control, 44 (1980), pp. 261-281.
- [10] F. HENNIE, *Iterative Arrays of Logical Circuits*, MIT Press, Cambridge, MA, 1961.
- [11] O. IBARRA, *A note concerning nondeterministic tape complexities*, J. Assoc. Comput. Mach., 19 (1972), pp. 608-612.
- [12] ———, *On two-way multihead automata*, J. Comput. System Sci., 7 (1973), pp. 28-36.
- [13] O. IBARRA AND S. KIM, *Characterizations and computational complexity of systolic trellis automata*, Theor. Comput. Sci., 29 (1984), pp. 123-153.
- [14] O. IBARRA, S. KIM AND M. PALIS, *Some results concerning linear iterative arrays*, in preparation.
- [15] S. KOSARAJU, *On some open problems in the theory of cellular automata*, IEEE Trans. Computers, C-23 (1974), pp. 561-565.
- [16] H. KUNG, *Let's design algorithms for VLSI systems*, Proc. Caltech Conference on Very Large Scale Integration, Ch. L. Seitz, ed., Pasadena, CA, 1979, pp. 65-69.
- [17] ———, *The structure of parallel algorithms*, Res. Rep. Department of Computer Science, Carnegie-Mellon University, 1979.
- [18] H. KUNG AND C. LEISERSON, *Systolic arrays (for VLSI)*, Sparse Matrix Proc., I. S. Duff and G. W. Stewart, eds., Society for Industrial and Applied Mathematics, Philadelphia, 1979, pp. 256-282.
- [19] C. LEISERSON AND J. SAXE, *Optimizing synchronous systems*, Proc. 22nd Annual Symposium on Foundations of Computer Science, Nashville, Tennessee, 1981, pp. 23-26.
- [20] W. PAUL, *On time hierarchies*, Proc. Ninth Annual ACM Symposium on the Theory of Computing, 1977, pp. 218-222.
- [21] S. RUBY AND P. FISCHER, *Translational method and computational complexity*, Proc. Sixth Annual IEEE Symp. on Switching Circuit Theory and Logical Design, 1965, pp. 173-178.
- [22] A. SMITH III, *Real-time language recognition by one-dimensional cellular automata*, J. Comput. System Sci., 6 (1972), pp. 233-253.
- [23] H. UMEO, K. MORITA AND K. SUGATA, *Deterministic one-way simulation of two-way real-time cellular automata and its related problems*, Inform. Proc. Letters, 14 (1982), pp. 158-161.
- [24] A. WAKSMAN, *An optimal solution to the firing squad synchronization problem*, Informat. Control, 9 (1966), pp. 66-78.

INTERSECTION AND CLOSEST-PAIR PROBLEMS FOR A SET OF PLANAR DISCS*

MICHA SHARIR†

Abstract. Efficient algorithms for detecting intersections and computing closest neighbors in a set of circular discs, are presented and analyzed. They adapt known techniques for solving these problems for sets of points or of line segments. The main portion of the paper contains the construction of a generalized Voronoi diagram for a set of n (possibly intersecting) circular discs in time $O(n \log^2 n)$, and its applications.

Key words. generalized Voronoi diagrams, computational geometry, intersection detection, closest-pair and nearest neighbor problems, area calculation

1. Introduction. Let S be a set of n closed convex two-dimensional bodies of relatively simple structure (e.g. circular discs, straight segments, polygons of few sides, or expansions by some amount of such objects). In this paper we consider a variety of problems associated with such a set S . Typical such problems are:

I. Do any two objects in S intersect?

II. More generally, suppose that we assign a "color" to each object in S . Does there exist a pair of objects in S having different colors and intersecting each other?

III. If no two objects in S intersect, what is the smallest distance between any two objects in S (or, in the "colored" version, what is the smallest distance between two objects in S having different colors)? More generally, for each object B in S find the object in S (of different color) nearest to B .

IV. Preprocess S so that, given an arbitrary "query point" X , the object in S nearest to X can be found quickly.

Problems of this sort arise in robotics and are related to the problem of detecting and avoiding collisions between a moving subpart of a robot system and stationary objects, or between two or more moving subparts of such a system. In this note we will simplify the problem by assuming that each of the robot subparts and the stationary obstacles is either a closed convex object of a simple form, or else is covered by finitely many such objects. Note that in the "colored" setting some of the objects involved in our problem may be known a priori to intersect (e.g. objects covering two robot subparts hinged together, or two objects covering some robot subpart which overlap each other may always intersect). The "colored" version of our problems allows for this situation by looking only for intersection of subparts which would not intersect under normal conditions (e.g. subparts belonging to two distinct robot "arms", or a robot subpart and an obstacle, etc.).

Efficient solution of these problems in the three-dimensional case would facilitate construction of an "off-line" debugging system for robot control programs to check whether collision occurs along a planned path of motion, and would also make it more feasible to check in real time whether a moving subpart of the system is getting dangerously close to another (moving or stationary) object. This paper addresses the much simpler 2-dimensional case and uses generalized Voronoi diagrams for solving some of the problems just noted. In the 3-dimensional case, efficient algorithms for

* Received by the editors March 7, 1983, and in revised form February 27, 1984. This research has been supported in part by the Office of Naval Research under grant N00014-82-K-0381 and by a grant from the U.S.-Israeli Binational Science Foundation.

† School of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel and Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, New York 10012.

these problems would have to use other techniques since Voronoi diagrams in 3 dimensions are generally quadratic in the number of objects involved. A 3-dimensional version of the simplest problem mentioned above, namely that of detecting intersection in a set of n arbitrary spheres, has been recently solved in [HSS] by an $O(n \log^2 n)$ algorithm which uses sweeping methods.

This paper is organized as follows. In § 2 we describe a simple sweeping technique for detecting intersection in a set of n planar “monotone” objects. This technique has already been noted by various other researchers, but was not published. It is in fact a straightforward generalization of Shamos’ algorithm for detecting intersection of line segments. We include it here for the sake of completeness, to indicate that the simplest intersection detection problem mentioned above can be solved efficiently for rather general planar objects. In § 3 we define the generalized Voronoi diagram associated with a set of n arbitrary and possibly intersecting planar discs, and analyze some of its properties. Section 4 presents an efficient algorithm for the construction of this diagram. Section 5 contains some applications of the diagram for the problems noted above, and § 6 presents another unrelated application, namely that of efficient calculation of the area of a union of many discs.

The study of the two-dimensional case carried out in this paper may find applications in collision detection and avoidance for robot systems whose underlying motion is 2-dimensional, such as roving robots moving on a floor, etc. We hope that some of the ideas suggested here would also be useful in attacking the three-dimensional case.

2. Detecting intersection of monotone objects. In this section we consider the first problem posed above and show that it can be solved in time $O(n \log n)$ by a straightforward modification of an algorithm due to Shamos [Sh] which tests for intersection of straight line segments. As noted in the introduction, the material presented in this section has already been noted by several researchers. We assume that the bodies in S are monotone in the x -direction (i.e. the boundary of each object $B \in S$ consists of an upper portion and a lower portion and both portions extend monotonously from left to right). Thus for each such B there exist exactly two vertical lines tangent to it, and any vertical line between these lines intersects B in a closed segment. The structure of the objects in S is assumed to be simple enough so that each of the following operations takes constant time:

- (i) Check whether two specific bodies in S intersect each other.
- (ii) For each $B \in S$, find the smallest and largest abscissae of points in B .
- (iii) For each $B \in S$, find, for a given abscissa x , a point $(x, y) \in B$.

Let B_1, \dots, B_n be the objects in S . For each $j = 1, \dots, n$ let a_j (resp. b_j) be the smallest (resp. largest) abscissa of a point in B_j . For simplicity we assume that the $2n$ numbers $a_j, b_j, j = 1, \dots, n$, are all distinct (see Fig. 1).

Note that if we draw a vertical line $x = x_0$ it will cut (some of) the objects in S in straight segments which, if the objects do not intersect each other along that line, are disjoint from each other. Hence the objects in S which intersect the line $x = x_0$ can be linearly ordered in a list $L(x_0)$ in which an object B_i precedes another B_j if the segment cut off B_i by $x = x_0$ lies below the corresponding segment cut off B_j . Note that since the objects in S are monotone, the list $L(x_0)$ remains unchanged as x_0 increases until either the line $x = x_0$ meets a new object B_k (this will happen when $x_0 = a_k$), or when it stops making contact with some object B_k (just after $x_0 = b_k$), or when two of the objects in $L(x_0)$ intersect at $x = x_0$. Moreover, the leftmost intersection (if any exists) of any two objects $B, B' \in S$ will occur at some $x = x_0$ such that, for x slightly less than x_0 , the list $L(x)$ contains B and B' as adjacent elements. (Here we ignore

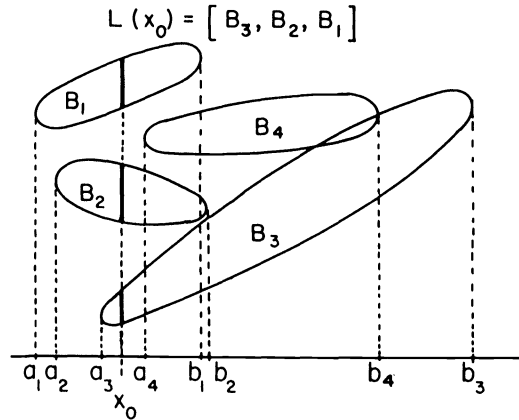


FIG. 1. An instance of the intersection problem.

the special case in which B and B' intersect at a point which is leftmost in one of these objects, which requires a slightly different argument.)

In view of these observations, to detect the presence or absence of an intersection one simply has to check repeatedly whether any two adjacent elements in the list $L(x_0)$ intersect each other. Plainly, these checks have to be performed only at points where $L(x_0)$ changes (i.e. at the points $x_0 = a_j, b_j, j = 1, \dots, n$), and one only needs to test newly adjacent pairs in $L(x_0)$. To facilitate execution of these operations, the list L can be maintained as a 2-3 tree, allowing all the required list-maintenance operations to be performed in time $O(\log n)$.

Details are as follows. The algorithm begins by sorting the $2n$ numbers $a_j, b_j, j = 1, \dots, n$, in increasing order, and then processes them from left to right. Initially, the list L is empty. Suppose that the abscissa currently being processed is one of the a_j . Then L is updated by inserting the object B_j into L in its proper place, using a standard 2-3 tree search during which comparison of two objects B, B' is accomplished by comparing two representative points in the intersection of $x = a_j$ with B, B' respectively (we have assumed that this can be done in constant time). After insertion, the algorithm finds the two objects B, B' immediately preceding and succeeding B_j in L , and checks whether either B or B' intersects B_j .

Similarly, if the abscissa currently being processed is b_j , then the object B_j is deleted from L , using essentially the technique just outlined. After deletion, the algorithm finds the two objects in L which immediately preceded and followed B_j prior to its deletion (these will have become newly adjacent in L after deletion of B_j), and determines whether they intersect each other.

The algorithm halts whenever an intersection is detected, or, if no intersection has been detected, when all the abscissae a_j and b_j have been processed, in which case the algorithm reports that there is no intersection between the objects in S .

The correctness of the algorithm follows from the preceding observations. The time complexity of the algorithm is $O(n \log n)$ since processing of each of the $2n$ abscissae a_j, b_j can be accomplished in $O(\log n)$ time, using a 2-3 tree representation for the list L .

3. Voronoi diagrams for circular objects. The algorithm presented in the preceding section does not solve the more complicated "colored object" intersection problem posed in the introduction. Indeed, the argument justifying the correctness of the algorithm breaks down as soon as an intersection is detected, so that if the first

intersection detected is between two objects having the same color we can no longer use the procedure described to find additional intersection points. To handle the colored intersections problem, we therefore propose a different approach based on generalized Voronoi diagrams. We will show such an approach which can be used to handle the special case where all the objects in the set S are circular discs, not necessarily of equal radii.

Let each of the objects $B_j \in S$ be a disc of radius r_j about the center x_j , for $j = 1, \dots, n$. These circular discs need not be disjoint from each other, and may intersect, or even contain, one another. We define a generalized Voronoi diagram $\text{Vor}_0(S)$ associated with the set S as follows. For each $i \neq j$ define

$$H(i, j) = \{y \in E^2: d(x_i, y) - r_i \leq d(x_j, y) - r_j\},$$

i.e. the set of all points whose distance from B_i is no greater than their distance from B_j (note that the distance of y from B_i is taken as the distance of y from the boundary of B_i with a positive sign if y lies outside B_i and with a negative sign if y lies inside B_i). Then define the (closed) *Voronoi cell* $V(i)$ associated with B_i to be

$$V(i) = \bigcap_{j \neq i} H(i, j),$$

i.e. the set of all points y whose distance from B_i is no greater than y 's distance from any other element of S . (We will sometimes refer to the point x_i as the *center* of this Voronoi cell; hence the center of $V(i)$ is the same as the center of the disc B_i .) Finally, the *Voronoi diagram* $\text{Vor}_0(S)$ is defined to be the set of points which belong to more than one Voronoi cell. For simplicity we assume that no point in $\text{Vor}_0(S)$ lies in more than three Voronoi cells. This assumption generalizes the familiar assumption concerning Voronoi diagrams associated with a set of points, in which one requires that no more than three of these points be cocircular. As in the case of points, this assumption is not essential, and is made just to simplify the description of the algorithm. Fig. 2. shows an example of such a Voronoi diagram.

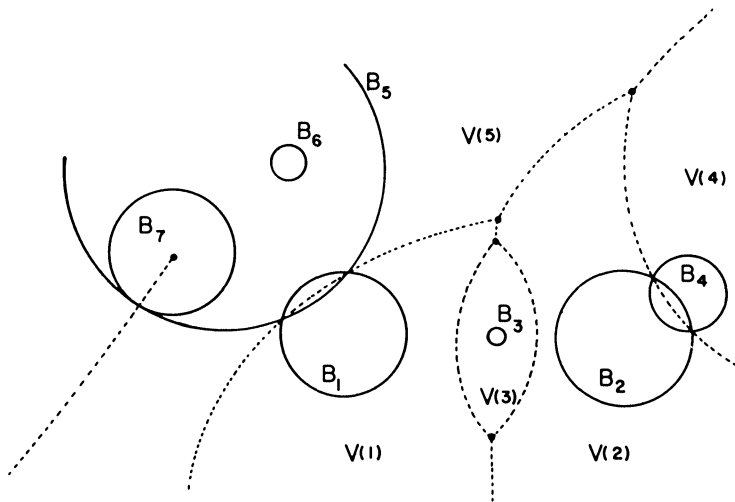


FIG. 2. The Voronoi diagram of a set of circular discs.

Generalized Voronoi diagrams have been previously introduced and analyzed by various authors; see e.g. Kirkpatrick [Ki], who considers Voronoi diagrams for a set of line segments and points, and Lee and Drysdale [LD], who consider Voronoi

diagrams for sets of line segments and for sets of circular objects. Our setup is somewhat different from that of [LD], in that we allow the objects in S to intersect each other, or even to contain one another, whereas in [LD] the circles are required to be disjoint. As we shall see below, there are some difficulties in adapting the technique of [LD] to the case of intersecting circles. The algorithm of Lee and Drysdale for constructing the Voronoi diagram of n disjoint circles runs in time $O(n \log^2 n)$, but is given in a very sketchy form without any analysis of the structure of the diagram, and without any proof of its correctness. Another algorithm for constructing generalized Voronoi diagrams for circles, which runs in time $O(nc^{(\log n)^{1/2}})$, has been presented by Drysdale and Lee [D], [DL].

The generalized Voronoi diagram just defined has the following properties.

(1) *The collection of Voronoi cells covers the whole plane.*

Proof. Immediate. Indeed, given any $y \in E^2$, it will belong to the cell $V(i)$ for which

$$d(x_i, y) - r_i = \min \{d(x_j, y) - r_j : j = 1, \dots, n\}.$$

(2) *$V(i)$ is empty iff B_i is wholly contained in the interior of another disc B_j ; $V(i)$ has an empty interior iff B_i is wholly contained in another disc B_j .*

Proof. For the first assertion, note that if $V(i)$ is not empty it contains a point x such that

$$d(x, x_i) - r_i \leq \min \{d(x, x_j) - r_j : j = 1, \dots, n\}.$$

The triangle inequality then shows that this same inequality holds for $x = x_i$, i.e.

$$-r_i = \min \{d(x_i, x_j) - r_j : j = 1, \dots, n\},$$

which implies that, for any of the other circles B_j of S , the point of B_i at maximum distance from x_j is not interior to B_j . Thus there is no B_j whose interior contains B_i . Conversely, if $V(i)$ is empty, then $x_i \notin V(i)$, so that there exists $j \neq i$ such that

$$d(x_i, x_j) - r_j < -r_i$$

i.e.

$$d(x_i, x_j) + r_i < r_j,$$

which is to say that B_j contains B_i in its interior. Similarly, if the interior of $V(i)$ is not empty, it contains a point x such that

$$d(x, x_i) - r_i < \min \{d(x, x_j) - r_j : j = 1, \dots, n\},$$

and again this inequality must hold for $x = x_i$. Thus

$$-r_i < d(x_i, x_j) - r_j \quad \text{for each } j \neq i.$$

Hence $r_j < d(x_i, x_j) + r_i$ for each j , i.e. the point of B_i at maximum distance from x_j does not belong to B_j . Thus there is no B_j which contains B_i .

Conversely, if the interior of $V(i)$ is empty, x_i does not belong to this interior, so that there exists $j \neq i$ such that

$$d(x_i, x_j) - r_j \leq -r_i$$

or

$$d(x_i, x_j) \leq r_j - r_i,$$

which is to say, B_i is wholly contained in B_j .

(3) $\text{Vor}_0(S)$ consists of straight or hyperbolic arcs.

Proof. Let $y \in \text{Vor}_0(S)$ be such that y lies in both Voronoi cells $V(i)$ and $V(j)$. Then we have

$$d(x_i, y) - d(x_j, y) = r_i - r_j.$$

The locus of points satisfying this condition is a hyperbolic arc having x_i and x_j as foci, or, if $r_i = r_j$, the perpendicular bisector to the segment $[x_i, x_j]$. (Note also that this (generally hyperbolic) locus degenerates into a half-line if $d(x_i, x_j) = \pm(r_i - r_j)$, i.e. if the discs B_i, B_j are tangent to one another with one of them wholly containing the other.)

(4) *Each nonempty Voronoi cell $V(i)$ is star-shaped with respect to the point x_i . Moreover, if $V(i)$ has nonempty interior then the interior of a segment connecting x_i to a point on the boundary of $V(i)$ does not intersect the interior of any other Voronoi cell, and such a segment can intersect another Voronoi cell $V(j)$ only if the corresponding disc B_j is wholly contained in B_i (so that, by remark (2), $V(j)$ has empty interior).*

Proof. Let $y \in V(i)$, and let I be the segment connecting x_i to y . We first claim that each point z in the interior of I is contained in $V(i)$. Indeed, if this were false then there would exist a point z in the interior of I which does not belong to $V(i)$. By (1) there would exist $j \neq i$ such that z belongs to $V(j)$. Hence

$$d(x_j, z) - r_j < d(x_i, z) - r_i.$$

By the triangle inequality (and since z lies between x_i and y on a straight line) we have

$$d(x_j, y) - r_j \leq d(x_j, z) + d(z, y) - r_j < d(x_i, z) + d(z, y) - r_i = d(x_i, y) - r_i.$$

Thus y cannot belong to $V(i)$, a contradiction which proves that $V(i)$ is star-shaped.

Next suppose that $V(i)$ has nonempty interior. Let $y \in V(i)$ and I be as above, and suppose that I contains an interior point z which also belongs to some other Voronoi cell $V(j)$. Then we have

$$d(x_j, z) - r_j = d(x_i, z) - r_i.$$

Using the triangle inequality as before, we obtain

$$d(x_j, y) - r_j \leq d(x_j, z) + d(z, y) - r_j = d(x_i, z) + d(z, y) - r_i = d(x_i, y) - r_i \leq d(x_j, y) - r_j,$$

since $y \in V(i)$. Hence x_j lies on the line containing I , and z lies between x_j and y on this line. However, this implies that

$$d(x_i, x_j) = \pm(r_i - r_j),$$

i.e. that one of the discs B_i, B_j contains wholly the other. But by (2) B_i is not wholly contained in any other disc, so that B_j is wholly contained in B_i , and consequently $V(j)$ has empty interior. This establishes our two final assertions.

DEFINITION. The modified Voronoi diagram $\text{Vor}(S)$ is defined to consist of the boundaries of all cells in $\text{Vor}_0(S)$ having nonempty interior. All other cells are discarded from the modified diagram.

(5) *The intersection I of three Voronoi cells $V(i), V(j), V(k)$ in $\text{Vor}(S)$ consists of at most two points.*

Proof. Let y be a point in I . By (4) the interior of the segment L_i (resp. L_j, L_k) connecting y with x_i (resp. x_j, x_k) is contained in $V(i)$ (resp. $V(j), V(k)$) and in no other cell. Let y' be another point in I , and let L'_i, L'_j, L'_k be the segments connecting y' with x_i, x_j, x_k respectively. It is clear that no two of these six segments can intersect one another (except at an endpoint). It follows that I cannot contain a third point z , because at least one of the segments connecting z to the three centers x_i, x_j, x_k would have to intersect one of the preceding six segments, which is impossible.

(6) Let D^* denote the dual of $\text{Vor}(S)$, defined to be a graph whose vertices are the points x_i , $i = 1, \dots, n$ for which $V(i)$ has nonempty interior, and which contains an edge $[x_i, x_j]$ if $V(i)$ and $V(j)$ have a nonempty intersection including at least one open arc. If $V(i)$ and $V(j)$ intersect in more than one arc, define the graph D^* to contain multiple edges connecting x_i to x_j , one for each such arc. Then D^* is a planar graph, and its natural planar embedding (described below) has all of its faces (except for the outer one) containing at least three edges.

Proof. We define an embedding of D^* in E^2 as follows. Each vertex x_i is mapped to itself (as a point in E^2). Let $e = [x_i, x_j] \in D^*$ be an edge corresponding to an open arc α in the intersection of $V(i)$ and $V(j)$. It follows from (5) that there must exist at least one point $y \in \alpha$ which does not belong to any other Voronoi cell. We then map the edge e to the path consisting of the two segments $[x_i, y]$ and $[y, x_j]$. To see that the resulting graph G is indeed a planar embedding of D^* , suppose to the contrary that two distinct edges $e_1 = [x_i, x_j]$, $e_2 = [x_k, x_l]$ of D^* map to paths which intersect each other at a point z which is not a vertex of G . Let z_1 (resp. z_2) be a point on the common boundary of $V(i)$ and $V(j)$ (resp. $V(k)$ and $V(l)$) such that e_1 (resp. e_2) appears in G as the union of the segments $[x_i, z_1]$ and $[z_1, x_j]$ (resp. $[x_k, z_2]$ and $[z_2, x_l]$). Suppose without loss of generality that the segments $s = [x_i, z_1]$ and $s' = [x_k, z_2]$ intersect at a point other than a common endpoint $x_i = x_k$. Since by (4) the interior of the segment s (resp. s') is contained in $V(i)$ (resp. $V(k)$) and in no other Voronoi cell having nonempty interior, we must have either $i = k$ or $z_1 = z_2$. In the first case s and s' meet at x_i , and so if they meet at another point they must overlap each other, which is possible only if either $z_1 = z_2$ or if one of these points (say z_1) lies in the interior of the other segments s' . The latter assumption would contradict the fact that the interior of s' is wholly contained in $V(i)$ and in no other cell having nonempty interior. Thus we must have $z_1 = z_2$, and then by the choice of these points it follows that $j = l$ too. Then it is plain that the two arcs of the common boundary of $V(i)$ and $V(j)$ which define our two paths are identical. Hence the two edges e_1 and e_2 are not distinct, contrary to assumption. All this shows that G is a planar embedding of D^* .

Note that the inner faces of the embedding G of D^* stand in 1-1 correspondence to the Voronoi vertices in $\text{Vor}(S)$. Hence the second part of our assertion will follow from the fact that each Voronoi vertex must be incident to three Voronoi edges. This property of Voronoi vertices will be established later in this section (see the Corollary to property (11) below), and will thus imply the property stated above.

(7) Since D^* contains $O(n)$ vertices, and since each of its faces contains at least three edges, it follows by Euler's formula that it has at most $O(n)$ edges, that is, $\text{Vor}(S)$ consists of at most $O(n)$ connected straight or hyperbolic arcs.

(8) Let C be the convex hull of all the discs $B_i \in S$. Note that the boundary of C consists of an alternating sequence of straight segments and circular arcs, the circular arcs being boundary portions of some of the discs, whereas the straight segments are tangents to a pair of discs in S . We will say that B_i and B_j are adjacent along the boundary of C , if this boundary contains a straight segment tangent to both B_i and B_j . Then the unbounded edges of $\text{Vor}(S)$ are those edges that are common to two cells $V(i)$, $V(j)$ for which B_i and B_j are adjacent along the boundary of C . (This property will not be used in the sequel, but is noted as a generalization of similar properties for other types of Voronoi diagrams [Sh].)

Proof. Let e be an unbounded edge of $\text{Vor}(S)$, common to two Voronoi cells $V(i)$ and $V(j)$. Since e is either a straight or hyperbolic arc, it tends asymptotically to some half-line l . Suppose, without loss of generality, that l is the positive y -axis, and let $a = [0, t]$ be a point on l . For sufficiently large t and for any $k = 1, \dots, n$, $d(a, B_k)$

behaves asymptotically as $t - \eta_k - r_k$, where η_k is the y -coordinate of x_k , i.e., as $d(a, B_{k'})$ where $B_{k'}$ is the image of B_k translated parallel to the x -axis until its center lies on the y -axis. It follows that

$$\eta_i + r_i = \eta_j + r_j \geq \eta_k + r_k$$

for every $k = 1, \dots, n$. This however is easily seen to imply that B_i and B_j are adjacent to each other along C , since the line $y = \eta_i + r_i$ is tangent to both discs B_i, B_j and all the discs in S lie in the lower half-plane which this line bounds, so that the portion of this line between its points of tangency with B_i and B_j belongs to the boundary of C . (An extreme case that we need consider is that in which the line $y = \eta_i + r_i$ is also tangent to a third disc B_k , at a point lying between its points of contact with B_i and B_j . However, if such a situation arises, it is easy to check that all the points on e sufficiently far away are nearer to B_k than to one of B_i, B_j , contradicting the definition of e . The converse statement, namely that any pair of adjacent discs along C induces an unbounded Voronoi edge, can be proved using the above argument in reverse.

(9) *Vor(S) need not be connected. In fact, it can have up to $O(n)$ connected components. However, every connected component of $Vor(S)$ is unbounded.*

Proof. Consider the following set S of discs, which consists of the unit disc B_1 , and of k additional small discs B_2, \dots, B_{k+1} all of radius ρ , such that the centers of these discs are placed on the boundary of B_1 at equally spaced positions. If ρ is chosen to be sufficiently small (e.g. of the order $O(1/k^2)$) then it is easily checked that for each $j = 2, \dots, k+1$ the discs B_1 and B_j are adjacent to each other along the boundary of the convex hull of the B_i 's. Moreover, it can also be shown that for ρ sufficiently small each unbounded edge common to $V(1)$ and some other $V(j)$ is a full branch of the corresponding hyperbola, and that no two such edges intersect each other. This shows that $Vor(S)$ can have as many as $O(n)$ connected components.

Remark. The example just given also shows that the boundary of a single Voronoi cell ($V(1)$ in the example) can have up to $O(n)$ disjoint connected components.

Suppose next that for some set S of discs $Vor(S)$ contains a bounded component K . Then the portion E of a sufficiently small neighborhood of K which lies exterior to K must be contained in a single Voronoi cell $V(i)$, since otherwise some arc of $Vor(S)$ would have to enter any such exterior neighborhood of K , contradicting the assumption that K is a connected component of $Vor(S)$. But if a whole neighborhood of K in the exterior of K lies in $V(i)$, there must exist a point $y \in \text{int}(V(i))$ such that the line connecting y to x_i intersects K , contradicting (4). Thus $Vor(S)$ cannot have any bounded connected component.

COROLLARY. *Vor(S) does not contain any isolated point. Moreover, by modifying the argument given above one can also show that each Voronoi vertex must belong to at least two edges. (Assertion (11) below will strengthen this claim, by showing that each such vertex must belong to three distinct Voronoi edges.)*

(10) *No two edges of $Vor(S)$ can be tangent to each other. Also, for each point z on a Voronoi edge e separating two cells $V(i)$ and $V(j)$, the segment connecting z to x_i (or to x_j) is not tangent to e .*

Proof. Since Voronoi edges are either straight segments of hyperbolic arcs it follows that if two Voronoi edges are tangent to each other then any Voronoi edges lying between them at their point of tangency must be tangent to both of them. Hence if there exist any two tangent Voronoi edges, then there exist two such edges which belong to the boundary of the same Voronoi cell. Assume this to be possible, and let $V(i)$ be a Voronoi cell whose boundary contains two edges e, e' which also belong to $V(j), V(k)$ respectively, and which are tangent to each other at some point y . It is

plainly impossible for both e and e' to be straight arcs; hence at least one must be part of a hyperbola. Let l be the line which is tangent at y to both hyperbolas (or to the hyperbola and straight line) containing e , e' respectively. As is well-known, a line tangent to a hyperbola at a point y bisects the angle $f_1 y f_2$, where f_1 and f_2 are the two foci of the hyperbola; furthermore, this result also holds trivially in case the hyperbola degenerates into the perpendicular bisector of the segment connecting the two points f_1, f_2 . Thus, in any case l bisects the angle between the two segments connecting y to x_i and x_j (resp. to x_i and x_k). It follows that these two angles must be equal, and consequently the three points y, x_j, x_k are collinear, with x_j and x_k lying on the same side of y . Suppose for definiteness that x_j lies between y and x_k . By definition of e and e' we then have

$$d(x_i, y) - d(x_j, y) = r_i - r_j,$$

$$d(x_i, y) - d(x_k, y) = r_i - r_k$$

or

$$d(x_k, y) - d(x_j, y) = d(x_j, x_k) = r_k - r_j,$$

which is to say, B_j is wholly contained in B_k , so that by definition $\text{Vor}(S)$ does not contain any edge bounding $V(j)$. This contradiction establishes our assertion.

The second assertion follows from the fact that no tangent to a hyperbola can pass through any of its foci. (The only exception is when the hyperbolic arc containing e degenerates into a half-line, but then either B_i or B_j must be wholly contained in some other disc, so that by convention e does not appear in our Voronoi diagram.)

(11) *Let e and e' be two adjacent edges along the boundary of a Voronoi cell $V(i)$. Then the interior angle in $V(i)$ between e and e' is less than 180 degrees. (Stated otherwise, in traversing the boundary of $V(i)$ with $V(i)$ to the right, we make a right turn as we pass from one of the edges e, e' to the other.)*

Proof. We have already shown that e and e' cannot be tangent to each other. Let y be their point of intersection, and let l be the line containing x_i and y . The two edges e and e' cannot both lie on the same side of l in the vicinity of y , because then the segment connecting x_i to a point on one of them sufficiently near y would have to intersect the other edge, contradicting (4). Suppose then that e lies on the left side of l (oriented from x_i to y), and that e' lies on the right side of l . Extend e along the hyperbolic branch containing it into the right side of l (note that a hyperbola is never tangent to the segment connecting a point lying on it to one of its foci), and denote the extended portion of e by e'' . Then e' must lie between x_i and e'' , for otherwise the segment connecting x_i to any point on e' lying sufficiently near to y will intersect e'' at a point z , and plainly no point $z \in e''$ can belong to the interior of $V(i)$ (because z is equidistant from B_i and from some other disc in S), again contradicting (4). It therefore follows that the interior angle between e and e' is less than 180 degrees, as asserted.

COROLLARY. *This argument shows that each Voronoi vertex must be incident to three distinct Voronoi edges, for if it belonged to just two edges, at least one of the angles between these edges would be interior to some Voronoi cell, and would be greater than 180 degrees, contrary to what we have just shown.*

4. Efficient construction of generalized Voronoi diagrams. Next we present an algorithm which adapts the divide-and-conquer methods used by Shamos [Sh] and by Kirkpatrick [Ki] to construct the modified Voronoi diagram of a set of points, and which computes the Voronoi diagram of a set S of n circular bodies in time $O(n \log^2 n)$. The algorithm produces a list of all Voronoi cells having nonempty interior, and for

each such cell constructs a circular list containing the edges on its boundary, arranged in clockwise order (for unbounded cells this list will also include “virtual edges” at infinity connecting pairs of unbounded edges e, e' such that the intersection of e' with an arbitrarily large circle lies immediately clockwise to the intersection of this circle with e). Finally, the algorithm produces a table in which each edge points to the two cells containing it.

For simplicity, we will assume in what follows that no two circles in S have distinct leftmost points lying on the same vertical line. If S does not have this property, then we can apply an infinitesimal rotation that will make the abscissae of the leftmost points of all circles in S distinct from each other. Cf. also [SS] where a similar technique based on infinitesimal perturbations is used to resolve degenerate configurations arising in other geometric problems. The algorithm begins by dividing S into two subsets R and L of equal size, such that the leftmost point w_i of each $B_i \in L$ lies to the left of the leftmost point w_j of every disc $B_j \in R$. This partitioning of S can easily be done by finding the median of these leftmost points in time $O(n)$. Note that it has the property that no disc $B_i \in L$ is wholly contained in another disc $B_j \in R$, although a reverse containment is possible.

Assume that the Voronoi diagrams $\text{Vor}(R)$ and $\text{Vor}(L)$ have been computed recursively. The main step of the algorithm is to merge these two diagrams into a single diagram $\text{Vor}(S)$. For this, one must compute the set C of points y which are simultaneously nearest to a disc $B_i \in L$ and to a disc $B_j \in R$. Following Kirkpatrick [Ki] we call C the *contour* separating R and L . We will see that, once C has been computed, $\text{Vor}(S)$ can be obtained by taking the union of $\text{Vor}(R)$, $\text{Vor}(L)$, and C , and then by discarding (portions of) edges belonging to one of the partial diagrams $\text{Vor}(R)$, $\text{Vor}(L)$, whose points have become nearer to some object belonging to the other set (these portions will be delimited by the intersections of these edges with C). Note that during this merging step some Voronoi cells $V(i)$ with $B_i \in R$ may be wholly deleted from $\text{Vor}(S)$ if B_i happens to lie wholly inside some disc $B_j \in L$.

Since C will be a curve consisting of straight and hyperbolic arcs, the complement of C will consist of finitely many open connected planar regions. Each such region M is either a union of cells $V(i)$ (in the final diagram $\text{Vor}(S)$) with $B_i \in L$ (in which case we will call M an L -region), or a union of cells $V(i)$ with $B_i \in R$ (in which case M will be called an R -region).

By assertion (9) of § 3 and the example provided there, the contour C may consist of several disjoint connected components, no matter how S is partitioned into R and L (note that in the point-based case [Sh] appropriate partitioning of S will produce a contour with a single component). Moreover, it is also possible for a connected component of C to be bounded, as shown by the example appearing in Fig. 3.

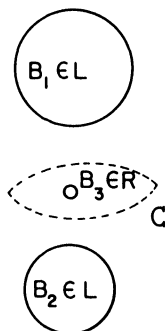


FIG. 3. A bounded connected component of C .

The following lemma restricts this complexity, and begins to develop some of the facts that will be needed to show that tracing all the components of C need not be expensive.

LEMMA 4.1. (a) C consists of a disjoint union of simple topologically closed curves without endpoints (i.e. each such curve is either closed or stretches to infinity in both directions).

(b) Let $B_i \in L$ and let u be the horizontal half-line whose rightmost endpoint is x_i . Then u does not intersect C (and consequently lies wholly inside an L -region).

(c) There exists precisely one L -region; all other components of the complement of C are R -regions.

(d) Each R -region has a connected boundary, consisting of a single (bounded or unbounded) component of C .

Proof. (a) It follows from its definition that C is a union of Voronoi edges of $\text{Vor}(S)$. It therefore suffices to show that for each Voronoi vertex ν of $\text{Vor}(S)$ lying on C , there are exactly two Voronoi edges emerging from ν which belong to C . Since we have ruled out degenerate configurations, we can assume that ν belongs to exactly three Voronoi cells $V(i)$, $V(j)$ and $V(k)$. Moreover, since $\nu \in C$, one of the discs B_i, B_j, B_k must belong to L , and another of these discs must belong to R . Assume first that $B_i, B_k \in L$ and that $B_j \in R$. Then, in the neighborhood of ν , the contour C consists of the two edges separating $V(j)$ from $V(i)$ and $V(k)$ respectively. Much the same argument applies if $B_i, B_k \in R$ and $B_j \in L$. This proves (a).

(b) Suppose the contrary, and let z be a contour point lying on u . Let $w \in u$ be a point at distance r_i from x_i (that is, w is the leftmost point of B_i). It follows that there exist discs $B_j \in R$ (which is not wholly contained in any other disc), and $B_k \in L$ such that

$$d(z, B_j) = d(z, B_k) \leq d(z, B_i) = d(z, w).$$

But then some point on B_j (and in particular its leftmost point) must lie to the left of w , or coincide with w . However, both these possibilities contradict the way in which L and R have been defined, a contradiction which proves (b).

(c) It suffices to show that the centers x_i, x_j of each pair B_i, B_j of discs in L are connected to each other via a path which does not intersect the contour C . Let u_i (resp. u_j) be the horizontal half-line whose rightmost endpoint is x_i (resp. x_j), and let $w_i \in u_i$ (resp. $w_j \in u_j$) be the leftmost point of B_i (resp. B_j). For each t , let y_i (resp. y_j) be a point on u_i (resp. u_j) whose abscissa is t . We claim that if t is negative and has a large enough absolute value, the segment $e = y_i y_j$ does not intersect the contour, which, together with (b), implies that x_i and x_j are connected to each other via the polygonal path $x_i y_i y_j x_j$ which is wholly contained in a single L -region. To see this, suppose to the contrary that there exists a contour point $z \in e$, nearest to some $B_k \in R$ and to some $B_m \in L$. Then

$$d(z, B_k) = d(z, B_m) \leq d(z, B_i) = (d^2(y_i, w_i) + d^2(z, y_i))^{1/2} \approx d(y_i, w_i),$$

and similarly

$$d(z, B_k) \leq d(z, B_j) \approx d(y_j, w_j)$$

as $t \rightarrow \infty$. It follows that the leftmost point w_k of B_k does not lie to the right of w_i or of w_j . Since, by definition of L, R , w_k cannot lie to the left of w_i, w_j , it follows that the three points w_i, w_j, w_k all have the same abscissa, again contradicting the way in which L and R are defined.

(d) Suppose that M is an R -region whose boundary consists of at least two disjoint connected components K_1, K_2 . By part (a) of the present lemma K_1 (resp. K_2) partitions the plane into two disjoint components, both bounded by K_1 (resp. K_2). Since K_1 and K_2 are disjoint, it follows that they both collectively partition the plane into three components, one of which contains M , whereas the other two contain L -regions. This, however, contradicts (c), thus proving our assertion. Q.E.D.

In general outline, the remainder of our argument is as follows. We first show that, given points z_K on each of the components K of C , the whole of C can be traced in a number of steps bounded by $O(n)$. Next we show how to find such a set of points z_K . This is done by noting that (by definition) every R -region must contain at least one center x_i of a disc $B_i \in R$ such that B_i is not contained in any other disc of R . Hence if we iterate over all such points x_i , and connect each one of them by a straight arc e to a point of an L -region; these e must together intersect all the components of C . We will show that in total time $O(n \log n)$ we can find such arcs e , each intersecting C in just one point z , and in this way can find a z on each component K of C . This leads to an $O(n \log n)$ merging step, and hence to an $O(n \log^2 n)$ overall algorithm. Note that in this generality our algorithm is very similar to that of [LD]; they differ in the actual implementation of the various steps just outlined.

The tracing of C during the merge phase of our algorithm is done in a manner quite similar to that described in [Ki]. To facilitate this tracing, we will find it convenient (following [Ki]) to partition each Voronoi cell $V(i)$ (of either $\text{Vor}(L)$ or $\text{Vor}(R)$) into *subcells* by connecting x_i to each vertex v of $V(i)$ by a straight segment (called a *spoke*, as in [Ki]). Clearly each subcell is an angular sector bounded by two spokes and one Voronoi edge. Note that, given a directed straight line or hyperbolic arc e , its intersection points with the boundary of any Voronoi subcell can be found in constant time, assuming that we use an appropriate representation of the corresponding diagram $\text{Vor}(L)$ or $\text{Vor}(R)$.

Suppose that we have somehow found a point $z \in C$ (but such that z is not in either $\text{Vor}(L)$ or $\text{Vor}(R)$), for which the two discs $B_i \in L, B_j \in R$ nearest to z are known, and suppose further that the two subcells of $V(i)$ in $\text{Vor}(L)$ and of $V(j)$ in $\text{Vor}(R)$ to which z belongs are also known. Then we can trace the component K of C containing z as follows. We first find the Voronoi edge e (in $\text{Vor}(S)$) containing z . Note that e is part of the straight line or hyperbolic arc equidistant from B_i and B_j , and is an edge lying on K . We begin tracing K by following e from z in some direction, and by computing its intersection points with the boundaries of the two subcells $U(i), U(j)$ of $V(i), V(j)$ respectively, containing z . Suppose for specificity that the nearest of these points along e is the point z' at which e intersects the boundary of $U(i)$. If z' lies on a Voronoi edge, then the contour K crosses this edge to another Voronoi cell $V(k)$ of $\text{Vor}(L)$ after z' (by assertion (10) of the previous section, two Voronoi edges are never tangent to one another). In this case, K continues after z' along the Voronoi edge e' containing points equidistant from B_k and B_j . On the other hand, if z' lies on a spoke, K will continue after z' along the edge e , but will cross into another subcell of $V(i)$. (Note that the contour can never be tangent to a Voronoi spoke, by the second part of assertion (10).)

Tracing the contour in this way, we either come back to z , in which case the component K is a bounded component of the contour, or else we reach an unbounded edge of the contour, in which case we have to repeat the tracing procedure just outlined by starting from z in the other direction of the edge e in order to obtain the entire component K .

Let M be the R -region bounded by K . Each cell $V(i)$ through which K passes

is cut by K into several portions, one of which belongs to an R -region (and contains x_i) while the others belong to the L -region. Moreover, all the cell portions belonging to R -regions belong to the same R -region M . Thus, as K is being traced, we can also note that all these cell portions cut by K belong to M . Observe that M may also contain additional *internal* cells of $\text{Vor}(R)$ which have not yet been encountered. These will be dealt with during later steps of the algorithm.

Next we show that if a point z_K is available on each of the components K of C , the total cost of constructing C is $O(n)$. The complexity of the tracing procedure just described is plainly $O(n_1^{(K)} + n_2^{(K)})$, where $n_1^{(K)}$ is the number of Voronoi edges in K , and where $n_2^{(K)}$ is the number of intersections of K with Voronoi spokes (in either $\text{Vor}(L)$ or $\text{Vor}(R)$). As in [Ki], we can show that the sum over all K of the quantities $n_1^{(K)} + n_2^{(K)}$ is $O(n)$. Namely we have the following lemma.

LEMMA 4.2. *Each Voronoi spoke (in either $\text{Vor}(L)$ or $\text{Vor}(R)$) is intersected by the contour C in at most one point, Moreover this remark also holds for any segment one of whose endpoints is the center x_i of some Voronoi cell $V(i)$ (in either diagram) and which is wholly contained in $V(i)$.*

Proof. Let e be a Voronoi spoke of a cell $V(i)$ in $\text{Vor}(L)$. Then each point z at which e and C intersect each other must lie on the boundary of the cell $V(i)$ in $\text{Vor}(S)$, and since this cell is star-shaped with respect to x_i it follows that at most one such intersection point can exist. Q.E.D.

COROLLARY. *The total time required to trace the contour, given a point z (and the two subcells in $\text{Vor}(L)$, $\text{Vor}(R)$ containing it) on each of its connected components, is $O(n)$.*

Proof. The total number of edges on C is $O(n)$, because C is a subset of $\text{Vor}(S)$. The total number of intersections of C with Voronoi spokes is also $O(n)$, by Lemma 4.2. Hence the total complexity of the tracing procedure applied to each component K of C is

$$\sum_K (n_1^{(K)} + n_2^{(K)}) = O(n). \quad \text{Q.E.D.}$$

The problem that now remains is that of finding a representative point z_K on each component K of the contour C (and also finding the subcells containing z_K). For this, Kirkpatrick [Ki] uses a technique which traces edges of minimum spanning trees for R and L . However, this technique, which works nicely for a set S of *points*, is not easily generalizable to sets of more general objects like the circles which now concern us. We will therefore present an alternative approach, which works for sets of circular discs, but whose complexity is $O(n \log n)$, instead of the linear complexity of Kirkpatrick's technique.

We iterate over all the cells $V(i)$ of $\text{Vor}(R)$ (and the corresponding circles B_i), proceeding as follows. Let $B_j \in L$ be such that $x_i \in V(j)$ (in $\text{Vor}(L)$) so that B_j is the circle of L which is "closest" to x_i . If B_j contains all of B_i , then, by definition, the final diagram $\text{Vor}(S)$ will not contain a cell $V(i)$. In this case we simply do not use the pair x_i, x_j to find a point on the contour, but go on to consider the other discs $B_l \in R$.

Next suppose that B_j does not contain all of B_i . Then no other disc $B_k \in L$ can have this property, and so it follows that the disc B_l in S for which $d(x_i, x_l) - r_l$ reaches its minimum belongs to R . Similarly, $d(x_j, x_l) - r_l$ attains its minimum when $l = j$, since no disc in R can contain the whole of a disc in L , and since the recursive construction of $\text{Vor}(L)$ described in the following paragraphs will eliminate discs that are wholly contained in other discs. Thus the segment $e = x_i x_j$ must contain a point z for which $d(z, x_i) - r_i$ reaches its minimum simultaneously for some $B_l \in L$ and for some other

disc $B_r \in R$, which is to say, a point z on C . Moreover, since e emanates from x_j and is wholly contained in $V(j)$, it follows by Lemma 4.2 that the contour C cannot intersect e in more than one point. Hence e intersects C in exactly one point. Note that the entire segment e is contained in a single subcell of $V(j)$ in $\text{Vor}(L)$. All that we have to show is that we can either find z , or assure ourselves that a point on the same component K (of C) as z has already been found, in total time $O(n \log n)$.

We can proceed to find the unique intersection z of e with C using a technique quite similar to the contour-tracing procedure described above. That is, we first find the Voronoi subcell (of $V(i)$) in $\text{Vor}(R)$ containing points on e near x_i (since e emerges from x_i , this amounts to finding the two Voronoi spokes of $V(i)$ between which e lies). We then find the intersection of e with the Voronoi edge bounding that subcell, beyond which e crosses into another subcell of $\text{Vor}(R)$. (In the extreme case in which e coincides with a Voronoi spoke of $V(i)$, e will exit the two subcells of $V(i)$ in which it lies at the Voronoi vertex which is the other endpoint of the spoke; it is then a bit more complicated, but still straightforward, to determine the Voronoi subcell into which e enters past this vertex.) Continuing in this manner, we partition e into subsegments e_1, \dots, e_m , each of which is contained in some Voronoi subcell of $\text{Vor}(R)$ or lies along a spoke of $\text{Vor}(R)$. As all this is done, we keep track of all the cells $V(k)$ of $\text{Vor}(R)$ that have already been encountered. If such a cell is encountered for the second time, tracing of the sequence of edges e_1, \dots, e_m stops immediately (this rule is justified by Lemma 4.3 below). This guarantees that the total cost of traversing subcells of $\text{Vor}(R)$ is bounded by the total number of such subcells, and hence by $O(n)$.

Remark. It is well to compare this tracing technique with the technique of Lee and Drysdale [LD] for locating points on the contour. First of all there is an inaccuracy in the description of their algorithm, even in the case of line segments. Namely (cf. [LD, top of p. 83]) they claim that if t is the nearest point on some segment of L to an endpoint q of a segment in R , then the nearest endpoint of a segment in R to t is q itself. This however is false, as can be easily checked. Lee and Drysdale correct themselves later by considering the midpoint m of the segment qt rather than t itself. In the case of circles, however, they also state an inaccurate statement of this form, but do not correct themselves. In the case of disjoint circles, using the midpoint of the segment uv (in the terminology of [LD, p. 86]) will result in a correct algorithm; it is not clear whether this amendment will also work in the case of intersecting circles.

For each $t = 1, \dots, m$ let $V(i_t)$ be the cell in $\text{Vor}(R)$ containing e_t . As tracing proceeds through the cell $V(i_t)$, we check whether $V(i_t)$ has been encountered before, and, if not, whether there exists $z \in e_t$ such that $d(z, B_j) = d(z, B_{i_t})$ (this can be done in constant time). As already shown, there will exist a unique point z on e having this property, and this z will be the required intersection point of e with C (the algorithm will reach this z only if tracing is not abandoned earlier, because a previously encountered cell of $\text{Vor}(R)$ is encountered again). Let e_s be the subsegment of e containing z . Note that z is found in time $O(p + s)$, where p is the number of subcells of $V(i)$ in $\text{Vor}(R)$.

To show that tracing of e can be abandoned as soon as any cell of $\text{Vor}(R)$ is encountered for the second time, we will use the following:

LEMMA 4.3. *Let s be as in the preceding paragraph. Then for each $t \leq s$, $\text{Vor}(S)$ contains a cell whose center is x_{i_s} , and the point x_{i_t} lies in the same R -region as x_i .*

Proof. Let M be the R -region containing $x_{i_1} = x_i$, and let $1 < t \leq s$. Pick any point $z_t \in e_t$ (but in case $t = s$, z_t must lie between x_i and z). Since $t \leq s$, the segment $x_i z_t$ does not intersect the contour, and therefore is wholly contained in M . As always, let B_{i_t} be the disc in L corresponding to the cell $V(i_t)$ of $\text{Vor}(R)$, and let x_{i_t} be its center.

Suppose for the moment that we have already shown that a cell $V(i_t)$ (with center x_{i_t}) appears in $\text{Vor}(S)$, i.e. that B_{i_t} is not wholly contained in any disc of L . The segment $J = z_t x_{i_t}$ is contained in the cell $V(i_t)$ of $\text{Vor}(R)$, which is star-shaped with respect to its center x_{i_t} , by property (4) of § 3. Moreover, since this segment emanates from x_{i_t} , it can intersect the contour in at most one point, by Lemma 4.2. But such an intersection is impossible, because both the endpoints of J lie in an R -region (x_{i_t} lies in an R -region because by assumption it belongs to $V(i_t)$ in $\text{Vor}(S)$). Therefore the polygonal path $x_{i_t} z_t x_{i_t}$ does not intersect the contour. But x_{i_t} and x_{i_t} are connected to each other via this path, and hence lie in the same R -region.

It only remains to show that a cell $V(i_t)$ with center x_{i_t} appears in $\text{Vor}(S)$. For this, note that the point z_t lies in an R -region, so that it is nearer to some disc of R than to any disc of L . From this it is plain that z_t is nearer to B_{i_t} than to any other disc of S . But then z_t must be an interior point of the cell $V(i_t)$ in $\text{Vor}_0(S)$, so that, by definition of $\text{Vor}(S)$, a cell $V(i_t)$ with center x_{i_t} appears in this diagram, as asserted. Q.E.D.

As we apply the procedure just described to each of the discs $B_i \in R$, one of the following three situations will arise: either

(a) We discard B_i immediately, because the nearest disc B_j in L wholly contains B_i ; or

(b) While tracing subcells of $\text{Vor}(R)$ crossed by the segment $e = x_i x_j$ (where $B_j \in L$ is the disc for which $x_i \in V(j)$), we encounter a subcell of some cell $V(r)$ whose R -region M in $\text{Vor}(S)$ was encountered before. In this case we conclude from Lemma 4.3 that B_i , as well as every other disc of R whose cell in $\text{Vor}(R)$ has been crossed by e so far, lies in the R -region M . In this case we can stop tracing e and go on to process other discs of R , since we can be sure that the component of the contour C intersected by e has already been explored. The algorithm will also note that all cells of $\text{Vor}(R)$ crossed by e so far belong to M , to avoid repeated processing of these cells later on. (note that this case will arise only when $V(i)$ is an inner cell in M , i.e. a cell not intersected by the contour); or

(c) The tracing procedure continues until an intersection z of e with the contour is found. In this case only new cells of $\text{Vor}(R)$ are being traced, and z will lie on a new component of C (this component must be new, because all the old components of C have already been traced, and all the R -subcells through which they pass have already been encountered and marked; hence before reaching any old component of the contour the scanning will stop by step (b) above). As in (b), we take note of the fact that all cells crossed by e during this tracing belong to the new R -region just found, to avoid repeated processing of these cells later on.

These observations imply that we can find a representative point on each component of the contour in total time $O(n)$, provided that, for each $B_j \in R$, the subcell of the cell $V(i)$ of $\text{Vor}(L)$ containing x_j is already known.

To obtain this final item of information, we can use a simple plane-sweeping algorithm, similar to those described by [Sh], [BO], [NP]. The algorithm sweeps the plane from left to right and maintains a vertical "front" $T(a)$ consisting of the segments lying along the line $x = a$ and delimited by the points of intersection of this line with the edges and spokes of $\text{Vor}(L)$. The structure of $T(a)$ will change only at points a which are either abscissae of Voronoi vertices of $\text{Vor}(L)$, or abscissae of centers of discs in L , or points for which the line $x = a$ is tangent to some Voronoi edge of $\text{Vor}(L)$. The number of such "transition points" is plainly $O(n)$, and the total number of segments of $T(a)$ is also $O(n)$ for any real a . To start the algorithm, sort the set A , consisting of all transition points of $\text{Vor}(L)$ and all centers x_i of discs $B_i \in R$, by

their x -coordinates, and initialize the list $T(a)$ as a 2-3 tree for some large enough negative real a . Both these tasks can be done in time $O(n \log n)$. Then scan A from left to right. For each $a \in A$, if a is a transition point of $\text{Vor}(L)$, update the list T by an appropriate combination of deletions, insertions, and merge operations applied to segments in T ; this can be done in time $O(k_a \log n)$, where k_a is the number of segments which undergo these changes. Note that if a is the abscissa of a center x_i of some disc $B_i \in L$, then k_a is the number of Voronoi edges on the boundary of the cell $V(i)$ in $\text{Vor}(L)$, which may be large. Nevertheless, the total sum of all the k_a 's over all transition points a is always $O(n)$. If a is the abscissa of a center c of some disc in R , search T to find the segment in T containing x_i , from which the Voronoi subcell of $\text{Vor}(L)$ containing c is readily obtained. Proceeding in this way, we locate all the centers of discs of R in $\text{Vor}(L)$ in time $O(n \log n)$.

Together, the details just described yield the following algorithm for constructing $\text{Vor}(S)$:

1. Split S into two equal-size subsets L, R such that the leftmost point of each $B_i \in L$ lies to the left of the leftmost point of every $B_j \in R$ (we have assumed that no two leftmost points have the same abscissa).
2. Compute $\text{Vor}(L)$ recursively.
3. Apply the plane-sweeping procedure described above to locate the subcell $V(j)$ of $\text{Vor}(L)$ containing x_i for all centers x_i of discs $B_i \in R$. Discard the disc $B_i \in R$ if it is wholly contained in B_j .
4. Let R' be the remaining set of discs of R . Compute $\text{Vor}(R')$ recursively.
5. Construct the "contour" C as follows:

For each disc $B_i \in R$ whose R -region (in $\text{Vor}(S)$) has not yet been identified

- a. Connect x_i to the center x_j of the disc $B_j \in L$ whose Voronoi cell $V(j)$ in $\text{Vor}(L)$ contains x_i . If B_j wholly contains B_i , then $\text{Vor}(S)$ will not contain a cell corresponding to B_i , and we go on to process other discs of R .
 - b. Find the unique intersection z of the contour with the segment $e = x_i x_j$ by applying the tracing procedure described above. If that procedure detects an intersection of e with a cell $V(k)$ of $\text{Vor}(R)$ whose R -region M has already been found, it assigns M as the R -region of B_i and of all other discs of R whose cells in $\text{Vor}(R)$ have been crossed by e before $V(k)$ has been reached, and continues with the main loop of this phase.
 - c. Trace the whole contour component K containing z . An R -region indication is thereby assigned to all discs $B_k \in R$ whose cells in $\text{Vor}(R)$ are encountered.
6. Obtain the final diagram $\text{Vor}(S)$ by taking the union of $\text{Vor}(R)$ and of $\text{Vor}(L)$ with C , and then by discarding (portions of) edges of $\text{Vor}(R)$ or of $\text{Vor}(L)$ which are cut off from their cells by C .

The algorithm just sketched runs in time $O(n \log^2 n)$. Its costliest phase is step 3, which locates subcells of $\text{Vor}(L)$ containing the centers of discs of R .

Note that it is a simple matter to modify the algorithm so that it also produces a mapping *contain*, which, for each disc B_i deleted from the diagram by the algorithm, gives the disc B_j containing B_i as found in step 5a of the algorithm.

5. Applications of the generalized Voronoi diagram; possible extensions. In this section we show how the generalized Voronoi diagram can be used to solve some of the intersection problems mentioned in the introduction to this paper.

Suppose that the generalized diagram $\text{Vor}(S)$ for a set S of circular discs has been constructed, and that it is represented by the data structures described in § 4.

First consider the problem of detecting the existence of an intersection between any pair of discs in S . This can be tested using the following procedure: First check whether there exists a Voronoi cell $V(i)$ having empty interior (i.e. a cell which is missing from $\text{Vor}(S)$). If so, B_i is wholly contained in some other disc, and an intersection has been found. Otherwise, for each edge e in $\text{Vor}(S)$ belonging to the common boundary of two Voronoi cells $V(i)$ and $V(j)$, compute the value

$$\rho(e) = \min \{d(x_i, y) - r_i : y \in e\} = \min \{d(x_j, y) - r_j : y \in e\}.$$

If $\rho(e) \leq 0$ for some $e \in \text{Vor}(S)$, then it is clear that the discs in S intersect. On the other hand, if $\rho(e) > 0$ for each $e \in \text{Vor}(S)$, then no two discs in S intersect. Indeed, suppose that two discs B_i and B_j intersect each other. Let I be the segment $[x_i, x_j]$. For each $y \in I$ consider the function

$$f(y) = \min \{d(x_k, y) - r_k : k = 1, \dots, n\}.$$

Note that $f(y) \leq 0$ for each $y \in I$, because each such y lies either in B_i or in B_j , so that either $d(x_i, y) - r_i$ or $d(x_j, y) - r_j$ is ≤ 0 . Moreover, by assumption both $V(i)$ and $V(j)$ have nonempty interiors, which implies by (2) that x_i belongs to $V(i)$ and to no other cell. Hence I must intersect $\text{Vor}(S)$ at least once. Let $y \in I$ be a point belonging to some edge e of $\text{Vor}(S)$, and let $V(k), V(l)$ be the two Voronoi cells containing e in their boundary. Then we have

$$\rho(e) \leq d(x_k, y) - r_k = f(y) \leq 0,$$

from which our claim follows immediately.

It is easy to see that a simple modification of the procedure just outlined yields a solution to the more complicated problem in which the discs in S are of several colors and we want to detect intersection between two discs of different colors. The appropriate procedure in this case is:

(a) First check whether there exists a disc B_i which is wholly contained in another disc B_j of a different color, i.e. if $V(i)$ has empty interior, and x_i belongs to a cell $V(j)$ of a disc with a different color. Once the Voronoi diagram has been constructed by the method described in the preceding section, and has been supplemented by the mapping *contain*, we can detect such cases in $O(n)$ time. Note that not every containment of a disc B_i of, say, red color in another disc B_j of a different color can be detected from the *contain* map, because B_j might be contained in another red disc B_k , and *contain* may map B_i directly to B_j . Nevertheless, if B_i is the *largest* possible disc contained in a disc of a different color, then *contain* will map B_i to some *differently-colored* disc containing it, so that if any disc is wholly contained in a disc of a different color, the procedure just described will detect at least one such situation.

(b) If step (a) detects no intersection, compute the quantities $\rho(e)$, as defined above, for all edges $e \in \text{Vor}(S)$. Then two discs of different colors intersect each other if and only if there exists an edge e common to two cells $V(i)$ and $V(j)$, for which B_i and B_j have distinct colors, such that $\rho(e) \leq 0$.

Proof. Plainly if $\rho(e) \leq 0$ for such an edge e , then e must contain a point y which lies inside both B_i and B_j , so that these two differently-colored discs intersect each other. Conversely, suppose that two differently-colored discs B_i and B_j intersect each other. Define the segment I and the function f on it as in the preceding paragraphs. We can assume without loss of generality that $V(i)$ and $V(j)$ have nonempty interiors, for if B_i (or B_j) had empty interior then it would have been wholly contained in another

disc B_k of the same color, so that we could replace B_i (or B_j) by B_k in what follows. Call the colors of B_i, B_j "red" and "green", and call a Voronoi cell $V(p)$ a "red" (resp. "green") cell if B_p is colored red (resp. green). Then x_i lies in (the interior of) a red cell, whereas x_j does not. Hence I must intersect $\text{Vor}(S)$ at an edge e which separates a red cell $V(k)$ and a cell $V(l)$ of a different color. Arguing as before, it follows that $\rho(e) \leq 0$ for this edge. Q.E.D.

Next consider the problem of determining the shortest distance between any two discs in S . Suppose that the discs in S do not intersect each other (if they do, the above procedures will detect this fact, and the distance that we seek will be 0), and let B_i, B_j be the two discs closest to each other among all pairs of discs in S . Let y be the point on the segment $I = [x_i, x_j]$ equidistant from B_i and B_j . We claim that $y \in \text{Vor}(S)$. For otherwise, there would exist another disc $B_k \in S$ such that $d(x_k, y) - r_k < d(x_i, y) - r_i$. But then, by the triangle inequality,

$$\begin{aligned} d(B_j, B_k) &= d(x_j, x_k) - r_j - r_k \leq d(x_j, y) - r_j + d(x_k, y) - r_k \\ &< d(x_j, y) - r_j + d(x_i, y) - r_i = d(x_i, x_j) - r_i - r_j = d(B_i, B_j), \end{aligned}$$

contrary to assumption. Thus $y \in \text{Vor}(S)$. Moreover, the function

$$f(z) = \min \{d(x_k, z) - r_k : k = 1, \dots, n\}$$

attains its minimum value on the whole Voronoi diagram $\text{Vor}(S)$ at the point y . This follows since by the triangle inequality we have $2f(z) \geq d(B_k, B_i)$ for each $z \in \text{Vor}(S)$, where $V(k)$ and $V(l)$ are the two Voronoi cells containing z . It follows by the definition of f and by the preceding definition of $\rho(e)$ that $f(y)$ is the smallest of the values $\rho(e)$, for edges e of $\text{Vor}(S)$. Taken together, these arguments show that the shortest distance between two discs in S is equal to

$$\min \{2\rho(e) : e \text{ an edge of } \text{Vor}(S)\},$$

and hence this distance can be found in time $O(n \log^2 n)$.

A similar technique can be used to find the nearest neighbor in S of each $B_i \in S$. Indeed, an easy generalization of the preceding argument implies that if B_j is the nearest neighbor of B_i , then $V(i)$ and $V(j)$ meet at a common Voronoi edge, and the shortest distance between B_i and any other disc in S is equal to

$$\min \{2\rho(e) : e \text{ a boundary edge of } V(i)\},$$

from which the nearest neighbor of B_i is easily found.

These arguments extend easily to the case in which each of the discs of S is assigned a certain color, and we want to find the shortest distance between any two differently-colored discs. For this, let B_i and B_j be two discs in S of different colors such that their distance is the smallest among all distances between two differently-colored discs in S . Let the colors of B_i, B_j be "red" and "green" respectively. Let y be the point on $[x_i, x_j]$ equidistant from B_i and B_j . We claim that $y \in \text{Vor}(S)$, for otherwise there would exist another disc B_k such that $d(x_k, y) - r_k < d(x_i, y) - r_i = d(x_j, y) - r_j$. If B_k is colored red, then arguing as above we would obtain $d(B_j, B_k) < d(B_i, B_j)$, i.e. a shorter distance between a red and a green disc. Similarly, if B_k is colored green, then we would have $d(B_i, B_k) < d(B_i, B_j)$, again a contradiction. Finally, if B_k is of another color, then both $d(B_i, B_k)$ and $d(B_j, B_k)$ are smaller than $d(B_i, B_j)$, again a contradiction.

Thus $y \in \text{Vor}(S)$, and similar arguments to those used above imply that $f(y)$ is the minimum of all $\rho(e)$, for edges e of $\text{Vor}(S)$ separating cells of different colors.

This shows that the shortest distance between two differently-colored discs in S is simply the smallest of the values $2\rho(e)$, taken over all edges e of $\text{Vor}(S)$ separating two differently-colored cells.

We next consider Problem IV listed in the introduction. For this we apply Kirkpatrick's algorithm [Ki2] for fast searching in planar subdivisions to the subdivision of the plane into Voronoi cells. This algorithm applies in our case since each Voronoi cell is star-shaped with respect to the corresponding disc center. The additional pre-processing cost is only $O(n)$ (after $\text{Vor}(S)$ has been constructed), and then one can find the Voronoi cell containing any specified point in time $O(\log n)$.

Using the Voronoi diagram $\text{Vor}(S)$ has enabled us to solve some of the problems noted in the introduction, but not all of them. In particular, we would like to use these methods for solving the second part of Problem III (that is, to find the differently-colored disc nearest to any given disc in S), and for solving Problem IV. Concerning Problem III, we note that in some special cases straightforward generalizations of the techniques presented above can be used to solve this problem. This is the case for example if each color class has a constant size. Note also that the following partial solution to the second part of Problem III is available in general: Consider the subgraph H of the dual graph D^* of $\text{Vor}(S)$ defined so that x_i and x_j are adjacent in H if and only if (they are adjacent in D^* and) B_i and B_j have the same color. Let P be a connected component of H (i.e. a "clustering" of discs having the same color). Then the shortest distance between some disc whose center appears in P and a differently colored disc can be found by tracing all Voronoi edges which separates a disc in P from a disc with a different color, using the same technique given above. In many applications partial results of this form are sufficient.

6. Computing the area of a union of discs. Let us consider next another interesting application of the generalized Voronoi diagram for discs. Let S be a set of n possibly intersecting discs in the plane, and let K denote the union of all discs in S . The problem at hand is to compute efficiently the area of K . We will show that, once the generalized diagram $\text{Vor}(S)$ has been computed, the area of K can be computed in linear time. More specifically, we will show that K can be decomposed in linear time into $O(n)$ disjoint subparts, each being either a circular sector or a quadrangle, so that the area of each such subpart can be readily calculated, and will be assumed to require constant time to calculate. This gives us the stated linear time bound.

Suppose that $\text{Vor}(S)$ has already been computed. We partition each cell $V(i)$ of $\text{Vor}(S)$ into subcells as in § 4, by connecting each vertex in $V(i)$ to x_i by a straight "spoke". Evidently there are overall $O(n)$ such subcells. We will make use of the following simple lemma.

LEMMA 6.1. *For each $B_i \in S$, we have $V(i) \cap K \subset B_i$.*

Proof. Let $y \in V(i) \cap K$, and suppose that $y \in B_j$ for some B_j in S . Since $y \in V(i)$ we have

$$d(x_i, y) - r_i \leq d(x_j, y) - r_j \leq 0,$$

so that $y \in B_i$ too. Q.E.D.

Let U be a subcell of some cell $V(i)$. The boundary of U consists of two spokes connecting x_i to two Voronoi vertices ν_1, ν_2 , and of a unique Voronoi edge e separating $V(i)$ from an adjacent cell $V(j)$ and connecting ν_1 and ν_2 (if e is an unbounded edge, ν_1 or ν_2 may lie at infinity). Suppose without loss of generality that x_i lies at the origin. By the preceding lemma, $U \cap K$ in polar coordinates is

$$U \cap K = \{(r, \theta) : \theta_1 \leq \theta \leq \theta_2, 0 \leq r \leq \min(r_i, e(\theta))\},$$

where θ_k is the orientation of the spoke connecting x_i to ν_k , $k = 1, 2$, and where $e(\theta)$ is the length of the segment at orientation θ connecting x_i to e .

It is now plain that $U \cap K$ can be decomposed into at most three subparts, each of which is either a circular sector (whose area is readily calculable) or a sector of the form

$$A = \{(r, \theta) : \theta' \leq \theta \leq \theta'', 0 \leq r \leq e(\theta)\}.$$

Although the area of such a sector can be easily calculated by straightforward integration, the formula giving this area is somewhat complicated and involves trigonometric and logarithmic terms. However we can avoid explicit calculation of the area of such sectors as follows. Let A be such a sector, and let y, y' be the two endpoints of the portion of the corresponding Voronoi edge e within A . Note that e separates the cell $V(i)$ containing A and an adjacent cell $V(j)$, so that when $V(j) \cap K$ is decomposed into its subparts, one of them will consist of another sector A' bounded by e and by the two segments $x_j y, x_j y'$. Hence, instead of computing the areas of A, A' separately, we can compute directly the area of $A \cup A'$, which is simply the quadrangle $x_i y x_j y'$.

We have thus shown that K can be decomposed into $O(n)$ subparts, each being either a circular sector or a quadrangle. It is also plain that this decomposition can be accomplished in linear time by a simple traversal of the Voronoi diagram, once the diagram has been calculated. Hence we have:

THEOREM 6.1. *The area of the union of n arbitrary discs can be calculated in time $O(n \log^2 n)$, as the sum of $O(n)$ circular sectors and quadrangles.*

Remarks.

(1) It is also conceivable that this area could be calculated efficiently by some sweeping technique. This is suggested by the fact that the number of points of intersection of pairs of circles in S , which lie on the boundary of K , is $O(n)$ (indeed, each such point lies on one of the $O(n)$ edges of $\text{Vor}(S)$, and each of these edges can contain at most two such points). Thus if we could perform a sweep in which only those intersection points are traced, while the other $O(n^2)$ "inner" intersection points are ignored, we could obtain the area of K as the sum of the areas of $O(n)$ vertical strips, each bounded by an upper and a lower circular arc. However, we do not know how to achieve this without calculation of the associated Voronoi diagram.

(2) A recent result of Spirakis [Sp] gives a Monte-Carlo algorithm for calculating the area of K in expected linear time.

7. Conclusion. In this paper we have introduced the notion of generalized Voronoi diagram for a set of n possibly intersecting circles in the plane, described an $O(n \log^2 n)$ algorithm for constructing this diagram, and presented several applications of this diagram, for detecting intersections and calculating proximities on one hand, and for calculating the area of the union of these circles on the other.

Other properties of these generalized Voronoi diagrams deserve study. Some additional problems concerning applications of this diagram have been noted in § 5. Another interesting problem is to find efficient techniques for dynamic maintenance of the generalized Voronoi diagram of a set S of circular objects, when one or more objects move continuously along specified trajectories, e.g. along straight lines. An efficient solution of this latter problem would facilitate efficient procedures for checking a prescribed motion of a robot system for collisions; the current alternative is to apply an appropriate static collision-detecting procedure, as described in § 5, to a sequence of configurations of the system, sufficiently close to each other, along the specified trajectory.

It would also be useful to generalize the techniques described in this paper to the case of a set S of objects other than circles, e.g. general convex bodies, or special forms of convex bodies, such as “cigar-shaped” displacements of straight segments, etc. A major problem in attempting such generalizations is that if (as we would like to do) we allow objects in S to intersect each other, then the corresponding Voronoi diagram may contain up to $O(n^2)$ cells (consider e.g. the case of n straight segments intersecting each other at $O(n^2)$ points). Note here that it is a remarkable property of *circular* bodies that the size of their Voronoi diagram is always linear, even though they can cut each other into $O(n^2)$ regions. A final open problem is whether the generalized Voronoi diagram for n circular discs can be constructed in time $O(n \log n)$ (even in the simpler case in which the discs are disjoint from each other).

Acknowledgment. The author wishes to thank Jacob Schwartz for suggesting the problems discussed in this paper, and for carefully reviewing the manuscript. His comments on the manuscript have led to substantial improvements in its presentation.

REFERENCES

- [BO] J. L. BENTLEY AND T. A. OTTMAN, *Algorithms for reporting and counting geometric intersections*, IEEE Trans. Comp., C-28 (1979), pp. 643–647.
- [D] R. L. DRYSDALE III, *Generalized Voronoi diagrams and geometric searching*, Ph.D. thesis, Dept. Computer Science, STAN-CS-79-705, Stanford Univ., Stanford, CA, 1979.
- [DL] R. L. DRYSDALE III AND D. T. LEE, *Generalized Voronoi diagrams in the plane*, Proc. 16th Annual Allerton Conference on Communications, Control and Computing, 1978, pp. 833–842.
- [HSS] J. E. HOPCROFT, J. T. SCHWARTZ AND M. SHARIR, *Efficient detection of intersection among spheres*, Robotics Research, 2 (1983) (4), pp. 77–80.
- [Ki] D. KIRKPATRICK, *Efficient computation of continuous skeletons*, Proc. 20th Symposium on Foundations of Computer Science, 1979, pp. 18–27.
- [Ki2] ———, *Optimal search in planar subdivisions*, this Journal, 12 (1983), pp. 28–35.
- [LD] D. T. LEE AND R. L. DRYSDALE III, *Generalizations of Voronoi diagrams in the plane*, this Journal, 10 (1981), pp. 73–87.
- [NP] J. NIEVERGELT AND F. P. PREPARATA, *Plane-sweeping algorithms for intersecting geometric figures*, Comm. ACM, 25 (1982), pp. 739–747.
- [Pr] F. P. PREPARATA, *A new approach to planar point location*, this Journal, 10 (1981), pp. 473–482.
- [Sh] M. I. SHAMOS, *Computational geometry*, Ph.D. Dissertation, Yale Univ., New Haven, CT, 1975.
- [SS] J. T. SCHWARTZ AND M. SHARIR, *On the piano movers’ problem: I. The special case of a rigid polygonal body moving amidst polygonal barriers*, Comm. Pure Appl. Math., 35 (1983), pp. 345–398.
- [Sp] P. SPIRAKIS, *Very fast algorithms for the area of the union of many circles*, Tech. Rept., Computer Science Dept., Courant Institute, New York Univ., New York, December 1983.

POLYNOMIAL-TIME REDUCTIONS FROM MULTIVARIATE TO BI- AND UNIVARIATE INTEGRAL POLYNOMIAL FACTORIZATION*

ERICH KALTOFEN†

Abstract. Consider a polynomial f with an arbitrary but fixed number of variables and with integral coefficients. We present an algorithm which reduces the problem of finding the irreducible factors of f in polynomial-time in the total degree of f and the coefficient lengths of f to factoring a univariate integral polynomial. Together with A. Lenstra's, H. Lenstra's and L. Lovász' polynomial-time factorization algorithm for univariate integral polynomials [Math. Ann., 261 (1982), pp. 515-534] this algorithm implies the following theorem. Factoring an integral polynomial with a fixed number of variables into irreducibles, except for the constant factors, can be accomplished in deterministic polynomial-time in the total degree and the size of its coefficients. Our algorithm can be generalized to factoring multivariate polynomials with coefficients in algebraic number fields and finite fields in polynomial-time. We also present a different algorithm, based on an effective version of a Hilbert Irreducibility Theorem, which polynomial-time reduces testing multivariate polynomials for irreducibility to testing bivariate integral polynomials for irreducibility.

Key words. polynomial factorization, polynomial-time complexity, algorithm analysis, Hensel lemma, Hilbert irreducibility theorem

1. Introduction. Both the classical Kronecker algorithm [17, p. 10] (see also van der Waerden [28, pp. 136-137]) and the modern multivariate Hensel algorithm (cf. Musser [26], Wang [29], Zippel [35]) solve the problem of factoring multivariate polynomials with integral coefficients by reduction to factoring univariate integral polynomials and reconstructing the multivariate factors from the univariate ones. However, as we will see in § 3, the running time of both methods suffers from the fact that, in rare cases, a number of factor candidates obtained from the univariate factorization which is exponential in the input degree may have to be tried to determine the true multivariate factors. In this paper we will present a new algorithm which does not require exponential-time in its worst case. But before we can state our result precisely, we need to clarify what we mean by input size. We will assume that our input polynomials are densely encoded, that is all coefficients including zeros are listed. Hence, the size of a polynomial with v variables, given that the absolutely largest coefficient has l digits and the highest degree of any variable is n , is of order $O(l(n+1)^v)$.

Let v , the number of variables, be a fixed integer. We will show that the problem of determining all irreducible factors of v -variate polynomials is polynomial-time (Turing-, Cook-) reducible to completely factoring univariate polynomials. Recently, A. Lenstra, H. Lenstra, and L. Lovász [22] have shown that factoring univariate rational polynomials is achievable in polynomial-time. Therefore, our result implies the following theorem. Factoring an integral polynomial with a fixed number of variables into irreducibles, except for the constant factors, can be accomplished in deterministic polynomial time in the total degree and the size of its coefficients. Our algorithm is a multivariate version of an algorithm due to H. Zassenhaus [34], which, instead of leading to an integer linear programming problem, as is the case for Zassenhaus' algorithm, leads to a system of linear equations for the coefficients of an irreducible multivariate factor.

* Received by the editors March 15, 1983, and in revised form April 9, 1984. This work was partially supported by the National Science Foundation under grant MCS-7909158 and by the Department of Energy, under grant DE-AS02-ER7602075.

† Department of Computer Science, University of Toronto, Toronto, Ontario M5S 1A4, Canada. Current address: Department of Mathematical Sciences, Rensselaer Polytechnic Institute, Troy, New York 12181.

In Kaltofen [12] we have already established a polynomial-time reduction from multivariate to bivariate polynomial factorization. However, our new algorithm is less complex. On the other hand, the results in Kaltofen [12] imply a polynomial-time (m-, Karp-) reduction for irreducibility testing, which our new algorithm does not provide. This older algorithm (cf. § 7) is based on an effective version of a Hilbert Irreducibility Theorem [11] (see also Franz [5]), an idea which since has been used successfully in von zur Gathen [7] to construct a probabilistic algorithm for factoring sparse multivariate polynomials with a growing number of variables, and in Chistov and Grigoryev [3] to provide another polynomial-time reduction from bivariate to univariate integral polynomial factorization.

If one does not fix the number of variables, our definition of input size may not be appropriate since the input size then grows exponentially with the number of variables. Although our algorithm remains polynomial in the expression $l(n+1)^v$, our size measure only applies to dense inputs and for sparse polynomials our algorithm is of exponential complexity in the number of variables.¹ Unfortunately, in the sparse case little is known about even the space complexity of the answer under these conditions. In § 8 open problem 1 corresponds to this question.

The question arises whether our algorithm is of practical importance. Unlike in the univariate case, in the multivariate Hensel algorithm the factors of the reduced univariate polynomial are almost always the true images of the multivariate factors, in which case no exponential running time occurs. This empirical observation can be explained by a distributive version of the Hilbert Irreducibility Theorem (cf. § 3) but there seems to be no known guarantee that one can always avoid bad reductions in polynomial-time. However, we like to point out that so far we know of no class of polynomials for which our polynomial-time algorithm could perform better than the standard multivariate Hensel algorithm. In this connection we state open problem 2 in § 8.

In this paper we only consider the problem of multivariate polynomial factorization with integral coefficients. However, the presented algorithms can be generalized to coefficient domains such as algebraic extensions of the rationals as well as finite fields. Besides outlining the necessary ideas in § 8 we refer to the papers by Chistov and Grigoryev [3], Landau [19], von zur Gathen and Kaltofen [8], and Lenstra [20] and [21].

We shall briefly outline the organization of this paper. Section 2 establishes our notation and some well-known facts about polynomials. Exponentially bad cases for both the Kronecker and the multivariate Hensel algorithm are then constructed in § 3. In § 4 we introduce some well-known preliminary transformations on our input polynomials and also establish that these transformations are polynomial-time reductions. The main algorithm is presented in § 5 including the necessary arguments for its correctness. Its complexity is analyzed in § 6. In particular we show that the size of all intermediately computed integers stays within polynomial bounds. An effective version of the Hilbert Irreducibility Theorem and its applications to the factorization problem are discussed in § 7. We conclude in § 8 by raising 3 open problems.

2. Notation. By \mathbf{Z} we denote the set of the integers, by \mathbf{Q} the set of the rational numbers and by \mathbf{C} the set of the complex numbers. \mathbf{Z}_p denotes the set of the residues modulo a prime number p . If D is an integral domain, $D[x_1, \dots, x_v]$ denotes the set of polynomials in x_1, \dots, x_v over D , $D(x_1, \dots, x_v)$ its field of quotients; $\deg_{x_1}(f)$ denotes the highest degree of x_1 in $f \in D[x_1, \dots, x_v]$, $\deg_{x_1, x_2}(f)$ the highest total

¹ Our algorithm remains even polynomial in some slightly sharper input size measures such as $l(d_1 + 1) \cdots (d_v + 1)$ where d_i is the degree of the i th variable.

degree of monomials in x_1 and x_2 in f , and $\deg(f) = \deg_{x_1, \dots, x_v}(f)$ the total degree of f . Thus, $\deg(f)$ is the maximum of all exponent sums of monomials in x_1, \dots, x_v with nonzero coefficients in f . The coefficient of the highest power of x_v in f is referred to as the leading coefficient of f in x_v and will be denoted by $\text{ldcf}_{x_v}(f)$. Notice that $\text{ldcf}_{x_v}(f) \in D[x_1, \dots, x_{v-1}]$. We call f monic in x_v if $\text{ldcf}_{x_v}(f)$ is the unity of D . As is well known, $D[x_1, \dots, x_v]$ is a unique factorization domain (UFD) provided that D is a UFD. In this case the content of $f \in D[x_1, \dots, x_v]$ in x_v , $\text{cont}_{x_v}(f)$, is the greatest common divisor (GCD) of all coefficients of $f(x_v)$ as elements in $D[x_1, \dots, x_{v-1}]$. Notice again that $\text{cont}_{x_v}(f) \in D[x_1, \dots, x_{v-1}]$. The primitive part of f in x_v is defined as

$$\text{pp}_{x_v}(f) = \frac{1}{\text{cont}_{x_v}(f)} f$$

and we call f primitive in x_v if $f = \text{pp}_{x_v}(f)$. We also note that the total degree of a factor of f is less than or equal to the total degree of f . The infinity norm of $f \in \mathbb{C}[x_1, \dots, x_v]$, the maximum of the absolute values of the complex coefficients of f , will be denoted by $|f|$. The square root of the sum of squares of the absolute values of the coefficients of f , the square norm of f , will be denoted by $|f|_2$.

Let $f(x_v) = a_l x_v^l + a_{l-1} x_v^{l-1} + \dots + a_0$ and $g(x_v) = b_m x_v^m + \dots + b_0$ with $a_i, b_j \in D[x_1, \dots, x_{v-1}]$. By $\text{Syl}_{x_v}(f, g)$ we denote the Sylvester matrix of f and g ,

$$\begin{bmatrix} a_l & a_{l-1} & & \dots & & a_1 & a_0 \\ & a_l & a_{l-1} & & \dots & a_2 & a_1 & a_0 \\ & & \ddots & \ddots & & & & \ddots \\ & & & \ddots & a_l & a_{l-1} & \dots & a_m & \dots & a_0 \\ b_m & b_{m-1} & & & b_1 & b_0 & & & & \\ & b_m & b_{m-1} & & & & b_0 & & & \\ & & & \ddots & & & & \ddots & & \\ & & & & & & & & & b_m & b_{m-1} & \dots & b_0 \end{bmatrix}$$

where the empty entries are assumed to be 0 (there are m rows with coefficients of f and l rows with coefficients of g and the matrix has $l+m$ columns). Its determinant is the resultant of f and g with respect to x_v and will be denoted by

$$\text{res}_{x_v}(f, g) = \det(\text{Syl}_{x_v}(f, g)).$$

In order to be able to manipulate with monomials in a short way we adopt the following vector notation: $k \equiv (k_1, \dots, k_v)$, $0 \equiv (0, \dots, 0)$, $y^k \equiv y_1^{k_1} \dots y_v^{k_v}$, $k \pm l \equiv (k_1 \pm l_1, \dots, k_v \pm l_v)$, $k \leq l$ if, for all i , $k_i \leq l_i$ and finally $|k| \equiv k_1 + \dots + k_v$, if $k \geq 0$, and $-\infty$ otherwise. By $\binom{n}{m}$ we denote the binomial coefficient $n!/(m!(n-m)!)$.

3. Exponential cases for the Kronecker and Hensel algorithms. We only consider bivariate polynomials though the constructions easily generalize. First, we discuss some exponential cases for the Kronecker algorithm. This algorithm transforms the bivariate polynomial $f(z, x)$ into $\tilde{f}(y) = f(y^d, y)$, $d = \max(\deg_z(f), \deg_x(f)) + 1$. Since the

degree in z or x of any factor $g(z, x)$ of $f(z, x)$ is less than d , $\bar{g}(y) = g(y^d, y)$, which is a factor of $\bar{f}(y)$, can be used to retrieve $g(z, x)$ in a quick and unambiguous way. Kronecker's algorithm proceeds in transforming all univariate factors of $\bar{f}(y)$ back to bivariate factor candidates for f and then tests whether any candidate is a true factor. However, it clearly requires time exponential in the degree of f in the case where f is irreducible, but \bar{f} splits into linear factors. It is easy to construct such f 's, as we do below, by working backward from $\bar{f}(y)$.

Example 1.

$$\begin{aligned}\bar{f}(y) &= (y-4)(y-3)(y-2)(y-1)(y+1)(y+2)(y+3)(y+4) \\ &= y^8 - 30y^6 + 273y^4 - 820y^2 + 576.\end{aligned}$$

Set $d = 3$: $f_1(z, x) = z^2x^2 - 30z^2 + 273xz - 820x^2 + 576$ which is irreducible. Kronecker's algorithm has to refute 127 factor candidates to determine irreducibility of f_1 .

Set $d = 5$: $f_2(z, x) = x^3z - 30xz + 273x^4 - 820x^2 + 576$ which is irreducible because $\deg_z(f) = 1$. This condition can always be enforced by choosing d large enough and yields exponential cases of arbitrarily high degree.

Example 2. Let $n = (\prod_{i=2}^k p_i) - 2$ with p_i the i th prime number. Let $f_3(z, x) = x^n - z^2$, which is irreducible since n is odd. We obtain $\bar{f}_3(y) = y^n(1 - y^{n+2})$ where $1 - y^{n+2}$ factors into 2^{k-1} cyclotomic polynomials (cf. van der Waerden [28, p. 113]). Since n is of order $O(e^{k \log(k)})$ (cf. Hardy and Wright [10, § 22.2]) the number of possible factor candidates cannot be polynomial in n .

The abundance of univariate factors usually disappears as soon as we choose a slightly different evaluation. For example,

$$f_1(3x^3, x) = 9x^8 - 270x^6 + 819x^4 - 820x^2 + 576$$

and

$$f_2(2x^5, x) = 2x^8 - 60x^6 + 273x^4 - 820x^2 + 576$$

are both irreducible. In Kaltofen [12] we have used a similar evaluation for polynomials with three variables which resulted in a deterministic reduction to bivariate factorization. There we also conjectured that it is highly probable that substituting $2x^d$ or $3x^d$ for z in $f(z, x)$ already preserves the irreducibility of f . However, to prove that a multiplier of polynomial length definitely works seems difficult, and we have only succeeded in showing this for the multivariate to bivariate reduction (cf. § 7, Theorem 3).

In order to give exponentially bad inputs for the multivariate Hensel algorithm we need an irreducible polynomial $f(y_1, \dots, y_b, x)$ such that $f(0, \dots, 0, x)$ has all linear factors. Such a polynomial is quite easy to obtain and the following example demonstrates the construction of a polynomial which has all linear factors for various evaluation points.

Example 3.3. Let $f(y, x)$ have $\deg_y(f) \leq 3$ and

$$f(-1, x) = (x-2)(x-1)(x+1)(x+2) = x^4 - 5x^2 + 4,$$

$$f(0, x) = (x-1)x(x+1)(x+2) = x^4 + 2x^3 - x^2 - 2x,$$

$$f(1, x) = (x-2)(x-1)x(x+1) = x^4 - 2x^3 - x^2 + 2x,$$

and $f(2, x) = x^4 + 2$. By interpolation $f(y, x) \in \mathbf{Q}[y, x]$ is determined uniquely, namely

$$f(y, x) = x^4 + (2y^3 - 3y^2 - 3y + 2)x^3 + \left(\frac{5}{6}y^3 - 2y^2 + \frac{7}{6}y - 1\right)x^2 + (-2y^3 + 3y^2 + 3y - 2)x - \frac{1}{3}y^3 + 2y^2 - \frac{5}{3}y.$$

We can also remove the rational denominators, namely

$$\begin{aligned} \bar{f}(y, x) &= 6^4 f\left(y, \frac{x}{6}\right) \\ &= x^4 + (12y^3 - 18y^2 - 18y + 12)x^3 + (30y^3 - 72y^2 + 42y - 36)x^2 \\ &\quad + (-432y^3 + 648y^2 + 648y - 432)x - 432y^3 + 2592y^2 - 2160y. \end{aligned}$$

Since $f(2, x)$ is irreducible, so is $\bar{f}(y, x)$, but

$$\begin{aligned} \bar{f}(-1, x) &= (x - 12)(x - 6)(x + 6)(x + 12), \\ \bar{f}(0, x) &= (x - 6)x(x + 6)(x + 12), \\ \bar{f}(1, x) &= (x - 12)(x - 6)x(x + 6). \end{aligned}$$

The above construction obviously generalizes for arbitrarily high degrees but the number of unlucky evaluation points (i.e. those integers b for which $f(b, x)$ splits into linear factors) seems bounded by the degree in y . The classical Hilbert irreducibility theorem states that for any irreducible polynomial $f(y, x) \in \mathbf{Z}[y, x]$ there exists an integer b such that $f(b, x)$ remains irreducible. It can be shown that the ratio of unlucky points to the size of the interval, from which the points are taken, tends to zero as the size of the interval goes to infinity (cf. Dörge [4]). The reader is referred to Kaltofen [14, Appendix B] for a short bibliography on the Hilbert Irreducibility Theorem. Unfortunately, we do not understand the distribution of unlucky evaluation points of small size. Open problem 2 in § 8 refers to this question.

4. Initial transformations. In this section we present an algorithm which transforms the problem of factoring the polynomial $\bar{f}(z_1, \dots, z_v, x)$ to factoring a polynomial $f(y_1, \dots, y_v, x)$ such that f is monic in x , $f(0, \dots, 0, x)$ is squarefree, i.e. each of its irreducible polynomial factors occurs with multiplicity 1, and both $\deg(f)$ and $\log(|f|)$ are polynomially bounded in $\deg(\bar{f})$ and $\log(|\bar{f}|)$. For simplicity we only consider finding a single irreducible factor of \bar{f} . In Lemma 2 we will state a uniform coefficient bound for all possible factors of \bar{f} which is of polynomial size in $\deg(\bar{f})$ and $\log(|\bar{f}|)$. Therefore, in order to obtain the complete factorization of \bar{f} into irreducible factors in polynomial-time we can apply our algorithm recursively to the co-factor of the irreducible factor.

We wish to emphasize that this version of our algorithm can be improved significantly, e.g. by resolving the recursion mentioned above. However, here we are most interested in the theoretical result, namely that the algorithm works in polynomial-time. For this reason we also allow ourselves to present rather crude upper bounds in our complexity analysis. We also do not consider the influence which the underlying data structure used to represent multivariate polynomials could have on our algorithm performance. Furthermore, we will formulate the asymptotic complexity as a function in the total degree rather than the maximum degree of individual variables. Since the number of variables is fixed, both notions for the degree are codominant.

The following algorithm computes a squarefree factor of the primitive part of the input polynomial. It then applies two classical transformations to this squarefree factor

to make the polynomial monic and squarefree also when evaluated at 0 for the minor variables.

ALGORITHM 1

[Given $\bar{f}(z_1, \dots, z_v, x) \in \mathbf{Z}[z_1, \dots, z_v, x]$, this algorithm constructs an irreducible factor $\bar{g}(z_1, \dots, z_v, x) \in \mathbf{Z}[z_1, \dots, z_v, x]$ of \bar{f} by preconditioning \bar{f} and calling Algorithm 2.]

(I) [Test for univariate case:]

IF cont(\bar{f}) or pp(\bar{f}) is univariate THEN factor it by a univariate factorization algorithm and return one irreducible factor, ELSE perform steps (S) through (E2).

(S) Determine a primitive squarefree factor $\bar{s}(z_1, \dots, z_v, x)$ of \bar{f} by a squarefree decomposition algorithm such as Yun's algorithm [32] or Wang and Trager's algorithm [30].

(M) [Transform \bar{s} into a polynomial s monic in x :] $n \leftarrow \deg_x(\bar{s})$;

$$c(z_1, \dots, z_v) \leftarrow \text{lcf}_x(\bar{s});$$

$$s(z_1, \dots, z_v, x) \leftarrow c(z_1, \dots, z_v)^{n-1} \bar{s}\left(z_1, \dots, z_v, \frac{x}{c(z_1, \dots, z_v)}\right).$$

[Notice that s is monic in x , an irreducible factor of which can be back-transformed to an irreducible factor of \bar{s} (see step (E2)).]

(T) [Find good integral evaluation points w_1, \dots, w_v such that $s(w_1, \dots, w_v, x)$ is squarefree.]

FOR ALL integers w_i with $|w_i| \leq \lceil (2n-1)/2 \deg_{z_i}(s) \rceil$, $1 \leq i \leq v$, DO

Test whether $s(w_1, \dots, w_v, x)$ is squarefree. If so, exit loop.

$$f(y_1, \dots, y_v, x) \leftarrow s(y_1 + w_1, \dots, y_v + w_v, x).$$

(R) Call Algorithm 2 given below to find an irreducible factor $g(y_1, \dots, y_v, x)$ of $f(y_1, \dots, y_v, x)$.

(E) [Recover a possibly nonmonic factor $\bar{g}(z_1, \dots, z_v, x)$ of $\bar{f}(z_1, \dots, z_v, x)$.]

(E1) $g(z_1, \dots, z_v, x) \leftarrow g(z_1 - w_1, \dots, z_v - w_v, x)$.

(E2) $\bar{g}(z_1, \dots, z_v, x) \leftarrow \text{pp}_x(g(z_1, \dots, z_v, c(z_1, \dots, z_v)x))$. \square

We shall first prove the correctness of the above algorithm. Obviously, if $g(y_1, \dots, y_v, x)$ divides f then $g(z_1, \dots, z_v, x)$ divides $s(z_1, \dots, z_v, x)$. The proof for the correctness of the transformations in the steps (M) and (E2) is quite easy and can be found in Knuth [16, p. 438, Exercise 18]. We first must show that step (T) will yield good evaluation points.

LEMMA 1. Let $s(z_1, \dots, z_v, x) \in \mathbf{Z}[z_1, \dots, z_v, x]$ be monic of degree n in x and squarefree. Then there exist integers w_i with $|w_i| \leq \lceil (2n-1)/2 \deg_{z_i}(s) \rceil$, $1 \leq i \leq v$, such that $s(w_1, \dots, w_v, x)$ is squarefree in $\mathbf{Z}[x]$.

Proof. Let $d_i = \deg_{z_i}(s)$ for $1 \leq i \leq v$. Since s is squarefree, its discriminant

$$\Delta(z_1, \dots, z_v) = \text{res}_x\left(s, \frac{\partial s}{\partial x}\right) \neq 0$$

(cf. van der Waerden [28, p. 86]). Since Δ is the given resultant, it follows that $\deg_{z_i}(\Delta) \leq (2n-1)d_i$ for $1 \leq i \leq v$. If we write $\Delta(z_1, \dots, z_v)$ as a polynomial in the variables z_2, \dots, z_v with coefficients in $\mathbf{Z}[z_1]$, not all these coefficients can be zero. Let $u(z_1)$ be one particular nonvanishing coefficient. Since $\deg(u) \leq (2n-1)d_1$ there exists

an integer w_1 with $|w_1| \leq \lceil (2n-1)/2d_1 \rceil$ and $u(w_1) \neq 0$. Therefore $\Delta(w_1, z_2, \dots, z_v) \neq 0$ and the lemma now follows by induction on the number of variables. \square

We now briefly discuss that the above algorithm is of polynomial complexity in $\deg(\bar{f})$ and $\log(|\bar{f}|)$ provided that this is also true for Algorithm 2. To obtain a squarefree factor \bar{s} of \bar{f} , we can use any of squarefree decomposition algorithms referred to in step (S), all of which employ polynomial GCD computations. Furthermore, any of the available GCD algorithms such as the primitive remainder, subresultant or the modular algorithm (cf. Brown [1]), or the EZGCD algorithm by Moses and Yun [25], takes for a fixed number of variables polynomially many steps in the maximum degree of the input polynomial and the size of its coefficients. That this time bound extends to the squarefree factorization process is shown, e.g., in Yun [33]. Of course, $\deg(\bar{s}) \leq \deg(\bar{f})$ in step (S), and a bound for $|\bar{s}|$ can be determined by the following lemma.

LEMMA 2. Let $g_1, \dots, g_m \in \mathbf{C}[x_1, \dots, x_v]$, let $f = g_1 \cdots g_m$ and let $n_j = \deg_{x_j}(f)$, $n = \sum_{j=1}^v n_j$. Then

$$\prod_{i=1}^m |g_i| \leq 2^n |f| \prod_{j=1}^v \left(\frac{n_j + 1}{2}\right)^{1/2} \leq c^n |f|$$

with $c < \sqrt{6} \approx 2.44949$ (cf. Gel'fand [9, pp. 135-139]).

Therefore $|\bar{s}| \leq c^{(v+1)\deg(\bar{f})} |\bar{f}|$. That the steps (M) and (T) take polynomial-time is quite easily established. As a matter of fact, some of the GCD algorithms used for the squarefree decomposition of \bar{f} in step (S) already provide the points w_1, \dots, w_v of step (T) as a by-product. Step (M) produces a substantially, yet polynomially, larger output compared to its input \bar{s} . (For example

$$\deg(s) \leq n \deg(\bar{s}) \quad \text{and} \quad |s| \leq (\deg(\bar{s}) + 1)^{vn} |\bar{s}|^n;$$

cf. Lemma 7.) Step (T) again may produce a larger result, but $|f|$ is clearly polynomial in the size of s . (For example

$$|f| \leq v^{\deg(s)} \deg(s)^v \deg(s)^{2\deg(s)} |s|;$$

see also Lemma 1 and Lemma 4.)

We wish to remark that step (M) could be entirely avoided by modifying Algorithm 2. However, these modifications would complicate the complexity analysis and for the reasons discussed above we shall retain the monicity condition on f during Algorithm 2. The matter becomes more manageable if the coefficients are in a finite field. Some details to this case can be found in von zur Gathen and Kaltofen [8].

Step (E1) is the counterpart of the transformation of step (T). Step (E2) is similar to step (M), but also involves a content computation. Both steps can obviously be performed in time polynomial in $\deg(g)$ and $\log(|g|)$.

5. The main algorithm. In this section, we shall discuss an algorithm which computes an irreducible factor of a polynomial $f(y_1, \dots, y_v, x)$, monic in x with $f(0, \dots, 0, x)$ squarefree, in polynomial-time in $\deg(f)$ and $\log(|f|)$. We will also prove the proposed algorithm correct. The analysis of its complexity is deferred to the next section. The algorithm first computes a multivariate Taylor series approximation of a root of f for x . It then finds the minimal polynomial for this root by solving a linear system in the coefficients of this polynomial.

ALGORITHM 2.

[Input: $f(y_1, \dots, y_v, x) \in \mathbf{Z}[y_1, \dots, y_v, x]$ monic in x such that $f(0, \dots, 0, x)$ is squarefree. \mathbf{Z} can be an arbitrary UFD and \mathbf{Q} its field of quotients. Output: An irreducible factor $g(y_1, \dots, y_v, x) \in \mathbf{Z}[y_1, \dots, y_v, x]$ of f]

(F) [Factor $f(0, \dots, 0, x)$:] $n \leftarrow \deg_x(f)$.

Compute an irreducible factor $t(x)$ of $f(0, \dots, 0, x)$; $m \leftarrow \deg(t)$.

[Let β be a root of t . In the following, we will perform computations in $\mathbf{Q}(\beta)$, whose elements are represented as polynomials in $\mathbf{Q}[\beta]$ modulo t .]

(N) [Newton iteration. For purposes of later analysis and reference, we emulate the Newton iteration by a Hensel lifting algorithm. Let J be the ideal in $\mathbf{Q}(\beta)[y_1, \dots, y_v]$ generated by $\{y_1, \dots, y_v\}$. The goal is to construct

$$\alpha_j(y_1, \dots, y_v) = \sum_{i=0}^j \sum_{|k|=i} a_k(\beta) y_{\underline{k}}, \quad \text{where } a_k(\beta) \in \mathbf{Q}(\beta),$$

for $j = 1, 2, \dots$ such that

$$f(y_1, \dots, y_v, \alpha_j(y_1, \dots, y_v)) \equiv 0 \pmod{J^{j+1}},$$

i.e. no monomials in y_1, \dots, y_v with total degree less than $j+1$ occur on the left-hand side of the given equation.]

Rewrite $f(y_1, \dots, y_v, x) = \sum_{k \geq 0} f_k(x) y_{\underline{k}}$, where $f_k(x) \in \mathbf{Z}[x]$. [Notice that $f_0(x) = f(0, \dots, 0, x)$ and, since f is monic and $\deg_x(f) = n$, $\deg(f_k) < n$ for $|k| \geq 1$.]

[Set order for approximation:] $d \leftarrow \deg_{y_1, \dots, y_v}(f)$; $K \leftarrow d(2n-1)$.

[Initialize for Hensel lifting:]

$$a_0 \leftarrow \beta; \quad g_0(x) \leftarrow x - \beta; \quad h_0(x) \leftarrow f_0(x)/g_0(x) \in \mathbf{Q}(\beta)[x].$$

FOR ALL \underline{k} with $1 \leq |k| \leq K$ DO steps (N1) and (N2). [The \underline{k} must be generated in an order such that $|k|$ is nondecreasing. We will compute polynomials $g_k(x)$ and $h_k(x) \in \mathbf{Q}(\beta)[x]$, $k \geq 0$, satisfying

$$(1) \quad \left(\sum_{k \geq 0} g_k(x) y_{\underline{k}} \right) \left(\sum_{k \geq 0} h_k(x) y_{\underline{k}} \right) = \sum_{k \geq 0} f_k(x) y_{\underline{k}}. \quad]$$

$$(N1) \quad b_k(x) \leftarrow f_k(x) - \sum_{0 \leq s \leq k, 1 \leq |s| \leq |k|-1} g_s(x) h_{k-s}(x).$$

(N2) [Step (N1) and (1) lead to

$$(2) \quad g_0(x) h_k(x) + h_0(x) g_k(x) = b_k(x)$$

with $g_k(x), h_k(x) \in \mathbf{Q}(\beta)[x]$, $\deg(g_k) \leq \deg(g_0) - 1 = 0$, $\deg(h_k) \leq \deg(h_0) - 1 = n - 2$. In the Hensel lifting algorithm, (2) is accomplished by the extended Euclidean algorithm (cf. Knuth [16, p. 417, Exercise 3], but since $\deg(g_k) = 0$ we can use direct formulas:]

$$a_k \leftarrow g_k(x) \leftarrow \frac{b_k(\beta)}{f_0'(\beta)}; \quad h_k(x) \leftarrow \frac{b_k(x) - h_0(x) g_k(x)}{g_0(x)}.$$

[Assign approximate root:] $\alpha_j \leftarrow \sum_{0 \leq |k| \leq j} a_k y_{\underline{k}}$ for $0 \leq j \leq K$.

(L) [Find minimal polynomial for α_K :]

[Compute powers of α_K :]

FOR $i \leftarrow 0, \dots, n-1$ DO $\alpha_K^{(i)} \leftarrow \alpha_K^i \pmod{J^{K+1}}$.

FOR $I \leftarrow m, \dots, n-1$ DO

$L \leftarrow d(I+n-1)$.

With $\alpha_L^{(i)} \equiv \alpha_K^{(i)} \pmod{J^{L+1}}$ try to solve the equation

$$(3) \quad \alpha_L^{(I)} + \sum_{i=0}^{I-1} u_i(y_1, \dots, y_v) \alpha_L^{(i)} \equiv 0 \pmod{J^{L+1}}$$

for polynomials $u_i(y_1, \dots, y_v) \in \mathbf{Q}[y_1, \dots, y_v]$ such that $\deg_{y_1, \dots, y_v}(u_i) \leq d$. Let $u_i(y_1, \dots, y_v) = \sum_{0 \leq |s| \leq d} u_{i,s} y^s$ and let

$$\alpha_L^{(i)} = \sum_{0 \leq |k| \leq L} \left(\sum_{j=0}^{m-1} a_{k,j}^{(i)} \beta^j \right) y^k.$$

Then (3) leads to the linear system

$$(4) \quad a_{k,j}^{(I)} + \sum_{i=0}^{I-1} \sum_{0 \leq |s| \leq d} \alpha_{k-s,j}^{(i)} u_{i,s} = 0$$

for $0 \leq |k| \leq L, j = 0, \dots, m-1$ in the variables $u_{i,s}, i = 0, \dots, I-1, 0 \leq |s| \leq d$. [There are $I \binom{v+d}{d}$ unknowns in $m \binom{v+L}{L}$ linear equations. (Cf. Lemma 4.)] IF (4) has a solution (which, as we will prove below, is then integral and unique) THEN

$$g(y_1, \dots, y_v, x) \leftarrow x^I + \sum_{i=0}^{I-1} u_i(y_1, \dots, y_v) x^i$$

and EXIT. [We will also show that then g is an irreducible factor of f .]

[At this point, the above FOR loop has not produced a solution to (3). In this case, f is irreducible.] $g \leftarrow f$. \square

Notice that L , the order of the approximation needed, grows with I , the possible degree of the minimal polynomial. Hence we could improve our algorithm by increasing the order of the approximation within the loop on I in step L instead of computing the best approximation eventually needed a priori in step (N). Also, a complete factorization of f_0 may exclude certain degrees for g . For example, if f_0 factors into irreducibles of even degree, then g cannot be of odd degree. (Cf. Knuth [16, p. 434 and § 4.6.2, Exercise 26].)

We shall now prove the correctness of the above algorithm. We first show that step (N) computes a root $\alpha_K(y_1, \dots, y_v)$ of $f(y_1, \dots, y_v, x)$ modulo J^{K+1} . The polynomials $g_k(x)$ and $h_k(x) \in \mathbf{Q}(\beta)[x], k \geq 0$, must satisfy (1) and thus (2). We now note that $g_0(\beta) = 0$ and $h_0(\beta) = f'_0(\beta)$. The second equation follows from the fact that if $\beta, \beta_2, \dots, \beta_n$ are the roots of $f_0(x)$ then $h_0(x) = \prod_{i=2}^n (x - \beta_i)$ and hence $h_0(\beta) = \prod_{i=2}^n (\beta - \beta_i) = f'_0(\beta)$. Therefore the unique solution of (2) with $\deg(g_k) = 0$ is $a_k = b_k(\beta)/f'_0(\beta)$. If we now solve (3) for $h_k(x)$, we get

$$h_k(x) = \frac{b_k(x) - h_0(x)g_k(x)}{g_0(x)}$$

which is a polynomial in x since $b_k(\beta) - h_0(\beta)a_k = 0$, and is of degree at most $n-2$. As we will see in § 6, the solution for (3) with $\deg(g_k) < \deg(g_0)$ and $\deg(h_k) < \deg(h_0)$ is uniquely determined by a linear system in n unknowns, whose coefficient matrix is the Sylvester matrix of $g_0(x)$ and $h_0(x)$, the determinant of which in our case happens to be equal to $f'_0(\beta)$.

We now conclude that

$$f(y_1, \dots, y_v, \alpha_K(y_1, \dots, y_v)) \equiv 0 \pmod{J^{K+1}}$$

because

$$\left(x - \sum_{0 \leq |k| \leq K} a_k y^k \right) \left(\sum_{0 \leq |k| \leq K} h_k(x) y^k \right) \equiv f(y_1, \dots, y_v, x) \pmod{J^{K+1}}.$$

The polynomial $g(y_1, \dots, y_v, x)$ is constructed in step (L) such that

$$g(y_1, \dots, y_v, \alpha_L(y_1, \dots, y_v)) \equiv 0 \pmod{J^{L+1}}.$$

We will now prove that g must divide f . Our argument will show that if g does not divide f , then (3) has a solution for $I < \deg(g)$. One main condition for this to be true is that our approximation is at least of order L . First, we must prove a simple lemma.

LEMMA 3. *Let $g(y_1, \dots, y_v, x)$ divide $f(y_1, \dots, y_v, x)$ in $\mathbf{Z}[y_1, \dots, y_v, x]$ and assume that $g(0, \dots, 0, \beta) = 0$ in $\mathbf{Q}(\beta)$. Then*

$$g(y_1, \dots, y_v, \alpha_j(y_1, \dots, y_v)) \equiv 0 \pmod{J^{j+1}}$$

for all $j \geq 1$ with $\alpha_j(y_1, \dots, y_v)$ as computed in step (N).

Proof. The reason is simply that since $x - \alpha_j(y_1, \dots, y_v)$ divides $f(y_1, \dots, y_v, x) \pmod{J^{j+1}}$ and β is a root of single multiplicity, $x - \alpha_j(y_1, \dots, y_v)$ must also divide $g(y_1, \dots, y_v, x) \pmod{J^{j+1}}$. This argument can be made formal but we shall provide a more indirect proof. Let p be the first index such that

$$g(y_1, \dots, y_v, \alpha_p(y_1, \dots, y_v)) \not\equiv 0 \pmod{J^{p+1}}.$$

Because p is the first index

$$g(y_1, \dots, y_v, \alpha_p(y_1, \dots, y_v)) \equiv \sum_{|k|=p} \gamma_k y^k \pmod{J^{p+1}}$$

with at least one $\gamma_k \neq 0$. Let h be the cofactor of g , i.e. $f = gh$. Since β is a single root, $r = h(0, \dots, 0, \beta) \neq 0$. Therefore

$$g(y_1, \dots, y_v, \alpha_p(y_1, \dots, y_v))h(y_1, \dots, y_v, \alpha_p(y_1, \dots, y_v)) \equiv \sum_{|k|=p} \gamma_k r y^k \not\equiv 0 \pmod{J^{p+1}}$$

in contradiction to $\alpha_p(y_1, \dots, y_v)$ being the p th approximation of a root of f . \square

THEOREM 1. *The first solution of (3) in step (L), as I increases, determines a proper factor g of f in $\mathbf{Z}[y_1, \dots, y_v, x]$. This factor is also irreducible.*

Proof. We show that g must divide f provided its coefficients satisfy (4). The irreducibility of g then follows immediately from the fact that the minimal polynomial for the root of $f(y_1, \dots, y_v, x)$ corresponding to α_L also provides a solution to (3) and hence (4). Let

$$D(y_1, \dots, y_v, x) = \text{GCD}(f(y_1, \dots, y_v, x), g(y_1, \dots, y_v, x))$$

and let $I = \deg_x(g)$, $j = \deg_x(D)$. We shall prove that the condition $j < I$ is impossible. Assume that this condition is satisfied, i.e. $0 \leq j < I$. Let $f = x^n + t_{n-1}x^{n-1} + \dots + t_0$ and $g = x^I + u_{I-1}x^{I-1} + \dots + u_0$ with $t_i, u_m \in \mathbf{Z}[y_1, \dots, y_v]$. Using the extended Euclidean algorithm (cf. Knuth [16, p. 417, Exercise 3]) we establish the existence of polynomials $U_j, V_j \in \mathbf{Q}(y_1, \dots, y_v)[x]$, $\deg_x(U_j) < I - j$ and $\deg_x(V_j) < n - j$, such that

$$(A) \quad U_j f + V_j g = D.$$

It is easy to show that under the given degree constraints these polynomials are uniquely determined. Therefore we must have a nonsingular coefficient matrix for the linear system derived from (A) for the coefficients of $x^j, \dots, x^{I+n-j-1}$ with the unknowns being the coefficients of U_j, V_j of x . By s_j we denote the determinant of this coefficient matrix namely

$$(B) \quad s_j = \det \begin{bmatrix} 1 & t_{n-1} & \cdots & & & t_{2j-I+1} \\ & & & 1 & t_{n-1} & \cdots & t_j \\ 1 & u_{I-1} & \cdots & u_{j+1} & \cdots & & u_{2j-n+1} \\ & & & & 1 & u_{I-1} & \cdots & u_j \end{bmatrix} \neq 0.$$

(In fact, s_j is the leading coefficient of the j th sub-resultant of f and g as polynomials in x ; cf. Brown and Traub [2].) Cramer's rule implies that $s_j U_j, s_j V_j \in \mathbf{Z}[y_1, \dots, y_v, x]$. Moreover,

$$f(y_1, \dots, y_v, \alpha_L) \equiv g(y_1, \dots, y_v, \alpha_L) \equiv 0 \pmod{J^{L+1}}$$

and hence

$$s_j(y_1, \dots, y_v) D(y_1, \dots, y_v, \alpha_L) \equiv 0 \pmod{J^{L+1}}.$$

However, from Lemma 3 and the fact that g is the polynomial of smallest degree solving (3) we conclude that $D(0, \dots, 0, \beta) \neq 0$, which implies with the previous congruence that

$$(C) \quad s_j(y_1, \dots, y_v) \equiv 0 \pmod{J^{L+1}}.$$

On the other hand, using (B) we can bound the degree of s_j by

$$\deg_{y_1, \dots, y_v}(s_j) \leq (I + n - 2j - 1)d \leq (I + n - 1)d = L$$

which together with (C) implies that $s_j = 0$, in contradiction to (B). \square

This concludes the correctness proof for Algorithm 2. In the case that $v = 1$ the bound K of step (N) and L of step (L) can be improved to $\lceil d(2n - 1)/m \rceil$ (cf. Kaltofen [13, Thm. 4.1]). However, this improvement seems not to carry over for $v \geq 2$, since $\mathbf{Q}[y_1, \dots, y_v]$ is not a Euclidean domain.

6. Complexity analysis of the reduction algorithm. The goal of this section is to prove that Algorithm 2 takes, for a fixed number of variables v , polynomially many steps in $\deg(\bar{f}) \log(|\bar{f}|)$, provided that we can factor f_0 in time polynomial in $\deg(f_0) \log(|f_0|)$.

Step (F). As A. Lenstra, H. Lenstra and L. Lovász have shown, $t(x)$ can be computed in at most $O(\deg(f_0)^{12} + \deg(f_0)^9 (\log|f_0|_2)^3)$ steps [22]. This complexity bound can be slightly improved using the results of Kaltofen [15].

Step (N). We first count the number of additions, subtractions and multiplications over $\mathbf{Q}(\beta)$ (which we shall call *ASM ops*) needed for this step. Then we bound the absolute value of all elements of $\mathbf{Q}(\beta)$ which appear as intermediate results. Finally, we bound the size of all computed rational numerators and denominators, and then we count the number of rational operations. The most difficult task will be to compute size bounds.

We can ignore the time it takes to retrieve the polynomials $f_k(x)$ as well as the execution time for the initializations of step (N). In order to count the number of times steps (N1) and (N2) are performed, we need the following lemma.

LEMMA 4. *There exist*

$$\binom{v+j-1}{v-1} \leq (j+1)^{v-1}$$

distinct v -dimensional integral vectors k with $|k| = j$. The number of vectors with $|k| \leq j$ is

$$\binom{v+j}{v} \leq (j+1)^v.$$

Therefore, steps (N1) and (N2) are executed at most $(K+1)^v$ times. Step (N1) requires at most $O(K^v n)$ *ASM ops* in $\mathbf{Q}(\beta)$. Clearly this bound also dominates the complexity of step (N2). Hence α_K can be calculated in $O(K^{2v} n)$ *ASM ops*.

We now proceed to compute an upper bound B_1 for all absolute values of the coefficients of α_k in $\mathbf{Q}(\beta)$.

LEMMA 5. Let $f(x) \in \mathbf{Z}[x]$ be monic, squarefree, of degree n and let $g(x), h(x) \in \mathbf{C}[x]$ be monic such that $f(x) = g(x)h(x)$.

a) Then both $|g|, |h| \leq 2^n |f|_2 \leq \sqrt{n+1} 2^n |f|$ and if β is any root of f , $|\beta| < 2|f|$.

b) If M is any $(n-1)$ by $(n-1)$ submatrix of the Sylvester matrix of g and h , then

$$|\det(M)| < T(f) = (n2^n |f|)^{n-1}.$$

c) The resultant of g and h is bounded by $1/S(f) < |\text{res}(g, h)| < 2T(f)$ with

$$S(f) = (4|f|)^{(n-1)(n-2)/2}.$$

Proof. a) The bound for $|g|$ and $|h|$ is the Landau-Mignotte bound translated to maximum norms [24]. Assume $f(x) = a_n x^n + \dots + a_0$ and let $\beta \in \mathbf{C}$ with $|\beta| \geq 2|f|$. Then

$$|a_{n-1}\beta^{n-1} + \dots + a_0| \leq |f| \frac{|\beta|^n - 1}{|\beta| - 1} < |\beta|^n \leq a_n |\beta|^n$$

because $|f| \geq 1$, therefore $f(\beta) \neq 0$. Notice that for this part the monicity of f is not required.

b) By part a), we know that each entry in the Sylvester matrix of g and h is bounded by $\sqrt{n+1} 2^n |f|$. Hadamard's determinant inequality (cf. Knuth [16, § 4.6.1, Exercise 15]) then gives the bound.

c) Let $g(x) = (x - \beta_1) \dots (x - \beta_k)$ and $h(x) = (x - \beta_{k+1}) \dots (x - \beta_n)$. Then

$$\text{res}(g, h) = \prod_{i=1, \dots, k; j=k+1, \dots, n} (\beta_i - \beta_j)$$

and the discriminant of f , $\Delta = \prod_{i \neq j} (\beta_i - \beta_j)$, is an integer not equal 0 (cf. van der Waerden [28, pp. 87-89]). From a) we conclude that $|\beta_i - \beta_j| < 4|f|$ for $1 \leq i < j \leq n$. Therefore

$$\begin{aligned} 1 \leq \sqrt{|\Delta|} &= \left(\prod_{1 \leq i < j \leq k} |\beta_i - \beta_j| \right) |\text{res}(g, h)| \left(\prod_{k+1 \leq i < j \leq n} |\beta_i - \beta_j| \right) \\ &< |\text{res}(g, h)| (4|f|)^{(n-1)(n-2)/2} \end{aligned}$$

because $k(k-1) + (n-k)(n-k-1) \leq (n-1)(n-2)$ for $1 \leq k \leq n-1$. The upper bound follows from b) and the fact that g and h are monic. \square

The following lemma estimates the size of a general version of the Catalan numbers.

LEMMA 6. Let $d_k = 1$ for all v -dimensional vectors k with $|k| = 1$ and let

$$d_k = \sum_{0 \leq s \leq k, 1 \leq |s| \leq |k|-1} d_s d_{k-s} \quad \text{for } |k| \geq 2.$$

Then

$$d_k = \frac{1}{|k|} \binom{2|k|-2}{|k|-1} \frac{|k|!}{k_1! \dots k_v!} < (4v)^{|k|}.$$

Proof. Let $G(y_1, \dots, y_v) = \sum_{|k| \geq 1} d_k y^k$ be the generating function for d_k . Then

$$G(y_1, \dots, y_v)^2 = G(y_1, \dots, y_v) - (y_1 + \dots + y_v)$$

and thus

$$G(y_1, \dots, y_v) = \frac{1}{2} \left(1 - \sqrt{1 - 4(y_1 + \dots + y_v)} \right) = \sum_{i=1}^{\infty} \frac{1}{i} \binom{2i-2}{i-1} (y_1 + \dots + y_v)^i$$

which yields our formula. Since $|k|!/(k_1! \cdots k_v!)$ is a multinomial coefficient, it is less than $v^{|k|}$. Similarly the given binomial coefficient is less than $2^{2|k|}$. \square

We are now in the position to formulate and prove the main theorem on the coefficient growth for the Hensel lifting algorithm. This theorem also resolves the growth problem left open by Kung and Traub [18] who considered the Newton iteration for the case that $v = 1$. We actually use a slightly more general approach which we will also use in § 7.

THEOREM 2. *Let $f(y_1, \dots, y_v, x) \in \mathbf{Z}[y_1, \dots, y_v, x]$ be monic of degree n in x , such that $f_0(x) = f(0, \dots, 0, x)$ is squarefree. Let β be an algebraic integer generating a subfield of the splitting field for f_0 . By $\mathbf{Z}[\beta]$ we denote the ring generated by \mathbf{Z} and $\{\beta\}$ whose elements are polynomials in β with integral coefficients of degree $[\mathbf{Q}(\beta) : \mathbf{Q}] - 1$. Let $g_0(x)h_0(x) = f_0(x)$ be a nontrivial factorization of f_0 in $(\mathbf{Z}[\beta])[x]$ with g_0 and h_0 both monic in x . Then there exist unique polynomials $g_k(x), h_k(x) \in \mathbf{Q}(\beta)[x]$ with $|k| \geq 1$ and $\deg(g_k) < \deg(g_0), \deg(h_k) < \deg(h_0)$ such that*

$$f(y_1, \dots, y_v, x) = \left(\sum_{k \geq 0} g_k(x) y^k \right) \left(\sum_{k \geq 0} h_k(x) y^k \right).$$

Furthermore, let

$$\frac{1}{\text{res}(g_0, h_0)} = \frac{1}{R} r(\beta) \quad \text{with } R \in \mathbf{Z}, r(\beta) \in \mathbf{Z}[\beta],$$

and let $S(f_0)$ and $T(f_0)$ be as defined in Lemma 5. Finally, let $N(f) = \max(n^2, n|f|)$, and let d_k be as defined in Lemma 6. Then for all k with $|k| \geq 1$

$$R^{2|k|-1} g_k(x), R^{2|k|-1} h_k(x) \in (\mathbf{Z}[\beta])[x]$$

and, independently of which root β of f_0 we choose,

$$|g_k|, |h_k| \leq d_k (N(f) S(f_0) T(f_0))^{2|k|-1}.$$

Proof. The existence and uniqueness of g_k and h_k follows from the fact that (2) has a unique solution with the given degree constraints, b_k being computed as in step (N1). Now let $C_k = \max(|g_k|, |h_k|, |f|)$ and let $D_k = |b_k|$. Since $\deg(g_s) < \deg(g_0)$ and $\deg(h_{k-s}) < \deg(h_0)$, we conclude that

$$|g_s h_{k-s}| \leq (n-1) |g_s| |h_{k-s}| \leq (n-1) C_s C_{k-s}.$$

By definition $C_s \geq |f|$ and thus we obtain from (N1)

$$(A) \quad D_k \leq n \sum_{0 \leq s \leq k, 1 \leq |s| \leq |k|-1} C_s C_{k-s}.$$

Let \vec{p} denote the coefficient vector (p_m, \dots, p_0) of the polynomial $p(x) = p_m x^m + \dots + p_0$. Now if we solve (2) by undetermined coefficients for g_k and h_k we encounter the Sylvester matrix of g_0 and h_0 , $\text{Syl}(g_0, h_0)$, as the coefficient matrix, namely

$$(B) \quad (\vec{h}_k, \vec{g}_k) \text{Syl}(g_0, h_0) = \vec{b}_k,$$

where (\vec{h}_k, \vec{g}_k) denotes the vector obtained by concatenation of the vectors \vec{h}_0 and \vec{g}_k . Using Cramer's rule for (B) and the fact that

$$\frac{1}{|\det(\text{Syl}(g_0, h_0))|} = \frac{1}{|\text{res}(g_0, h_0)|} < S(f_0)$$

(by Lemma 5c), we get the estimate

$$(C) \quad C_k \leq \max(|f|, nD_k S(f_0) T(f_0))$$

(also using Lemma 5b). We now prove our claims by induction on $|k|$.

Case $|k| = 1$. Since $b_k = f_k \in \mathbf{Z}[x]$, Cramer's rule applied to (B) yields $Rg_k, Rh_k \in (\mathbf{Z}[\beta])[x]$. (Notice that β is an algebraic integer.) Also $D_k \leq |f|$ and hence by (C)

$$C_k \leq \max(|f|, n|f|S(f_0) T(f_0)) \leq d_k N(f) S(f_0) T(f_0).$$

Case $|k| > 1$. By hypothesis and from (N1) we obtain $R^{2|k|-2} b_k \in (\mathbf{Z}[\beta])[x]$. Cramer's rule applied to (B) then yields $R^{2|k|-1} g_k, R^{2|k|-1} h_k \in (\mathbf{Z}[\beta])[x]$. From (A) and the hypothesis we also get

$$\begin{aligned} D_k &\leq n \sum_{0 \leq s \leq k, 1 \leq |s| \leq |k|-1} C_s C_{k-s} \\ &\leq n(N(f)S(f_0)T(f_0))^{2|k|-2} \left(\sum_{0 \leq s \leq k, 1 \leq |s| \leq |k|-1} d_s d_{k-s} \right) \\ &= n(N(f)S(f_0)T(f_0))^{2|k|-2} d_k. \end{aligned}$$

By (C) we finally obtain

$$\begin{aligned} C_k &\leq \max(|f|, nD_k S(f_0) T(f_0)) \\ &\leq d_k \frac{n^2}{N(f)} (N(f)S(f_0)T(f_0))^{2|k|-1} \\ &\leq d_k (N(f)S(f_0)T(f_0))^{2|k|-1}. \end{aligned} \quad \square$$

Since the polynomials g_k and h_k are unique, we can conclude from Theorem 2 that

$$|a_k| \leq d_k (N(f)S(f_0)T(f_0))^{2|k|-1} \quad \text{for } 1 \leq |k| \leq K.$$

From Lemmas 5 and 6 we obtain

$$(5) \quad \begin{aligned} |\alpha_K| &\leq B_1(f) = (4v)^K (n^2|f|(4|f|)^{n^2/2} 2^{n^2} (n|f|)^n)^{2K-1} \\ &< (4v)^K (2n|f|)^{2Kn^2}, \end{aligned}$$

assuming that $n \geq 4$. Obviously, $\log(B_1(f))$ is polynomial in $\deg(f)$ and $\log(|f|)$.

We now demonstrate for the polynomials $g_0 = x - \beta$ and h_0 as computed in step (N), that

$$\frac{R}{\text{res}(g_0, h_0)} \in \mathbf{Z}[\beta], \quad \text{with } R = \text{res}(t(x), f'_0(x)),$$

where t is the minimal polynomial of β . Let β_2, \dots, β_n be the roots of h_0 . Then

$$\text{res}(g_0, h_0) = \prod_{i=2}^n (\beta - \beta_i) = f'_0(\beta).$$

There exist polynomials $A(x)$ and $B(x) \in \mathbf{Z}[x]$ such that $At + Bf'_0 = R$. Thus $R/f'_0(\beta) = B(\beta) \in \mathbf{Z}[\beta]$, which we wanted to show. Now let $m = \deg(t)$. By Lemma 5a $|t| \leq \sqrt{n+1} 2^n |f_0|$, and using Hadamard's determinant inequality for the resultant $\text{res}(t, f'_0)$ we obtain

$$(6) \quad \begin{aligned} |R| &\leq (\sqrt{(m+1)(n+1)} 2^n |f_0|)^{n-1} (\sqrt{nn} |f_0|)^m \\ &< ((n+1) 2^n |f_0|)^{m+n} < (2n|f|)^{n^3/2}, \end{aligned}$$

for $n \geq 4$. Again, we note that $\log(|R|)$ is bounded by a polynomial in $\deg(f)$ and $\log(|f|)$. From Theorem 2 we can also conclude that

$$(7) \quad R^{2|k|-1} a_k \in \mathbf{Z}[\beta] \quad \text{for } 1 \leq |k| \leq K.$$

We now extend our estimates to the powers of $\alpha_K \bmod J^{K+1}$ as well as count the ASM ops needed to compute the powers of α_K .

LEMMA 7. Let $\alpha_K^{(i)} = \sum_{0 \leq |k| \leq K} a_k^{(i)} y^k$ for $2 \leq i \leq n-1$, then

$$|a_k^{(i)}| \leq (K+1)^{v(i-1)} B_1(f)^i \text{ and } R^{2|k|-1} a_k^{(i)} \in \mathbf{Z}[\beta],$$

with R as defined above. All $\alpha_K^{(i)}$, $2 \leq i \leq n-1$, can be computed in $O(K^{2v}n)$ ASM ops.

Proof. It is easy to show that

$$a_k^{(i+1)} = \sum_{0 \leq s \leq k} a_s^{(i)} a_{k-s}, \quad 0 \leq |k| \leq K, \quad i \geq 1,$$

where there are, by Lemma 4, at most $(|k|+1)^v \leq (K+1)^v$ terms under the right-hand sum. The lemma now follows by induction on i . \square

Therefore we get from (5) for all $0 \leq i \leq n-1$ and for $n \geq 4$

$$(8) \quad |\alpha_K^{(i)}| \leq B_2(f) = ((K+1)^v B_1(f))^{n-1} < 2^{3vnK} (2n|f|)^{2K(n^3-n^2)}.$$

Lemma 7 also establishes that the common denominator of any rational coefficient computed throughout step (N) is R^{2K-1} . We are now in the position of estimating the size of any numerator of the rational coefficients of $\alpha_K^{(i)}$, $1 \leq i \leq n-1$. To do this, we shall state a useful lemma.

LEMMA 8. Let β be any root of $t(x) \in \mathbf{Z}[x]$, monic, squarefree of degree m . Let A be a real upper bound for the absolute value of any conjugate β_j , $1 \leq j \leq m$, of β . Assume that for all $1 \leq j \leq m$

$$\left| \sum_{i=0}^{m-1} c_i \beta_j^i \right| \leq C \quad \text{with } c_i \in \mathbf{Z}.$$

Furthermore, let D be the absolute value of the discriminant of t . Then

$$|c_i| \leq \frac{Cm! A^{m(m-1)/2}}{\sqrt{D}}, \quad 0 \leq i < m$$

(cf. Weinberger and Rothschild [31, Lemma 8.3]).

In our case, we can choose $A = 2|f_0|$, by Lemma 5a), $C = B_2(f)R^{2K-1}$, and $D \geq 1$. Therefore, if we bring all rationals computed in step (N) to the common denominator R^{2K-1} , we have shown that the absolute values of the numerators are bounded by

$$(9) \quad B_3(f, m) = R^{2K-1} B_2(f) m! (2|f_0|)^{m(m-1)/2} < 2^{3vnK} (2n|f|)^{3Kn^3}, \quad (9)$$

using (6), (8) and $n \geq 4$. Though this bound is quite large, it is of length polynomial in $\deg(f)$ and $\log(|f|)$. This bound also implies, that all ASM ops are computable in time polynomial in $\deg(f)$ and $\log(|f|)$. Addition and subtraction in $\mathbf{Q}(\beta)$ means adding or subtracting the numerators of polynomials in $\mathbf{Q}[\beta]$ of degree $m-1$, after eventually multiplying them with a power of R to produce a common denominator. Multiplication in $\mathbf{Q}(\beta)$ is multiplication of $m-1$ degree polynomials in $\mathbf{Q}[\beta]$ followed by a remainder computation w.r.t. $t(\beta)$. Again a common denominator can be extracted a priori. Any ASM op takes at most $O(m^2)$ integral operations.

Step (L). By Lemma 4, it follows that (4) consists of

$$p = m \binom{v+K}{K} \leq m(K+1)^v$$

equations in

$$q = I \binom{v+d}{d} \cong (n-1)(d+1)^v$$

unknowns. Applying Gaussian elimination to (4) takes $O(pq^2)$ rational operations. It is easy to show that this is the dominant operation count, which, expressed in input terms, is

$$(10) \quad O(mn^{v+3}d^{3v}).$$

From the previous analysis, we know that all $a_{kj}^{(i)}$ can be brought to the common denominator R^{2K-1} and their numerators, $\text{num}(a_{kj}^{(i)})$, then satisfy $|\text{num}(a_{kj}^{(i)})| \cong B_3(f, m)$. As can be shown with little effort, all intermediate rationals computed during the Gaussian elimination process are fractions of subdeterminants of the coefficient matrix for (4) extended by the vector of constants (cf. Gantmacher [6, Chap. 2]). It is not necessary to calculate the GCD of the numerator and denominator of a newly obtained rational since, as can also be shown, the denominator of the row used for the elimination in subsequent rows divides the numerators and denominators in these rows after the elimination step. Thus Hadamard’s determinant inequality produces a bound for the size of any intermediately computed integer which is polynomial in $\deg(f) \log(|f|)$. E.g., one such bound is

$$B_4(f, m) = (\sqrt{q}B_3(f, m))^q$$

whose logarithm is by (8) of order

$$(11) \quad \log(B_4(f, m)) = O(d^{v+1}vn^5 \log(n|f|)).$$

Hence, step (L) also takes at most polynomial-time in $\deg(f)$ and $\log(|f|)$. Notice that (10) and (11) give a very crude bound for the complexity of the steps (N) and (L). Since we know that any solution of (4) must be integral of quite a small size, due to Lemma 2, a Chinese remaindering algorithm could be used to solve (4) (cf. McClellan [23]) and we believe that this approach will be much more efficient, in practice.

7. Multivariate irreducibility testing. As we have seen in § 5, in order to establish the irreducibility of the polynomial f by Algorithm 2 we need to factor f_0 . Reducibility of f_0 does, of course, not imply reducibility of f . The following theorem partially fills this gap by constructing from a polynomial $f(y_1, \dots, y_v, x)$, monic in x with $f(0, \dots, 0, x)$ squarefree, a polynomial $g(y_1, x)$ in time polynomial in $\deg(f)$ and $\log(|f|)$, such that g is irreducible if and only if f is irreducible. Unfortunately, our approach does not allow us to eliminate y_1 . We could include this as an open problem, but in view of the polynomial-time algorithm for bivariate factorization a solution appears not to be so significant.

LEMMA 9. *Let $t(y_1, \dots, y_v) \in \mathbf{Z}[y_1, \dots, y_v]$ be a nonzero polynomial. Then $t(y_1, cy_1, y_3, \dots, y_v) \neq 0$ for an integer c with $|c| \cong 2|t|$.*

Proof. Let $ay_1^{e_1} \dots y_v^{e_v}$ be a monomial in t with $a \neq 0$. Then $t(y_1, cy_1, \dots, y_v)$ contains the monomial $b(c)y_1^{e_1+e_2}y_3^{e_3} \dots y_v^{e_v}$ where $b(c)$ is an integral polynomial in c with degree at most $e_1 + e_2$. Since $b(c) = \dots + ac^{e_2} + \dots$ it cannot, as a polynomial, be identical to 0. From Lemma 5a and the fact that $|b| \cong |t|$ we conclude that $b(c) \neq 0$ for any integer of the stated size. \square

THEOREM 3. *Let $f(y_1, \dots, y_v, x) \in \mathbf{Z}[y_1, \dots, y_v, x]$ be monic of degree n in x such that $f_0 = f(0, \dots, 0, x)$ is squarefree. Let $T(f_0)$ be as in Lemma 5b, and let $N(f)$ be as in Theorem 2. Furthermore, assume that $f(y_1, \dots, y_v, x)$ is irreducible. Finally let*

$d = \deg_{y_1, \dots, y_v}(f)$. Then for any integer c with

$$|c| \geq B_5(f) = 2(4v)^{2d} (2N(f)T(f_0)^2)^{4d-1}$$

$f(y_1, cy_1, y_3, \dots, y_v, x)$ is irreducible in $\mathbf{Z}[y_1, y_3, \dots, y_v, x]$.

Proof. Let $\mathbf{Q}[[y_1, \dots, y_v]]$ denote the domain of formal power series in y_1, \dots, y_v over \mathbf{Q} , and let

$$g_c(y_1, y_3, \dots, x) = f(y_1, cy_1, y_3, \dots, x).$$

Then each factor of $g_c(y_1, y_3, \dots, x) \in \mathbf{Q}[[y_1, y_3, \dots, y_v]][x]$ corresponds to a factor of $f(y_1, y_2, \dots, x) \in \mathbf{Q}[[y_1, y_2, \dots, y_v]][x]$ with y_2 replaced by cy_1 . For, if a factor of g_c were not obtainable from a factor of f , we could present two different factorizations of g_c which, when evaluated at $y_1 = y_3 = \dots = y_v = 0$, would result in one and the same factorization of $g_c(0, \dots, 0, x) \in \mathbf{Q}[x]$. But this is impossible due to the uniqueness of the Hensel lifting procedure as proven in Theorem 2.² We will show that for an integer c of the stated size no factor derived from f in such a way can be an integral polynomial dividing g_c . Our plan is the following: We first show that any candidate factor $h(y_1, y_2, \dots, x)$ of $f(y_1, y_2, \dots, x) \in \mathbf{Q}[[y_1, \dots, y_v]][x]$ contains at least one monomial $b_{p,m} y_2^p x^m$ with $b_{p,m} \neq 0$ and $d < |p| \leq 2d$. From it we get a polynomial coefficient of x^m in h whose total degree in y_1, \dots, y_v equals $|p|$. By choosing c sufficiently large (cf. Lemma 9) we will be able to preserve this coefficient throughout $h_c = h(y_1, cy_1, \dots, x)$. Hence such an h_c contains a monomial in y_1, y_3, \dots, y_v of total degree $|p| > d$. Therefore h_c cannot be a polynomial dividing g_c for otherwise its total degree in y_1, y_3, \dots, y_v could not be larger than d . Let

$$h(y_1, \dots, y_v, x) = \sum_{i=0}^l \sum_{k \equiv 0} b_{k,i} y_2^k x^i$$

be a factor of $f(y_1, y_2, \dots, x)$ in $\mathbf{Q}[[y_1, \dots, y_v]][x]$ and let

$$\bar{h}(y_1, \dots, y_v, x) = \sum_{i=0}^{n-1} \sum_{k \equiv 0} \bar{b}_{k,i} y_2^k x^i$$

be its cofactor, i.e. $f = h\bar{h}$. We first can assume that

$$h(0, \dots, 0, x) = \sum_{i=0}^l b_{0,i} x^i \in \mathbf{Z}[x].$$

Otherwise $h(y_1, cy_1, y_3, \dots, x)$ could not be an integral polynomial for any choice of c .

Now there must exist at least on $b_{k,i}$ or $\bar{b}_{k,i}$ with

$$d < |k| \leq 2d \quad \text{and} \quad (b_{k,i} \neq 0 \text{ or } \bar{b}_{k,i} \neq 0).$$

To see this, assume the contrary. Then

$$\left(\sum_{i=0}^l \sum_{0 \leq |k| \leq d} b_{k,i} y_2^k x^i \right) \left(\sum_{i=0}^{n-1} \sum_{0 \leq |k| \leq d} \bar{b}_{k,i} y_2^k x^i \right) = f(y_1, \dots, y_v, x)$$

since no monomial $ay^k x^i$, a a nonzero rational, with $d < |k| \leq 2d$ in the left product could be canceled by higher terms in the product of the complete expansion of h and \bar{h} . Notice that f does not contain a monomial in y_2 of degree larger than d . But this contradicts the fact that f is irreducible. Without loss of generality we now can assume

² I owe this argument to Prof. Hendrik W. Lenstra, Jr.

the existence of a vector \underline{p} and an integer m such that

$$b_{\underline{p},m} \neq 0 \quad \text{with } d < |\underline{p}| \leq 2d \text{ and } 0 \leq m \leq l.$$

Let us consider the coefficient of x^m in h whose total degree in y_1, \dots, y_v is $|\underline{p}|$. Set

$$t_{\underline{p},m}(y_1, \dots, y_v) = \sum_{|j|=|\underline{p}|} b_{j,m} y^j$$

which is a polynomial in $\mathbf{Q}[y_1, \dots, y_v]$ not identical to 0.

We now apply Theorem 2 with $\beta = 1$, $g_0(x) = h(0, \dots, 0, x) \in \mathbf{Z}[x]$ and $h_0(x) = \bar{h}(0, \dots, 0, x) \in \mathbf{Z}[x]$. First notice that, since f_0 is squarefree, $0 \neq R = \text{res}(g_0, h_0) \in \mathbf{Z}$ and hence $1/|R| \leq 1$ meaning that we can set $S(f_0) = 1$. Secondly,

$$|b_{j,m}| \leq |g_j| \leq (4v)^{|j|} (N(f)T(f_0))^{2|j|-1} \leq (4v)^{2d} (N(f)T(f_0))^{4d-1}$$

because of Lemma 6 and $|j| = |\underline{p}| \leq 2d$. Finally,

$$R^{2|j|-1} b_{j,m} \in \mathbf{Z} \quad \text{and} \quad R^{2|j|-1} \leq R^{4d-1} < (2T(f_0))^{4d-1},$$

the last inequality by Lemma 5c. In summary,

$$0 \neq R^{4d-1} t_{\underline{p},m}(y_1, \dots, y_v) \in \mathbf{Z}[y_1, \dots, y_v]$$

and

$$|R^{4d-1} t_{\underline{p},m}| < (4v)^{2d} (2N(f)T(f_0)^2)^{4d-1} = \frac{1}{2} B_5(f).$$

From Lemma 9 we now conclude that for any integer $c \geq B_5(f)$

$$t_{\underline{p},m}(y_1, cy_1, y_3, \dots, y_v) \neq 0.$$

Therefore $h(y_1, cy_1, y_3, \dots, x)$ contains a nonzero monomial in y_1, y_3, \dots, y_v of total degree larger than d and cannot be a polynomial factor of $f(y_1, cy_1, y_3, \dots, x)$, as argued above. Our given bound then obviously works for any factor candidate h . \square

Our irreducibility test can now be constructed easily by induction. We compute the integers c_1, \dots, c_{v-1} such that for the sequence of polynomials $f_1 = f$,

$$f_2(y_1, y_3, \dots, x) = f_1(y_1, c_1 y_1, y_3, \dots, x),$$

$$f_3(y_1, y_4, \dots, x) = f_2(y_1, c_2 y_1, y_2, \dots, x), \dots,$$

$$f_v(y_1, x) = f_{v-1}(y_1, c_{v-1} y_1, x), \quad g = f_v,$$

we have $c_i \geq B_5(f_i)$ for all $1 \leq i \leq v-1$. Since v is assumed to be fixed and since $B_5(f_i)$ is of size polynomial in $\text{deg}(f_i)$ and $\log(|f_i|)$, g can be constructed in time polynomial in $\text{deg}(f)$ and $\log(|f|)$. By Theorem 3, g is irreducible if f is irreducible. On the other hand, if $f = h_1 h_2$ then

$$g(y_1, x) = h_1(y_1, c_1 y_1, \dots, c_{v-1} y_1, x) h_2(y_1, c_1 y_1, \dots, c_{v-1} y_1, x).$$

One can prove Theorem 3 for the more general substitution $y_2 = cy_1^s$, s being an arbitrary positive integer. Since the bound $B_5(f)$ grows monotonically in $|f|$ we can, in the case that f is reducible, find a bound for c using Lemma 2 such that the given substitution maps all irreducible factors of f into irreducible polynomials in one less variable. Together with a Kronecker like algorithm this then leads to a different polynomial-time reduction from multivariate to bivariate polynomial factorization. In the case of $v = 2$ the complete proof is given in Kaltofen [12], which, following the lines of the proof for Theorem 3, is readily extended to any fixed v . Instead of using Kronecker's algorithm one can also apply the multivariate Hensel lifting algorithm by

Musser [26] with the coefficients in $\mathbf{Q}(y_1)$. Since our evaluation guarantees that no extraneous factors can occur, all computed coefficients must actually lie in $\mathbf{Z}[y_1]$. A version of Theorem 3 can also be formulated if the coefficients are from a finite field (cf. Chistov and Grigoryev [3, Thm. 4]).

The type of substitution $y_2 = cy_1^s$ is derived from a version of the Hilbert irreducibility theorem by Franz [5] and Theorem 3 can be regarded as its effective counterpart. For the classical Hilbert irreducibility theorem, no such an effective formulation seems to be known. (See open problem 2 in § 8.)

8. Conclusion. We have shown how to overcome the extraneous factor problem during the multivariate Hensel algorithm by approximating a root and then determining its minimal polynomial, which leads to solving a system of linear equations. Our main algorithm was formulated for coefficients from a unique factorization domain and hence can also be applied to polynomials over Galois fields or algebraic extensions of the rationals. It can be shown that in both cases the algorithm works in polynomial-time.

In the case of algebraic coefficients we need a polynomial-time algorithm for univariate factorization. That this is possible is a consequence of the polynomial-time algorithm for factoring univariate polynomials over the integers (cf. Landau [19]). One usually describes an algebraic extension of the rationals by the minimal polynomial of an algebraic integer generating the field and then reduces the problem to factoring polynomials with coefficients which are algebraic integers. The ring of algebraic integers is in general not a unique factorization domain. Therefore we cannot guarantee that a solution of (4) consists of algebraic integers but one can prove that the numbers are algebraic integers within an integral quotient (cf. Weinberger and Rothschild [31, Lemma 7.1]).

In the case that the coefficients are elements from a finite field one may not be able to carry out all transformations of § 4. It may happen that good translation points w_i do not exist within the coefficient field. Then the coefficient domain has to be extended to a larger field and thus the factors returned by Algorithm 2 may have coefficients which are not in the original coefficient field. A simple trick by taking the norm (cf. Trager [27]) can then be used to determine the irreducible factors in the smaller field. This approach together with the Berlekamp algorithm (cf. Knuth [16, § 4.6.2]) gives an algorithm which works in time polynomial in the total degree of the input polynomial and the cardinality of the coefficient field, as shown in von zur Gathen and Kaltofen [8].

We conclude this paper with a list of open problems.

Problem 1. Do there exist a polynomial $p(d, v)$ and an infinite sequence of polynomials $f(x_1, \dots, x_v) \in \mathbf{Z}[x_1, \dots, x_v]$ with the following property: Any f in the sequence contains less than $p(d(f), v)$ monomials with nonzero coefficients where

$$d(f) = \max_{i=1, \dots, v} \{\deg_{x_i}(f)\};$$

moreover, there does not exist a polynomial $q(d, v)$ such that any factor of f contains less than $q(d(f), v)$ monomials with nonzero coefficients? In simple words, are there sparse polynomials with dense factors? See von zur Gathen [7] for a partial positive answer.

Problem 2. Given any polynomial $p(n)$, does there exist an infinite sequence of irreducible polynomials $f(y, x) \in \mathbf{Z}[y, x]$, $n = \deg(f)$, such that for all integers $i < p(n)$ all polynomials $f(i, x)$ are reducible? This problem asks whether there is a strongly effective version of the Hilbert irreducibility theorem.

Problem 3. Given a polynomial $f(x_1, \dots, x_v) \in \mathbf{Z}_p[x_1, \dots, x_v]$, p prime, can one determine irreducibility of f in deterministic time polynomial in $\log(p)$ deg(f)?

Acknowledgments. The problem of polynomial-time reductions for multivariate polynomial factorization was brought to my attention by Prof. George Collins. I also wish to thank Prof. Bobby Caviness and Prof. B. David Saunders for all their support. The final presentation has also benefitted from the careful remarks of one referee. This paper could not have been typeset without the help of my wife Hoang.

The examples in § 3 were computed on the MACSYMA system.

Note added in proof. A. K. Lenstra has presented another polynomial-time algorithm for factoring multivariate integral polynomials at the 10th International Colloquium on Automata, Languages and Programming. Cf. Lecture Notes in Computer Science 154, Springer, Berlin 1983, pp. 458–465.

REFERENCES

- [1] W. S. BROWN, *On Euclid's algorithm and the computation of polynomial greatest common divisors*, J. ACM, 18 (1971), pp. 478–504.
- [2] W. S. BROWN AND J. F. TRAUB, *On Euclid's algorithm and the theory of subresultants*, J. ACM, 18 (1971), pp. 505–514.
- [3] A. L. CHISTOV AND D. Y. GRIGORYEV, *Polynomial-time factoring of the multivariable polynomials over a global field*, Lomi preprint E-5-82, Leningrad 1982.
- [4] K. DÖRGE, *Zum Hilbertschen Irreduzibilitätssatz*, Math. Ann., 95 (1926), pp. 84–97.
- [5] W. FRANZ, *Untersuchungen zum Hilbertschen Irreduzibilitätssatz*, Math. Z., 33 (1931), pp. 275–293.
- [6] F. R. GANTMACHER, *Matrix Theory*, Vol. 1, Chelsea, New York, 1959.
- [7] J. VON ZUR GATHEN, *Factoring sparse multivariate polynomials*, Proc. 1983 IEEE Symposium on Foundations of Computer Science, pp. 172–197.
- [8] J. VON ZUR GATHEN AND E. KALTOFEN, *A polynomial-time algorithm for factoring multivariate polynomials over finite fields*, Proc. 1983 International Conference on Automata, Languages and Programming. Lecture Notes in Computer Science 154, Springer, Berlin, 1983, pp. 250–263.
- [9] A. O. GEL'FAND, *Transcendental and Algebraic Numbers*, Dover, New York, 1960.
- [10] G. H. HARDY AND E. M. WRIGHT, *An Introduction to the Theory of Numbers*, 5th ed. Oxford Univ. Press, Cambridge, 1979.
- [11] D. HILBERT, *Über die Irreduzibilität ganzer rationaler Funktionen mit ganzzahligen Koeffizienten*, J. Reine Angew. Math., 110 (1892), pp. 104–29.
- [12] E. KALTOFEN, *A polynomial reduction from multivariate to bivariate integral polynomial factorization*, Proc. 1982 ACM Symposium Theory of Computers, pp. 261–266.
- [13] ———, *A polynomial-time reduction from bivariate to univariate integral polynomial factorization*, Proc. 23rd, IEEE Symposium on Foundations of Computer Science, 1982, pp. 57–64.
- [14] ———, *On the complexity of factoring polynomials with integer coefficients*, Ph.D. thesis, Rensselaer Polytechnic Institute, Troy, NY, December 1982.
- [15] ———, *On the complexity of finding short vectors in an integer lattice*, Proc. 1983 European Computational Algebra Conference, Lecture Notes in Computer Science, 162, Springer, Berlin, 1983, pp. 236–244.
- [16] D. E. KNUTH, *The Art of Computer Programming*, Vol 2, *Seminumerical Algorithms*, 2nd ed., Addison-Wesley, Reading, MA, 1981.
- [17] L. KRONECKER, *Grundzüge einer arithmetischen Theorie der algebraischen Grössen*. J. Reine Angew. Math., 92 (1882), pp. 1–122.
- [18] H. T. KUNG AND J. F. TRAUB, *All algebraic functions can be computed fast*, J. ACM, 25 (1978), pp. 245–260.
- [19] S. LANDAU, *Factoring polynomials over algebraic number fields is in polynomial time*, this Journal, 14 (1985), pp. 184–195.
- [20] A. K. LENSTRA, *Factoring polynomials over algebraic number fields*, Proc. 1983 European Computational Algebra Conference, Lecture Notes in Computer Science 162, Springer, Berlin, 1983, pp. 245–254.
- [21] ———, *Factoring multivariate polynomials over a finite field*, Proc. 1983 ACM Symposium on Theory of Computers, pp. 189–192.
- [22] A. K. LENSTRA, H. W. LENSTRA AND L. LOVÁSZ, *Factoring polynomials with rational coefficients*, Math. Ann., 261 (1982), pp. 515–534.

- [23] M. T. MCCLELLAN, *The exact solution of systems of linear equations with polynomial coefficients*, J. ACM 20 (1973), pp. 563-588.
- [24] M. MIGNOTTE, *An inequality about factors of polynomials*, Math. Comp., 28 (1974), pp. 1153-1157.
- [25] J. MOSES AND D. Y. Y. YUN, *The EZGCD algorithm*, Proc. 1973 ACM National Conference, pp. 159-166.
- [26] D. R. MUSSER, *Multivariate polynomial factorization*, J. ACM, 22 (1976), pp. 291-308.
- [27] B. M. TRAGER, *Algebraic factoring and rational function integration*, in Proc. ACM Symposium on Symbolic and Algebraic Computations 1976, R. Jenks, ed., pp. 219-226.
- [28] B. L. VAN DER WAERDEN, *Modern Algebra, Vol. 1*, Engl. transl. by F. Blum, Ungar Publ., New York, 1953.
- [29] P. S. WANG, *An improved multivariate polynomial factoring algorithm*, Math. Comp., 32 (1978), pp. 1215-1231.
- [30] P. S. WANG AND B. M. TRAGER, *New algorithms for polynomial square-free decomposition over the integers*, this Journal, 8 (1979), pp. 300-305.
- [31] P. WEINBERGER AND L. P. ROTHSCHILD, *Factoring polynomials over algebraic number fields*, ACM Trans. Math. Software, 2 (1976), pp. 335-350.
- [32] D. Y. Y. YUN, *On squarefree decomposition algorithms*, in R. Jenks, ed., Proc. ACM Symposium on Symbolic and Algebraic Computations 1976 ACM, pp. 26-35.
- [33] ———, *On the equivalence of polynomial GCD and squarefree factorization problems*, Proc. MACSYMA User's Conference 77 NASA, Washington, DC, 1977, pp. 65-70.
- [34] H. ZASSENHAUS, *Polynomial time factoring of integral polynomials*, ACM SIGSAM Bulletin, 15 (May 1981), pp. 6-7.
- [35] R. E. ZIPPEL, *Probabilistic algorithms for sparse polynomials*, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, 1979.

OPTIMUM ALGORITHMS FOR A MODEL OF DIRECT CHAINING*

JEFFREY SCOTT VITTER† AND WEN-CHIN CHEN‡

Abstract. Direct chaining is a popular and efficient class of hashing algorithms. In this paper we study optimum algorithms among direct chaining methods, under the restrictions that the records in the hash table are not moved after they are inserted, that for each chain the relative ordering of the records in the chain does not change after more insertions, and that only one link field is used per table slot. The *varied-insertion coalesced hashing method* (VICH), which is proposed and analyzed in [CV84], is conjectured to be optimum among all direct chaining algorithms in this class. We give strong evidence in favor of the conjecture by showing that VICH is optimum under fairly general conditions.

Key words. analysis of algorithms, searching, information retrieval, hashing, coalesced hashing, data structures, optimality

1. Introduction. There are many classes of hashing algorithms in use today: separate chaining, coalesced hashing, linear probing, double hashing, and quadratic probing, to name a few. (More details can be found in [Knu73].) Comparisons between hashing algorithms in different classes are often difficult, because each class has its own assumptions, storage requirements, and tradeoffs. For example, some hashing algorithms (as in [Bre73]) do extra work during insertion in order to speed up later searches. In some applications, the preferred class of hashing methods is determined by the special nature and requirements of the application. The task is then to find the optimum algorithm within that class.

This paper is concerned with optimum algorithms within one popular class of hashing algorithms—*direct chaining without restructuring*. This implies that the lists coalesce. Throughout this paper, we will denote the number of inserted records by N and the number of slots in the hash table by M' . We assume that there is a predefined and quickly computed *hash function*

$$(1) \quad \text{hash: } \{\text{all possible keys}\} \rightarrow \{1, 2, \dots, M\}$$

that assigns each record to its *hash address* in a uniform manner. The first M slots, which serve as the range of the hash function, are called the *address region*; the remaining $M' - M$ slots make up the *cellar*.

Direct chaining works as follows: The search for a record with key K begins at slot $\text{hash}(K)$ and continues through the linked chain of records until either the record is found (i.e., the search is *successful*) or else the end of the list is reached (i.e., the search is *unsuccessful*). When a record with a key K is inserted, it must become part of the chain that includes slot $\text{hash}(K)$, so that later searches for that record will succeed.

In this paper we study *optimum* direct chaining algorithms under the following model: the records cannot be moved once they are inserted into the hash table (e.g., the records might be “pinned” to their locations by pointers to them from outside the table, or the records might be very large so that moving them is expensive), the relative ordering of the records in each chain does not change after more records are added,

* Received by the editors December 20, 1983, and in revised form September 24, 1984. This research was supported by an IBM contract.

† Department of Computer Science, Brown University, Providence, Rhode Island 02912. The research of this author was supported in part by the National Science Foundation under grant MCS-81-05324 and by ONR and DARPA under contract N00014-83-K-0146 and ARPA Order No. 4786.

‡ Department of Computer Science, Brown University, Providence, Rhode Island 02912. Present address, GTE Laboratories Inc., Waltham, Massachusetts 02245.

and there is only one link field per table slot. This model does not allow restructuring of the hash table while the table is being constructed.

Under this model, when a record *collides* with another record during insertion (i.e., its hash address is already occupied), an empty slot is allocated to store the new record, and that slot is linked into the chain containing slot *hash* (K) at some point in the chain after slot *hash* (K). We call a record that collides during insertion a *collider*. Insertion algorithms in this model can differ from one another only in the ways that the following two decisions are made:

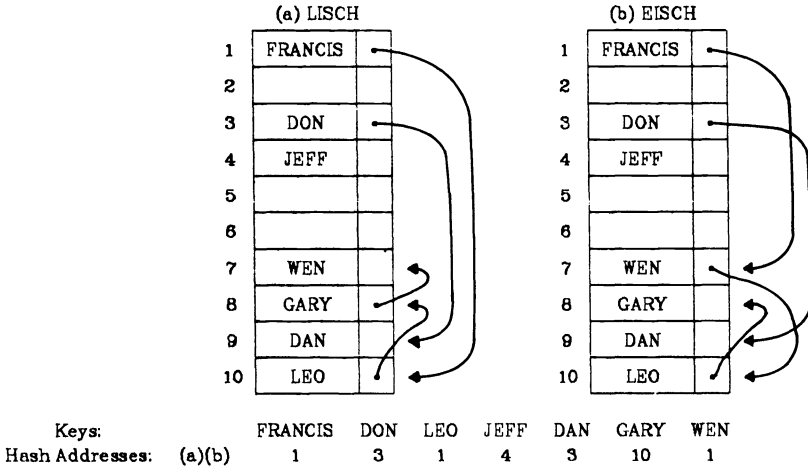
- (1) Which empty slot is allocated to store the collider?
- (2) At what point in the chain following slot *hash* (K) should the collider be linked?

The measures of performance we use to compare algorithms is the *number of probes per successful search* and the *number of probes per unsuccessful search*. In both cases this is the number of distinct slots accessed during the search. We use the probability model that the M^N sequences of hash addresses are equally likely. For successful searches, we also assume that each of the N inserted records in the hash table is equally likely to be the object of the successful search. For insertions and unsuccessful searches, we assume that each of the M address region slots is equally likely to be the hash address for the unsuccessful search. In other words, insertions and searches are assumed to be *random*.

When there is no cellar, the way in which decision 1 is made is not important, as far as the average search performance is concerned. In the case in which there is a cellar, most methods use a statically-ordered available-slot list, in which empty slots are allocated in some fixed relative order. Performance seems to be best when cellar slots get higher priority over noncellar slots on the available-slot list. When that is the case, a collider is stored in an empty cellar slot, if one is available. When the cellar gets full, subsequent colliders must be stored in the address region. This may cause collisions with records inserted later. For example, in Fig. 1, LEO collides with FRANCIS during insertion and is stored in the address region (in slot 10), since there is no cellar. When GARY is inserted later, GARY collides with LEO at slot 10. Thus GARY and LEO become part of the same chain, even though they have different hash addresses. This phenomenon, which we call *coalescing*, tends to make search times longer. Intuitively, it makes sense to give higher priority to the cellar slots on the available-slot list, because storing a collider in the address region can cause coalescing to occur during a later insertion.

Several methods have been proposed for handling decision 2, all having the generic name of *coalesced hashing*. The original method, *late-insertion coalesced hashing* (LICH), was introduced in [Wil59] and analyzed in [Knu73], [Gui76], [GK81], [Vit82b], [Vit83], and [CV84]. In LICH, a collider is linked to the *end* of the chain that contains slot *hash* (K). The *early-insertion coalesced hashing* method (EICH) proposed in [Vit82b] and [Kno84] inserts each collider into the chain at the point *immediately after* slot *hash* (K).

For the special case of *standard coalesced hashing* (in which there is no cellar) these two methods are referred to as LISCH and EISCH. An example is given in Fig. 1. The record WEN collides with FRANCIS at slot 1. In the LISCH method illustrated in Fig. 1(a), WEN is linked at the end of the chain containing FRANCIS. With EISCH in Fig. 1(b), WEN is inserted into the chain at the point between FRANCIS and LEO. The average successful search time in Fig. 1(b) is slightly better than in Fig. 1(a), because inserting WEN immediately after FRANCIS (rather than at the end of the chain) reduces the search time for WEN from four probes to two and increases the



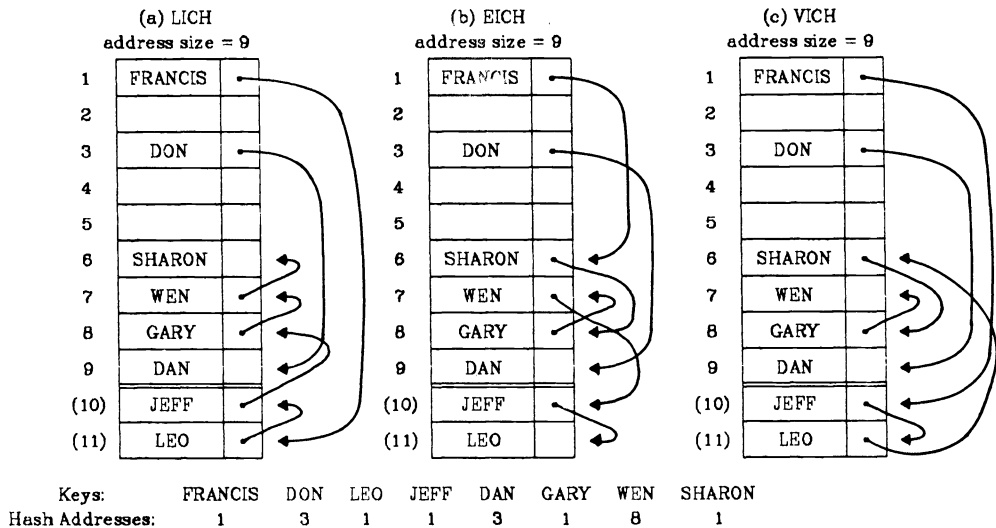
ave. # probes per succ. search: (a) $13/7 \approx 1.86$, (b) $12/7 \approx 1.71$.
 ave. # probes per unsucc. search: (a) $17/10 = 1.7$, (b) $17/10 = 1.7$.

FIG. 1. Standard coalesced hashing, $M' = M = 10$, $N = 7$. (a) LISCH, (b) EISCH.

search time for LEO from two probes to three. The result is a net decrease of one probe. The expected search performance for EISCH is derived in [CV83] and [Kno84]; EISCH results in faster searches, on the average, than does LISCH.

When there is a cellar, however, LICH performs better than EICH, as illustrated in Fig. 2. The insertion of WEN using EICH in Fig. 2(b) causes both cellar records LEO and JEFF to move one more link further from their hash addresses. That does not happen using LICH in Fig. 2(a).

The varied-insertion coalesced hashing method (VICH) was introduced in [Vit82b] as a means of combining the strong points of both LICH and EICH without their



ave. # probes per unsucc. search: (a) $18/9 = 2.0$, (b) $24/9 \approx 2.67$, (c) $18/9 = 2.0$.
 ave. # probes per succ. search: (a) $21/8 \approx 2.63$, (b) $22/8 = 2.75$, (c) $20/8 = 2.5$.

FIG. 2. Coalesced hashing, $M' = 11$, $M = 9$, $N = 8$. (a) LICH, (b) EICH, and (c) VICH.

weaknesses. In VICH, the collider is linked into the chain directly after its hash address (as in EICH) *except* when the cellar is full, there is at least one cellar slot in the chain, and the hash address of the collider is the location of the first record in the chain; in that case, the collider is linked into the chain directly after the last cellar slot in the chain. When there is no cellar, VISCH is identical to EISCH. An example of VICH appears in Fig. 2(c). The analyses of LICH, EICH, and VICH given in [CV84] show that VICH performs better, on the average, than both LICH and EICH.

In the next section we conjecture that VICH is *optimum* among all direct chaining methods, under the model explained above. The main result of this paper is a vote in favor of this conjecture, showing that VICH is optimum under the above conditions when cellar slots are given priority on the available-slot list.

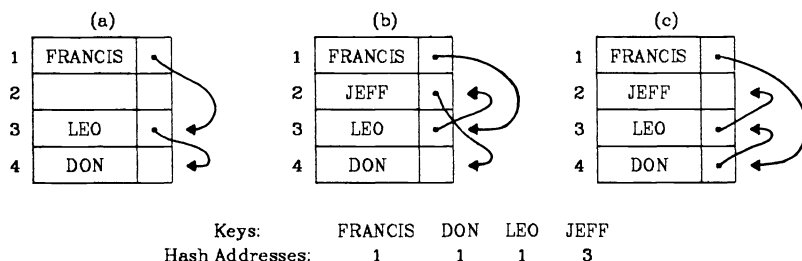
2. Search-time optimality of varied-insertion. In this section we investigate the search-time optimality of VICH among chaining methods that insert records directly into the hash table. The sizes of the address region and cellar are fixed. We assume that records cannot be moved once they are inserted. We also assume that the relative ordering of the records in the chains does not change after more records are inserted. In other words, the optimization illustrated in Fig. 3(c) is not allowed. Finally, we assume that there is only one link field per table slot.

The major open question is whether VICH is optimum under this model. We conjecture that the answer is yes. We will use the term *admissible* to refer to any direct chaining insertion algorithm that satisfies the assumptions of the above model.

Conjecture. Varied-Insertion Coalesced Hashing (VICH) gives the optimum expected search times among all admissible chaining methods, with the probability assumptions that the M^N sequences of hash addresses are equally likely and that insertions and searches are random.

In this section we give strong support for the conjecture: We show that VICH uses a greedy method of inserting records, that is, VICH is a locally optimum admissible method for inserting a single record. For the special case in which there is no cellar, we show that VICH is not only locally optimum, but also globally optimum. The main result is showing that VICH is globally optimum among all admissible chaining methods that give the cellar slots priority on the available-slot list.

In coalesced hashing we typically allocate available empty slots in the order M' , $M' - 1$, $M' - 2$, \dots , which means that the cellar slots are allocated before the address region slots. However Lemma 1 shows that for any given ordering of the available-slot



ave. # probes per succ. search: (b) $9/4 = 2.25$, (c) $8/4 = 2.0$.

FIG. 3. Hash table (a) is an optimum arrangement of the first three inserted records FRANCIS, DON, and LEO, as far as subsequent searches are concerned. Table (b) pictures the result of inserting JEFF using VICH. Table (c) achieves better successful search times than (b) by reordering the chain so that DON precedes LEO. Assuming that the records cannot be moved once they are inserted, there is no optimum arrangement of the four records in which LEO precedes DON as in (a).

list, the chains that are formed are the same regardless of which admissible chaining method is used, except for the order of the individual records within the chains.

LEMMA 1. *For any given ordering of the available-slot list and for any admissible chaining methods, the partition of the inserted records into chains is the same.*

Proof. We prove this lemma by induction on the number of records inserted in the table. Assume that the partition of the inserted records into chains is the same for all admissible chaining methods after k records are inserted. Let record R (with key K) be the next record inserted. If R does not collide when inserted, then all admissible chaining methods insert R at its hash address $hash(K)$. If R collides when inserted, then all admissible chaining methods link R into the chain containing location $hash(K)$. Thus the partition of the records into chains is still the same for all admissible chaining methods after $k+1$ insertions. This proves the lemma. \square

It is easy to see that permuting the order of the cellar slots on the available-slot list does not affect search performance at all. The following lemma shows that the order of the address region slots on the available-slot list can be permuted among themselves without affecting the *average* search performance. The rigorous statement of the lemma uses the terminology that two hash tables are *homotopic* if and only if their chains can be paired off in a 1-1 correspondence so that the search times for the i th records in two corresponding chains are the same, for each i and for any pair of corresponding chains. The formal definition of homotopic appears in [Vit82a], which used the notion in connection with deletion algorithms that preserve randomness.

LEMMA 2. *For any admissible chaining algorithm, and for any two orderings of the address region slots in the same fixed positions on the available-slot list, there is a 1-1 correspondence between the two classes of M^N hash tables obtained by using the two available-slot lists such that each pair of corresponding tables is homotopic.*

Proof. We prove this lemma by constructing explicitly the 1-1 correspondence. Let a_1, a_2, \dots, a_M and $\sigma(a_1), \sigma(a_2), \dots, \sigma(a_M)$ be the two orderings of the available-slot list. The hash table obtained by inserting k records with hash addresses h_1, h_2, \dots, h_k using the first ordering of the available-slot list is homotopic to the hash table obtained by the insertion of records with hash addresses $\sigma(h_1), \sigma(h_2), \dots, \sigma(h_k)$ using the latter ordering. For each i , the record in slot i in the first table is in slot $\sigma(i)$ in the second table. \square

The following definitions are used to prove the remaining results of this section. A chain of records R_1, R_2, \dots, R_k , inserted in that order by linking algorithms A_1, A_2, \dots, A_k , respectively, can be defined as follows: The chain $[(R_1, A_1)]$ is the chain containing R_1 only. Given a chain $L_1 = [[\dots[(R_1, A_1)], (R_2, A_2)], \dots], (R_{k-1}, A_{k-1})]$, the chain $L = [L_1, (R_k, A_k)]$ is obtained by linking R_k into L_1 by using algorithm A_k . For simplicity, we will use $[(R_1, A_1), \dots, (R_k, A_k)]$ to denote L . If a record R is contained in a chain $L = [(R_1, A_1), \dots, (R_k, A_k)]$, then we define $Search(R, L)$ to be the number of probes required to search successfully for R in L , and $Search(L)$ to be $\sum_{1 \leq i \leq k} Search(R_i, L)$. We also define $loc(R)$ to be the absolute location in the hash table of the slot containing R , and $\phi(R, L)$ to be the number of records that are stored in slots in the chain L following R , whose hash addresses are either $loc(R)$ or one of the slots before R in L .

Theorem 1 shows that for successful searches, VICH is locally optimum among all admissible chaining methods. VICH uses a greedy method of inserting records, in that each individual insertion is done so as to minimize the resulting average successful search time.

THEOREM 1. *Given a record R with key K and a chain L containing location $hash(K)$, an optimum place to link R into L in order to minimize $Search(L')$, where L'*

is the chain obtained by linking R into L , is immediately before the first noncellar slot following slot $hash(K)$ in the chain, or else at the end of the chain if no such noncellar slot exists.

Proof. If slot $hash(K)$ is the last slot in L , then all admissible chaining methods link R immediately after slot $hash(K)$. Therefore, we will assume that before R is inserted, slot $hash(K)$ is not the last slot in L and that record R_1 is stored in a slot in chain L that follows slot $hash(K)$. Suppose that there are two different methods A and B that link record R into chain L . Method A links R immediately after the slot containing R_1 . Method B links R immediately before the slot containing R_1 . Let L_A denote the chain $[L, (R, A)]$ and L_B denote the chain $[L, (R, B)]$. We will prove the theorem by showing that

$$(2) \quad Search(L_A) \cong Search(L_B),$$

and that if the slot that contains R_1 is a cellar slot, then

$$(3) \quad Search(L_A) = Search(L_B).$$

Since method A links R into chain L immediately after the slot containing R_1 and method B links R into chain L immediately before the slot containing R_1 , we have

$$(4) \quad Search(R, L_A) = Search(R, L_B) + 1;$$

and

$$(5) \quad Search(R_1, L_B) = Search(R_1, L_A) + 1.$$

Formula (5) follows from the fact that R_1 cannot be stored at its hash address, or else it would not be in the chain L . For those records R' that are stored in slots following R_1 in L , and whose hash addresses are $loc(R_1)$, we have

$$(6) \quad Search(R', L_A) = Search(R', L_B) + 1.$$

Since the hash address of R' is $loc(R_1)$ and R is linked immediately after R_1 in L_A , it requires one more probe to search for R' in L_A . Note that if R_1 is stored in the cellar, then there are no records R' with hash addresses $loc(R_1)$. Finally, if \bar{R} is one of the remaining records in both chains, we have

$$(7) \quad Search(\bar{R}, L_A) = Search(\bar{R}, L_B).$$

This proves (2) and (3), and thus proves the theorem. \square

The following theorem shows that in the case in which there is no cellar, then VISCH is not only locally optimum, but also globally optimum.

THEOREM 2. *Assume that there is no cellar. For any set of records R_1, R_2, \dots, R_k that forms a chain, with the assumption that the M^k possible sequences of hash addresses h_1, h_2, \dots, h_k are equally likely, the average unsuccessful and successful search times on the chain are optimized by the VISCH (=EISCH) ordering of the records in the chain.*

Proof. Lemma 1 shows that the partition of the inserted records into chains is the same for all admissible chaining methods. Since the average unsuccessful search time on a chain depends on the length of the chain and not on the ordering of the records within the chain, then by Lemma 1, the average unsuccessful search times are the same for all admissible chaining methods.

To prove that VISCH is optimum for the successful search case, we need the following formula. Let L be the chain $[(R_1, A_1), \dots, (R_k, A_k)]$; then

$$(8) \quad Search(L) = k + \sum_{1 \leq i \leq k} \phi(R_i, L).$$

This formula can be proved as follows: From the definition of $Search(L)$, we have

$$(9) \quad Search(L) = \sum_{1 \leq i \leq k} Search(R_i, L).$$

Now consider each record R_i in chain L . If $Search(R_i, L) = s$, then s probes are required to search for R_i in L . Let $R_{i_1}, \dots, R_{i_{s-1}}$ be the records stored in the slots in L that are probed while searching for R_i . The search for R_i contributes 1 to each of the following terms: $\phi(R_{i_1}, L), \dots, \phi(R_{i_{s-1}}, L)$. It also contributes 1 to the term k in the right-hand side of (8). Therefore, the search for R_i contributes s to both sides of (8). This proves (8).

Now we prove the optimality of VISCH for successful searches by induction on k , the number of records in the chain. Assume that for a given set of records R_1, \dots, R_k , with random hash addresses h_1, \dots, h_k , the average successful search times on the chain are optimized by the VISCH ordering of the records in the chain. We will show that VISCH still gives the optimum ordering for the records R_1, \dots, R_{k+1} , where R_{k+1} is the next record inserted. The hash address of R_{k+1} can be any of the M address region slots, with equal probability. Formally, if we let L_V denote the chain $[(R_1, VISCH), \dots, (R_k, VISCH)]$ and let L_A denote the chain $[(R_1, A), \dots, (R_k, A)]$, for any admissible chaining method A , then with the assumption of induction

$$(10) \quad \sum_{*} Search(L_V) \leq \sum_{*} Search(L_A),$$

we will show

$$(11) \quad \sum_{*} \sum_{1 \leq i \leq k} Search([L_V, (S_i, VISCH)]) \leq \sum_{*} \sum_{1 \leq i \leq k} Search([L_A, (S_i, VISCH)])$$

and

$$(12) \quad \sum_{*} \sum_{1 \leq i \leq k} Search([L_A, (S_i, VISCH)]) \leq \sum_{*} \sum_{1 \leq i \leq k} Search([L_A, (S_i, A)]),$$

where S_i is a record with hash address $loc(R_i)$ and the symbol $*$ under \sum represents the summation condition “all possible sequences of hash addresses h_1, \dots, h_k such that records R_1, \dots, R_k are linked together to form a chain.” Inequalities (11) and (12) combined prove that VISCH is optimum for successful searches.

Inequality (12) is true from Theorem 1, which showed that VICH is locally optimum. Inequality (11) is shown in the following way by applying (8): If the record S_i is linked into chain L_V immediately after the slot containing R_i by using VISCH, then we have

$$(13) \quad Search([L_V, (S_i, VISCH)]) = 2 + \phi(R_i, L_V) + Search(L_V).$$

This formula is true, since after the insertion of S_i , it requires two probes to search for S_i in $[L_V, (S_i, VISCH)]$, and it requires one more probe to search for those records stored in slots in L_V that follow R_i and whose hash addresses are either $loc(R_i)$ or one of the slot before R_i in L_V . From (13), the left-hand side of (11) is equal to

$$(14) \quad \begin{aligned} \sum_{*} \sum_{1 \leq i \leq k} (2 + \phi(R_i, L_V) + Search(L_V)) &= \sum_{*} (2k + k Search(L_V) + \sum_{1 \leq i \leq k} \phi(R_i, L_V)) \\ &= \sum_{*} (k + (k + 1) Search(L_V)). \end{aligned}$$

The last equality follows from (8). Similarly, the right-hand side of (11) is equal to

$$(15) \quad \sum_{*} (k + (k + 1) \text{Search}(L_A)).$$

Thus, (11) follows from (10) immediately.

From the above arguments, we conclude that VISCH gives the optimum ordering of records in the chain that minimizes both the successful and unsuccessful searches times. This proves the theorem. \square

The weakness of Theorem 2 is that EICH is also a greedy insertion method, but with a cellar EICH is definitely not optimum. The next theorem strengthens the previous results. It shows that when the cellar slots get priority on the available-slot list, VICH is globally optimum.

THEOREM 3. *Assume that the cellar slots are given priority on the available-slot list. For any set of records R_1, R_2, \dots, R_k that forms a chain, with the assumption that all possible hash addresses h_1, h_2, \dots, h_k are equally likely, the average unsuccessful and successful search times on the chain are optimized by the VICH ordering of the records in the chain.*

Proof. We first note that for each chain the hash address of each cellar slot is the location of the first slot of the chain. Now we show that VICH optimizes the average unsuccessful search times. Let us assume that the noncellar slots in the chain are in relative positions $p_1, p_2, \dots, p_{a-1}, p_a$ in the chain, counting backwards from the end of the chain. Note that $1 \leq p_1 \leq \dots \leq p_{a-1} \leq p_a = k$, where k is the length of the chain. The average unsuccessful search time on the chain is

$$(16) \quad \frac{1}{a}(p_a + p_{a-1} + \dots + p_1),$$

which can be minimized by letting $p_1 = 1, p_2 = 2, \dots, p_{a-1} = a - 1$. This means that all the cellar slots in the chain immediately follow the first slot in the chain. This is feasible, since the hash addresses of all the cellar slots are the location of the first slot of the chain. VICH yields the above ordering, and thus minimizes the average unsuccessful search times on the chain.

To prove that VICH optimizes the average successful search times, we will show that (a) the optimum placements of the cellar slots in the chain are immediately after the first slot of the chain, and that (b) the optimum relative ordering of the noncellar slots is the ordering obtained by using VICH.

(a) Assume that an ordering of records in the chain is obtained by using method A , in which a cellar slot immediately follows a noncellar slot. Let records R_1 and R_2 be the records stored in the noncellar slot and the cellar slot, respectively. We assume that R_1 is *not* the start of the chain. Assume also that another ordering of records in the chain is obtained by using method B , which is the same as that for A except that R_2 immediately precedes R_1 in the chain. Let L_A denote the chain obtained by using method A , and L_B denote the chain obtained by using method B . It suffices to show that

$$(17) \quad \text{Search}(L_A) \cong \text{Search}(L_B).$$

Since the hash address of R_2 is the location of the first slot of the chain, we have

$$(18) \quad \text{Search}(R_2, L_A) = \text{Search}(R_2, L_B) + 1.$$

Also, we have

$$(19) \quad \text{Search}(R_1, L_B) = \text{Search}(R_1, L_A) + 1,$$

since the slot containing R_2 precedes immediately the slot containing R_1 in L_B . For those records R' that are stored in slots following R_1 in L_A , and whose hash addresses are $loc(R_1)$, we have

$$(20) \quad Search(R', L_A) = Search(R', L_B) + 1.$$

This follows since $loc(R_2)$ must be probed to search for R' in chain L_A . For the remaining records \bar{R} in both chains, we have

$$(21) \quad Search(\bar{R}, L_A) = Search(\bar{R}, L_B).$$

This shows that the optimum placements of the cellar slots are immediately after the first slot of the chain.

(b) By the same arguments as those in the proof of Theorem 2, we can prove that the optimum relative ordering of the noncellar slots is the ordering obtained by using the VICH method.

From (a) and (b), we conclude that the average successful search time on the chain is optimized by the VICH ordering of the records in the chain. \square

In order to prove the conjecture that VICH is globally optimum, it suffices to show that the best way to allocate empty slots for colliders is to have an available-slot list in which the cellar slots have higher priority than the address region slots. The conjecture would then follow from Lemma 1, Lemma 2 and Theorem 3.

The difficulty with proving the conjecture is due to the many exotic hash algorithms that must be considered if the cellar slots are not given priority on the available-slot list. The priorities assigned to slots on the available-slot list might be dynamic, for example. For that reason, the optimum algorithm in our model could turn out to be a method that is not practical. We believe that such is not the case. We conjecture that VICH, which has a very efficient implementation, is optimum in our model.

3. Conclusions and open problems. We have given strong evidence in support of our conjecture that VICH is optimum among all direct chaining methods, under the assumptions that the records are not moved once they are inserted, that for each chain the relative ordering of its record does not change after further insertions, and that there is only one link field per table slot. In particular, we have shown that the conjecture is true under the additional condition that cellar slots are given priority on the available-slot list. Intuition suggests that this extra condition will always be true for optimum algorithms under the above assumptions, however, determining whether the conjecture is true in general seems to be quite challenging. If VICH is shown not to be optimum, it is hopeful that the insights gained from the proof will lead to the construction of an optimum algorithm.

There are other interesting open problems concerning this model of hashing. One problem is to study the performance of coalesced hashing in external searching, as discussed in [Vit82b]. Another problem concerns deletion algorithms that *preserve randomness*. Preserving randomness means that deleting a record is in some sense like never having inserted it. In particular, the formulas for the average search times after N random insertions intermixed with d deletions are the same as the formulas for the average search times after $N - d$ random insertions. The formal notion of what it means to preserve randomness is defined in [Vit82a]. A deletion algorithm for coalesced hashing is given in [Vit82a] and shown to preserve randomness for late-insertion standard coalesced hashing (LISCH). The authors have recently discovered deletion algorithms that preserve randomness for LICH, EICH, and VICH. It seems that in order to preserve randomness, a deletion algorithm must relocate some records from

time to time, which may not be possible if the records are “pinned” to their locations and are not allowed to be moved. Deletion algorithms that do not relocate records (and do not preserve randomness) should therefore also be studied. It is an open problem to determine how the average search times are affected by deletion algorithms that do not preserve randomness.

REFERENCES

- [Bre73] R. P. BRENT, *Reducing the retrieval time of scatter storage techniques*, Comm. ACM, 16 (1973), pp. 105–109.
- [CV83] W. C. CHEN AND J. S. VITTER, *Analysis of early-insertion standard coalesced hashing*, this Journal, 12 (1983), pp. 667–676.
- [CV84] ———, *Analysis of new variants of coalesced hashing*, ACM Trans. Database Systems, to appear.
- [GK81] D. H. GREENE AND D. E. KNUTH, *Mathematics for the Analysis of Algorithms*, Birkhauser, Boston, 1981.
- [Gui76] L. J. GUIBAS, *The analysis of hashing algorithms*, PhD dissertation, Computer Science Dept., Technical Report STAN-CS-76-556, Stanford Univ., Stanford, CA, August 1976.
- [Kno84] G. D. KNOTT, *Direct chaining with coalesced lists*, J. Algorithms, 5(1) (1984), pp. 7–21.
- [Knu73] D. E. KNUTH, *The Art of Computer Programming. Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
- [Vit82a] J. S. VITTER, *Deletion algorithms for hashing that preserve randomness*, J. Algorithms, 3 (1982), pp. 261–275.
- [Vit82b] ———, *Implementations for coalesced hashing*, Comm. ACM, 25 (1982), pp. 911–926.
- [Vit83] ———, *Analysis of the search performance of coalesced hashing*, J. Assoc. Comput. Mach., 30 (1983), pp. 231–258.
- [Wil59] F. A. WILLIAMS, *Handling identifiers as internal symbols in language processors*, Comm. ACM, 2 (1959), pp. 21–24.

COMPOSING FUNCTIONS TO MINIMIZE IMAGE SIZE*

M. R. GAREY† AND D. S. JOHNSON†

Abstract. We show that, given a collection F of functions from a finite set D to itself, one can, in polynomial time, find a composition f of functions in F for which the size of $f(D)$ is minimized. This is to be contrasted with the fact that it is PSPACE-complete to determine whether a *specific* function f is a composition of functions in F . The running time of our algorithm is $O(|D|^2(|D| + |F|))$, and this bound can be improved if an appropriately abbreviated representation of F is used. The problem first arose in connection with the minimization of conjunctive queries for relational databases.

Key words. Computational complexity, algorithms, database theory, minimization

1. Introduction. Let D be a finite set, and let $F = \{f_1, f_2, \dots, f_n\}$ be a collection of (not necessarily one-to-one) functions from D to itself, represented as sets of ordered pairs. Define F^* to be the closure of F under function composition, and let $F_{\min} = \min\{|f(D)| : f \in F^*\}$. In this paper we consider the problem of finding a function $f \in F^*$ such that $|f(D)| = F_{\min}$.

A restricted version of this problem arose in a problem of optimizing queries for relational databases [2], in which the set D corresponds to the collection of “conjuncts” in a “conjunctive query” Q , the functions in F are homomorphisms of the query to itself, and F_{\min} is the number of conjuncts in the minimum query equivalent to Q . The set F in this application was determined by the internal structure of the conjuncts, and hence was constrained in a variety of ways, not all of them easily quantifiable. However, by using this internal structure, [2] was able to derive an algorithm that computes F_{\min} for such sets, and runs in time $O(|D|^2|Q|)$, where $|Q| \geq |D|$ measures the total *size* of the query Q , taking into account the internal structure of its conjuncts (the elements of D).

When one turns to the general problem, where *arbitrary* sets of functions F are allowed and the elements of D have no internal structure, the possibility of a polynomial time algorithm for finding an $f \in F^*$ with $|f(D)| = F_{\min}$ seems considerably diminished. One’s pessimism is further bolstered by the fact that the closely related problem, “given D, F , and a function $f : D \rightarrow D$, is $f \in F^*$?” is PSPACE-complete [3].

In this paper we show that, surprisingly, the general problem *can* be solved in polynomial time, with a time bound comparable to that for the special case mentioned above. Our approach is to divide the overall problem into two appropriately formulated subtasks, each involving a specially designed data structure. In §2 we consider the second of these subtasks; in §3 we consider the first, and in §4 we put the two parts together to obtain an $O(|D|^2(|D| + |F|))$ algorithm. We also observe how to modify the algorithm to take advantage of special properties of the set F , such as those that hold in the above-mentioned database application.

2. The collapsibility graph. The key intermediate structure needed to solve our problem is an edge-labelled graph $G_C(F)$, which we shall call the *collapsibility graph* for F . The graph $G_C(F)$ has D for its vertex set, and an edge $\{d, d'\}$ if and only if there is some $f \in F^*$ such that $f(d) = f(d')$, with the label for that edge being a description of such a function f . It is not difficult to see that this graph can be

*Received by the editors January 25, 1983, and in revised form November 30, 1983. This paper was typeset at AT&T Bell Laboratories, Murray Hill, New Jersey, using the *troff* program running under the UNIX® operating system. Final copy was produced on December 17, 1984.

†AT&T Bell Laboratories, Murray Hill, New Jersey 07974

constructed in polynomial time, but we shall postpone the details of its construction for the next section. For now, let us consider the crucial property of the graph.

LEMMA 1. *If $D' \subseteq D$ is an independent set in $G_C(F)$, then $F_{\min} \geq |D'|$.*

Proof. By the definition of $G_C(F)$, if D' is an independent set, then, for all $f \in F$, $f(D')$ is also an independent set and $|f(D')| = |D'|$. Thus, by induction, for all $f \in F^*$, the same properties hold. Therefore, for all such f , $|f(D)| \geq |f(D')| \geq |D'|$, and the lemma follows from the definition of F_{\min} . \square

In fact, we will see that F_{\min} equals the size of the *maximum* independent set in $G_C(F)$. Although the problem of finding maximum independent sets is in general NP-hard, in this case we are more fortunate. The following procedure determines F_{\min} and constructs our desired function f , given $G_C(F)$. The procedure can be described in terms of “pebbles.”

Collapsing Procedure

1. Assign a pebble to each vertex d of $G_C(F)$.
Let $f^* = I$ (the identity function).
2. While there are two pebbled vertices connected by an edge, do the following:
 Let f be a function labelling an edge connecting two pebbled vertices,
 and set $f^* = f \circ f^*$ (the composition of f with f^*).
 For each pebbled vertex d (in parallel), move the pebble on d to $f(d)$.
 Delete all but one pebble from each multiply-pebbled vertex.

LEMMA 2. *After executing the “Collapsing Procedure,” the number of pebbles remaining equals F_{\min} and f^* is a function in F^* with $f^*(D) = F_{\min}$.*

Proof. Since f^* is a composition of functions in F^* , it is itself in F^* by definition. Its image is the set of vertices that are pebbled when the procedure halts. Since the procedure did halt, these vertices must constitute an independent set in G_C , and hence $F_{\min} \geq |f^*(D)|$, by Lemma 1. Equality follows by the definition of F_{\min} , since, as argued above, $f^* \in F^*$. \square

The time required for performing the Collapsing Procedure can be bounded by $O(|D|^3)$: The *while* loop can be executed at most $|D|$ times, since each execution results in the removal of a pebble. Each execution spends at most $O(|D|^2)$ time looking for adjacent pebbled vertices, and then at most $O(|D|)$ time moving pebbles. Standard data structures can be used to keep track of pebble locations efficiently. If one wishes to include the time for reading in the graph $G_C(F)$, the same time bound holds, since $G_C(F)$ has $|D|$ vertices, fewer than $|D|^2$ edges, and each edge label is a function description consisting of $|D|$ ordered pairs of elements of D - we assume that the computer word length is at least $\log_2 |D|$ and hence each element of D can be described using a single word. In what follows we also assume that random access memory is sufficiently large that elements of D and F can be accessed in constant time when needed. If either of these assumptions is violated, our stated running time bounds will need to be multiplied by appropriate logarithmic factors.

3. The pair-map graph. To complete our algorithm, we must show how to construct $G_C(F)$. In other words, for each pair (d, d') of distinct elements in D , we must determine whether there exists a function $f \in F^*$ such that $f(d) = f(d')$ and, if so, generate such a function. We can do this using a second labelled graph data structure, the *pair-map graph* $G_P(F)$.

The directed graph $G_P(F)$ has a vertex set $D \times D$, with $((d, d'), (e, e'))$ in the arc set if and only if there is an $f \in F$ such that $f(d) = e$ and $f(d') = e'$, in which case the

arc is labelled by such an f . Note that $G_P(F)$ has $|D|^2$ vertices and at most $|F|$ outgoing arcs per vertex, for a total size of $O(|D|^2|F|)$, and that $G_P(F)$ can be constructed in time proportional to this size, given descriptions of the functions in F as sets of ordered pairs.

Given the pair-map graph, the existence of a function $f \in F^*$ with $f(d) = f(d')$ can easily be determined in time proportional to the size of $G_P(F)$, merely by doing a breadth-first search from the vertex (d, d') , looking for vertices of the form (e, e) . If one is found, the path from (d, d') to it can be at most $O(|D|^2)$ arcs long, so the desired function f can be computed by $O(|D|^2)$ composition operations, for a total time of $O(|D|^3)$. Since this must be done for every pair (d, d') of distinct elements of D , the overall time for constructing $G_C(F)$ in this straightforward manner is $O(|D|^4(|D| + |F|))$.

However, we can be more efficient than this. Suppose that we start at the vertices of the form (e, e) and work backwards, thus determining all vertices that can reach such a vertex during just *one* pass over $G_P(F)$, i.e., in time $O(|D|^2|F|)$. We can also avoid unnecessary recomputations of function compositions. The idea is to label each vertex (d, d') , as it is encountered, with a function $f_{d, d'}$ that collapses d and d' . Initially, vertices of the form (e, e) are labelled with the identity function. Suppose now that (e, e) is labelled, and we encounter a new vertex (d, d') in our backward search by traveling back along the arc $a = ((d, d'), (e, e))$ from (e, e) . If f_a is the label of arc a in $G_P(F)$, then the label of (d, d') is $f_{d, d'} = f_a \circ f_{e, e}$. This reduces the total amount of time to compute all the edge labels for $G_C(F)$ to at most $O(|D|)$ for each vertex in $G_P(F)$, or a total of $O(|D|^3)$, for an overall time of $O(|D|^2(|D| + |F|))$ to construct $G_C(F)$.

4. Special case efficiency. Given the procedures specified in the last two sections for constructing the collapsibility graph $G_C(F)$ and using it, we conclude that the problem of determining F_{\min} and finding an $f \in F^*$ such that $|f(D)| = F_{\min}$ can be solved in time $O(|D|^2(|D| + |F|))$. Assuming $|F| > |D|$, the overall running time is dominated by that for constructing $G_C(F)$, since the collapsing procedure of §2 only takes time $O(|D|^3)$.

In this section we observe how the construction of $G_C(F)$ can be modified to take advantage of certain properties of the set of functions F , such as those encountered in the already mentioned database application of [2]. In that application, the functions in F obey the following restriction (among others):

(R). If f, f' are distinct functions in F , then $f(a) = f'(a)$ implies $f(a) = a$.

Restriction (R) implies that, if the functions of F are considered as sets of ordered pairs, then for any two distinct elements a, a' of D , the ordered pair (a, a') can occur in at most one of the functions of F . Thus $|F| \leq |D|^2$, and the more functions there are, the more likely they are to leave a large portion of their domain untouched. We can take advantage of this last observation by using a more concise representation for the functions in F . So far, we have considered each to be a set of $|D|$ ordered pairs, for a total of $O(|D| \cdot |F|)$ storage space. We can save some of this space, without any loss of information, by deleting all pairs of the form (e, e) . This can be done in time $O(|D| \cdot |F|)$, which will still be dominated by the times for the later parts of the algorithm. Let $|S|$ denote the size of the resulting representation of F . Note that restriction (R), mentioned above, implies that $|S| = O(|D|^2)$.

We now show how we can construct $G_P(F)$ in time $O(|D| \cdot |S|)$ using this new representation, and thus convert any savings in storage to savings in running time

(and, at the same time, obtain a tighter bound on the size of $G_P(F)$). We begin by constructing a set S of triples (f, d, d') , where $f \in F$, $d \in D$, and $f(d) = d' \neq d$. This takes time $O(|S|)$ given the concise representation of F . In constructing these triples, we use a bucket-sort technique [1] to partition them into sets S_d , $d \in D$, where S_d consists of all those triples with d as second component. Assuming that the original description of F had its pairs ordered by first component, with all the pairs for the first function preceding all those for the second, etc., this partition can also be created in time $O(|S|)$. In addition, we can assume that each set S_d is ordered by its first components.

Then, to determine all the arcs leaving the vertex (d, d') in $G_P(F)$, all we need do is merge the two ordered sets S_d and $S_{d'}$ (in linear time [1]), and read off the results. If (f, d, e) occurs adjacent to (f, d', e') , we know there is an arc from (d, d') to (e, e') that we can label with f . If the first is missing, there is an arc from (d, d') to (d, e') labelled by f ; if the second is missing, there is an arc from (d, d') to (e, d') labelled by f . If both are missing we skip function f entirely, since it is the identity on d and d' , which only results in a self-loop in $G_P(F)$. (The removal of loops from $G_P(F)$ does not interfere with construction of $G_C(F)$ from $G_P(F)$.)

The total time for constructing and labelling all arcs out of the vertex (d, d') in $G_P(F)$ is thus $O(|S_d| + |S_{d'}|)$. Summing over all pairs (d, d') , we obtain the claimed overall time bound of $O(|D| \cdot |S|)$. Since, as we observed in the previous section, $G_C(F)$ can be constructed from $G_P(F)$ in time proportional to $|D|^3$ plus the size of $G_C(F)$, this means that we can construct $G_C(F)$ within time $O(|D|(|D|^2 + |S|))$, and this becomes the bound on the overall process of finding a function $f \in F^*$ with $|f(D)| = F_{\min}$. The savings in time is the same as the original savings in space for storing F : we replace a factor of $|D| \cdot |F|$ by $|S|$.

In the database application, where $|S| = O(|D|^2)$, this reduces the overall running time to $O(|D|^3)$, which is comparable to the time claimed in [2], although the running time there was expressed in terms of the internal structure of the elements of D , and F was only generated implicitly. If F were generated explicitly in that application, and then the current algorithm applied, the resulting hybrid procedure, although quite different from the original one in that paper, would have precisely the same running time. It would also be more general, since, by relying less on the internal structure of the elements of D (the conjuncts), the class of queries it can handle expands to all those queries whose self-homomorphisms obey restriction (R) above and can be constructed efficiently.

Another restriction that holds in [2] says that every element of F^* is either in F or else the composition of two functions in F . This imposes further structural restrictions on $G_P(F)$, but we have been unable to use it to obtain an improvement in the overall running time of the algorithm. We leave such improvements, both in the restricted case and the general one, to ambitious readers.

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] D. S. JOHNSON AND A. KLUG, *Optimizing conjunctive queries that contain untyped variables*, SIAM J. Comput., 12 (1983), pp. 616-640.
- [3] D. KOZEN, *Lower bounds for natural proof systems*, Proc. 18th Ann. Symp. on Foundations of Computer Science, IEEE Computer Society, Long Beach, CA, 1978, pp. 254-266.

SHIFT-REGISTER SYNTHESIS (MODULO m)*

J. A. REEDS† AND N. J. A. SLOANE†

Abstract. The Berlekamp–Massey algorithm takes a sequence of elements from a field and finds the shortest linear recurrence (or linear feedback shift register) that can generate the sequence. In this paper we extend the algorithm to the case when the elements of the sequence are integers modulo m , where m is an arbitrary integer with known prime decomposition.

Key words. Berlekamp–Massey algorithm, shift-register synthesis, linear recurrences

1. Introduction. The Berlekamp–Massey algorithm used in decoding BCH codes also solves the following problem: given a sequence S_0, S_1, \dots, S_{n-1} of elements from a field, find the shortest linear recurrence (or linear feedback shift register) that will generate the sequence [1, Chapt. 7], [21]. The algorithm can be used to decode other codes [18]–[20], [23], [27], [32], is related to the Euclidean algorithm and the computation of Padé approximations [6], [7], [22], [34], and has been extensively studied [2], [8]–[10], [13], [26], [33], [35]. W. F. Lunnon, in an unpublished manuscript [17], has pointed out that a version of the quotient-difference algorithm [12], [14], [15] can be used to find the shortest linear recurrence which generates a given sequence of complex numbers. This is not as efficient as the Berlekamp–Massey algorithm, although it has the advantage of being easier to remember, at least in its simplest form. Games and Chan [11] give a fast algorithm for finding the shortest linear recurrence in the special case of a binary sequence of period 2^k . Nevertheless, in spite of this extensive literature, it appears that until now no one has extended the Berlekamp–Massey algorithm so as to find the shortest linear recurrence that will generate a given sequence of numbers modulo m , where m is an arbitrary (but known) integer. (For the case when m is unknown, see [24, 25].) In this note we describe such an algorithm. The original algorithm [1], [21] fails in this case because not all numbers have inverses modulo m , and other versions—such as those involving the Euclidean algorithm [32]—fail because certain polynomial rings are no longer principal ideal domains.

There are several obvious applications of the new algorithm, for example in certifying random number generators (see the discussion in [29]), or in decoding BCH codes defined over the integers modulo m (see [3], [4], [28], [30], [31]).

Section 2 establishes our notation for linear recurrences, and § 3 uses the Chinese remainder theorem to reduce the problem to the case when m is a prime power. The algorithm itself is given in § 4 and its justification in § 5. The final section contains an example.

Gustavson [13] has analyzed the complexity of the Berlekamp–Massey algorithm, and the complexity of our algorithm is essentially the same. If the modulus m is fixed, $O(n^2)$ steps are required to synthesize a sequence of length n . Changing from a sequence modulo p to a sequence modulo p^e increases the number of steps by a factor of e .

2. Linear recurrences and linear feedback shift registers. Our notation follows Massey's description [21] of the Berlekamp–Massey algorithm, and some familiarity with that paper would be helpful (although not essential) in reading this one. Let R be a commutative ring containing the unit element 1, and let R^* denote the set of all

* Received by the editors May 17, 1983, and in revised form March 13, 1984.

† Mathematics and Statistics Research Center, Bell Laboratories, Murray Hill, New Jersey 07974.

units (or invertible elements) of R [16]. The sequence S_0, S_1, \dots, S_{n-1} , where all $S_i \in R$, is said to obey a *linear recurrence of length l* , or to be generated by a *linear feedback shift-register of length l* , if there are elements $a_0 = 1, a_1, \dots, a_l \in R$ such that

$$(1) \quad \sum_{i=0}^l a_i S_{j-i} = 0 \quad \text{for } j = l, \dots, n-1.$$

It is convenient to express (1) in terms of polynomials from $R[x]$. Let $a(x) = a_0 + a_1x + \dots + a_lx^l, S(x) = S_0 + S_1x + \dots + S_{n-1}x^{n-1}$. Then (1) is equivalent to

$$(2) \quad \begin{aligned} S(x)a(x) &\equiv b(x) \pmod{x^n}, \\ a(0) &= 1, \end{aligned}$$

for some polynomial $b(x) \in R[x]$ of degree $\leq l-1$. Thus the length of the recurrence or shift register is $l \geq \max \{\deg a(x), 1 + \deg b(x)\}$, and in fact there is no loss of generality in assuming that $l = \max \{\deg a(x), 1 + \deg b(x)\}$. We write $A = (a(x), b(x))$ and define $L(A) = \max \{\deg a(x), 1 + \deg b(x)\}$. By convention $\deg(0) = -\infty$.

3. The Chinese remainder theorem. In our problem R is the ring $\mathbf{Z}_m = \mathbf{Z}/m\mathbf{Z}$ of integers modulo m , where $m \geq 2$ is a given integer whose factorization is known. We wish to find an algorithm which, when presented with a sequence S_0, \dots, S_{n-1} , all $S_i \in \mathbf{Z}_m$, will find a linear recurrence $A = (a(x), b(x))$ that generates the sequence, i.e., satisfies (2), and has minimal length $l = L(A)$. By the Chinese remainder theorem [16] it is enough to solve the problem for the case when the modulus is a prime power. For suppose $m = \prod_i p_i^{e_i}, e_i \geq 1$, where the p_i are distinct primes, and assume that for each i we have found a minimal length recurrence $(a^{(i)}(x), b^{(i)}(x))$, of length l_i say, that generates the sequence S_0, \dots, S_{n-1} modulo $p_i^{e_i}$. By the Chinese remainder theorem we can find a pair $(a(x), b(x))$ with

$$a(x) \equiv a^{(i)}(x), \quad b(x) \equiv b^{(i)}(x) \pmod{p_i^{e_i}}$$

for all i , of length $l = \max \{l_i\}$, and it is straightforward to show that $(a(x), b(x))$ is a minimal length recurrence generating S_0, \dots, S_{n-1} modulo m .

4. The algorithm. Given $S_0, S_1, \dots, S_{n-1} \in R$, where $R = \mathbf{Z}_p^e, p = \text{prime}, e \geq 1$, we wish to find a linear recurrence $A = (a(x), b(x))$ of minimal length $l = L(A)$ satisfying (2). The key idea is to consider not just (2) but the following more general problem. For all $\eta = 0, 1, \dots, e-1$, find pairs $A_\eta = (a_\eta(x), b_\eta(x))$ such that

$$(3) \quad \begin{aligned} S(x)a_\eta(x) &\equiv b_\eta(x) \pmod{x^n}, \\ a_\eta(0) &= p^\eta, \end{aligned}$$

and $L(A_\eta) = l_\eta$ is minimized.

Our algorithm is an iterative procedure that, for all $0 \leq k \leq n, 0 \leq \eta < e$, calculates pairs

$$A_\eta^{(k)} = (a_\eta^{(k)}(x), b_\eta^{(k)}(x))$$

satisfying

$$\begin{aligned} S(x)a_\eta^{(k)}(x) &\equiv b_\eta^{(k)}(x) \pmod{x^k}, \\ a_\eta^{(k)}(0) &= p^\eta \end{aligned}$$

and minimizing $L(A_\eta^{(k)})$. Let $p^{u_{\eta k}} (0 \leq u_{\eta k} \leq e)$ be the highest power of p dividing the coefficient of x^k in

$$S(x)a_\eta^{(k)}(x) - b_\eta^{(k)}(x).$$

(If this coefficient is zero we take $u_{\eta k} = e$.) Then at the k th step in the iteration, the following property holds for all $0 \leq r < k$:

(P_r) For all $0 \leq g < e$, either

$$(4) \quad L(A_g^{(r+1)}) = L(A_g^{(r)})$$

or else there exists an $h = f(g, r)$ with

$$(5) \quad g + u_{hr} < e,$$

$$(6) \quad L(A_g^{(r+1)}) = r + 1 - L(A_h^{(r)}),$$

$$(7) \quad L(A_g^{(r+1)}) > L(A_g^{(r)}).$$

(This property is the analogue for our problem of the conditions that Berlekamp gives in [1, p. 183, eq. (7.314)] and Massey gives in [21, p. 123, eqs. (11)-(13)].) Given this data, our algorithm calculates $A_\eta^{(k+1)}$ and $f(\eta, k)$, $0 \leq \eta < e$, such that P_k holds. The quantities $L(A_\eta^{(k)})$ also obey the inequalities

$$(8) \quad L(A_{\eta+1}^{(k)}) \leq L(A_\eta^{(k)}) \leq L(A_\eta^{(k+1)}).$$

We can now state the algorithm. (A more compact version, suitable for computer implementation, is given at the end of this section.)

The algorithm (theorem-proving version). Given S_0, S_1, \dots, S_{n-1} , all $S_i \in R = \mathbb{Z}_p^e$, we wish to find a pair $A = (a(x), b(x))$ such that $S(x)a(x) \equiv b(x) \pmod{x^n}$, $a(0) = 1$, and the length $l = L(A) = \max \{\deg a(x), 1 + \deg b(x)\}$ is minimized.

Step 0. We start the algorithm with $k = 0$, and for each $\eta = 0, 1, \dots, e - 1$ define

$$a_\eta^{(0)}(x) = p^\eta, \quad b_\eta^{(0)}(x) = 0, \quad a_\eta^{(1)}(x) = p^\eta, \quad b_\eta^{(1)}(x) = p^\eta S_0,$$

and $A_\eta^{(i)} = (a_\eta^{(i)}(x), b_\eta^{(i)}(x))$, for $i = 0, 1$. Let $S_0 = \delta p^\epsilon$ for $\delta \in R^*$, $0 \leq \epsilon \leq e$ (if $S_0 = 0$ set $\delta = 1$ and $\epsilon = e$). Then $L(A_\eta^{(0)}) = 0$, and $L(A_\eta^{(1)}) = 1$ if $\eta + \epsilon < e$ or $= 0$ if $\eta + \epsilon \geq e$. We also define

$$\begin{aligned} \theta_{\eta 0} &= \delta, & u_{\eta 0} &= \eta + \epsilon & \text{if } \eta + \epsilon < e, \\ \theta_{\eta 0} &= 1, & u_{\eta 0} &= e & \text{if } \eta + \epsilon \geq e \end{aligned}$$

(these values are consistent with (9) below). Finally we set $f(\eta, 0) = 0$ for all η .

The following step is carried out for each $k = 1, 2, \dots, n - 1$.

Step k. This produces $A_\eta^{(k+1)}$. For each $\eta = 0, 1, \dots, e - 1$ we perform the following calculations. Define $\theta_{\eta k} \in R^*$ and $0 \leq u_{\eta k} \leq e$ by

$$(9) \quad S(x)a_\eta^{(k)}(x) \equiv b_\eta^{(k)}(x) + \theta_{\eta k} p^{u_{\eta k}} x^k \pmod{x^{k+1}}.$$

($\theta_{\eta k} p^{u_{\eta k}}$ is the *current discrepancy* in the notation of [21].)

Case I. If $u_{\eta k} = e$ set $A_\eta^{(k+1)} = A_\eta^{(k)}$.

Case II. If $u_{\eta k} < e$ define

$$(10) \quad g = e - 1 - u_{\eta k},$$

so that $0 \leq g < e$, and put

$$(11) \quad f(\eta, k) = g.$$

There are now two subcases.

Case IIa. If $L(A_g^{(k)}) = 0$ we set

$$(12) \quad A_\eta^{(k+1)} = A_\eta^{(k)} + (0, \theta_{\eta k} p^{u_{\eta k}} x^k).$$

Case IIb. If $L(A_g^{(k)}) > 0$ then for some $0 \leq r < k$ we have

$$(13) \quad L(A_g^{(r)}) < L(A_g^{(r+1)}) = L(A_g^{(k)}).$$

r is the time of the most recent length change in the sequence $L(A_g^{(0)}), L(A_g^{(1)}), \dots$. From (5), (6) and (13) it follows that

$$(14) \quad L(A_g^{(k)}) = L(A_g^{(r+1)}) = r + 1 - L(A_h^{(r)}),$$

where $h = f(g, r)$ and

$$(15) \quad g + u_{hr} < e.$$

From (10) and (15), $u_{hr} \leq u_{\eta k}$. Thus the power of p from the past can be used to annihilate the power of p in the current discrepancy, and we define

$$(16) \quad a_\eta^{(k+1)}(x) = a_\eta^{(k)}(x) - \theta_{\eta k} \theta_{hr}^{-1} p^{u_{\eta k} - u_{hr}} x^{k-r} a_h^{(r)}(x),$$

$$(17) \quad b_\eta^{(k+1)}(x) = b_\eta^{(k)}(x) - \theta_{\eta k} \theta_{hr}^{-1} p^{u_{\eta k} - u_{hr}} x^{k-r} b_h^{(r)}(x)$$

and $A_\eta^{(k+1)} = (a_\eta^{(k+1)}(x), b_\eta^{(k+1)}(x))$. Then

$$S(x) a_\eta^{(k+1)}(x) \equiv b_\eta^{(k+1)}(x) \pmod{x^{k+1}},$$

$$a_\eta^{(k+1)}(0) = p^n.$$

This concludes Step k .

At the end of Step $n-1$ the algorithm terminates and the desired pair $A = (a(x), b(x))$ is given by $A_0^{(n)} = (a_0^{(n)}(x), b_0^{(n)}(x))$.

The initial values of $A_\eta^{(k)}$ are as follows. Let $S_i = S_i^* p^{\varepsilon_i}$, where $S_i^* \in R^*$, $0 \leq \varepsilon_i \leq e$, $i = 0$ or 1 . Then

$$(18) \quad A_\eta^{(0)} = (p^\eta, 0), \quad L(A_\eta^{(0)}) = 0,$$

$$(19) \quad A_\eta^{(1)} = (p^\eta, p^\eta S_0), \quad L(A_\eta^{(1)}) = \begin{cases} 1 & \text{for } \eta \leq e - \varepsilon_0 - 1, \\ 0 & \text{for } \eta \geq e - \varepsilon_0, \end{cases}$$

and

$$(20) \quad A_\eta^{(2)} = \begin{cases} (p^\eta, p^\eta S_0) & \text{for } e - \varepsilon_1 \leq \eta, \\ (p^\eta, p^\eta (S_0 + S_1 x)) & \text{for } 0 \leq \eta \leq \varepsilon_0 - \varepsilon_1 - 1, \\ (p^\eta - p^{\eta - \varepsilon_0 + \varepsilon_1} (S_0^*)^{-1} S_1^* x, S_0) & \text{for } \varepsilon_0 - \varepsilon_1 \leq \eta \leq e - \varepsilon_1 - 1. \end{cases}$$

The algorithm as presented above calculates and saves various intermediate quantities not needed in subsequent steps. The following is a more streamlined version that needs to remember only $O(e)$ intermediate quantities, some of which are polynomials.

The algorithm (computer-implementation version). Given S_0, S_1, \dots, S_{n-1} , all $S_i \in R = \mathbf{Z}_p^e$, this algorithm produces a pair $A = (a(x), b(x))$ such that $S(x)a(x) \equiv b(x) \pmod{x^n}$, $a(0) = 1$, and the length $l = L(A) = \max \{\deg a(x), 1 + \deg b(x)\}$ is minimized.

Step 0. For each $\eta = 0, 1, \dots, e-1$ set

$$a_\eta(x) = p^\eta, \quad b_\eta(x) = 0, \quad a_\eta^{\text{new}}(x) = p^\eta, \quad b_\eta^{\text{new}}(x) = p^\eta S_0,$$

and find θ_η and u_η , $\theta_\eta \in R^*$, $0 \leq u_\eta \leq e$, such that

$$S(x)a_\eta(x) - b_\eta(x) \equiv \theta_\eta p^{u_\eta} \pmod{x}.$$

The following step is carried out for each $k = 1, 2, \dots, n - 1$.

Step k. There are three parts.

First, for each $g = 0, 1, \dots, e - 1$, if $L(a_g^{\text{new}}(x), b_g^{\text{new}}(x)) > L(a_g(x), b_g(x))$, set

$$\begin{aligned} a_g^{\text{old}}(x) &= a_h(x), & u_g^{\text{old}} &= u_h, \\ b_g^{\text{old}}(x) &= b_h(x), & r_g &= k - 1, \\ \theta_g^{\text{old}} &= \theta_h, \end{aligned}$$

where $h = e - 1 - u_g$.

Second, for each $\eta = 0, 1, \dots, e - 1$, set $a_\eta(x) = a_\eta^{\text{new}}(x)$ and $b_\eta(x) = b_\eta^{\text{new}}(x)$.

Third, for each $\eta = 0, 1, \dots, e - 1$, find θ_η and u_η , $\theta_\eta \in R^*$, $0 \leq u_\eta \leq e$, such that

$$S(x)a_\eta(x) - b_\eta(x) \equiv \theta_\eta p^{u_\eta} x^k \pmod{x^{k+1}}.$$

Set $g = e - 1 - u_\eta$. Then (I) if $u_\eta = e$ set

$$a_\eta^{\text{new}}(x) = a_\eta(x), \quad b_\eta^{\text{new}}(x) = b_\eta(x);$$

or (IIa) if $u_\eta \neq e$ and $L(a_g(x), b_g(x)) = 0$, set

$$a_\eta^{\text{new}}(x) = a_\eta(x), \quad b_\eta^{\text{new}}(x) = b_\eta(x) + \theta_\eta p^{u_\eta} x^k;$$

or (IIb) if $u_\eta \neq e$ and $L(a_g(x), b_g(x)) \neq 0$, set

$$\begin{aligned} a_\eta^{\text{new}}(x) &= a_\eta(x) - \theta_\eta (\theta_g^{\text{old}})^{-1} p^{u_\eta - u_g^{\text{old}}} x^{k - r_g} a_g^{\text{old}}(x), \\ b_\eta^{\text{new}}(x) &= b_\eta(x) - \theta_\eta (\theta_g^{\text{old}})^{-1} p^{u_\eta - u_g^{\text{old}}} x^{k - r_g} b_g^{\text{old}}(x). \end{aligned}$$

At the end of Step $n - 1$ the algorithm terminates and the desired pair $(a(x), b(x))$ is given by $(a_0^{\text{new}}(x), b_0^{\text{new}}(x))$.

The variables in this version of the algorithm are related to those in the original version as follows. At the conclusion of Step k we have, for each η ,

$$\begin{aligned} (a_\eta(x), b_\eta(x)) &= (a_\eta^{(k)}(x), b_\eta^{(k)}(x)), \\ (a_\eta^{\text{new}}(x), b_\eta^{\text{new}}(x)) &= (a_\eta^{(k+1)}(x), b_\eta^{(k+1)}(x)), \\ \theta_\eta &= \theta_{\eta k}, \quad u_\eta = u_{\eta k}. \end{aligned}$$

If

$$g = g_\eta = e - 1 - u_\eta = e - 1 - u_{\eta k}$$

then

$$\begin{aligned} r_\eta &= \max \{r: r < k \text{ and } L(a_g^{(r)}(x), b_g^{(r)}(x)) < L(a_g^{(r+1)}(x), b_g^{(r+1)}(x))\}, \\ \theta_g^{\text{old}} &= \theta_{hr_\eta}, \\ u_g^{\text{old}} &= u_{hr_\eta}, \end{aligned}$$

where $h = f(g, r_\eta)$.

5. Proof of correctness. It is convenient to denote the set of all pairs $(a(x), b(x))$ satisfying

$$(21) \quad \begin{aligned} S(x)a(x) &\equiv b(x) \pmod{x^k}, \\ a(0) &= p^\eta \end{aligned}$$

by $\mathcal{E}_\eta^{(k)}$, and to let

$$\mathcal{B}_\eta^{(k)} = \{(a(x), b(x)): S(x)a(x) \equiv b(x) + \theta p^\eta x^k \pmod{x^{k+1}} \text{ for some } \theta \in R^*\}$$

for $0 \leq \eta \leq e$. Note that if $(a(x), b(x)) \in \mathcal{E}_\eta^{(k)}$ then $(pa(x), pb(x)) \in \mathcal{E}_{\eta+1}^{(k)}$, and $(a(x), b(x))$ is in $\mathcal{B}_u^{(k)}$ for some u , $0 \leq u \leq e$; while if $u = e$ then $(a(x), b(x)) \in \mathcal{E}_\eta^{(k+1)}$. Furthermore, from (9),

$$(22) \quad A_\eta^{(k)} \in \mathcal{E}_\eta^{(k)} \cap \mathcal{B}_{u_{\eta k}}^{(k)}.$$

The proof that the algorithm works is based on two lemmas. The first is a generalization of [21, Lemma 1] and part of [1, Thm. 7.42].

LEMMA 1. *If $(a(x), b(x)) \in \mathcal{E}_\eta^{(k)}$ and $(c(x), d(x)) \in \mathcal{B}_u^{(k-1)}$, where $\eta + u < e$, then*

$$(23) \quad L(a(x), b(x)) + L(c(x), d(x)) \geq k.$$

Proof. Working modulo x^k we have

$$S(x)a(x) \equiv b(x), \quad S(x)c(x) \equiv d(x) + \theta p^u x^{k-1},$$

for $\theta \in R^*$, so

$$(24) \quad b(x)c(x) - a(x)d(x) \equiv \theta p^u x^{k-1} a(x) \equiv \theta p^u x^{k-1} a(0) = \theta p^{\eta+u} x^{k-1},$$

which does not vanish. Therefore the degree of the left-hand side of (24) is at least $k-1$. But

$$(25) \quad \begin{aligned} \deg(b(x)c(x) - a(x)d(x)) &\leq \max\{\deg(b(x)c(x)), \deg(a(x)d(x))\} \\ &\leq L(a(x), b(x)) + L(c(x), d(x)) - 1, \end{aligned}$$

as required.

A pair $(a(x), b(x))$ is said to have *minimal length* in $\mathcal{E}_\eta^{(k)}$ if $(a(x), b(x)) \in \mathcal{E}_\eta^{(k)}$ and if $L(a(x), b(x)) \leq L(a'(x), b'(x))$ holds for all $(a'(x), b'(x)) \in \mathcal{E}_\eta^{(k)}$. The second lemma shows how Lemma 1 can be used to verify that a particular pair has minimal length.

LEMMA 2. *Suppose in addition to the hypotheses of Lemma 1 that equality holds in (23). Then $(a(x), b(x))$ has minimal length in $\mathcal{E}_\eta^{(k)}$.*

This is an immediate consequence of Lemma 1. We can now justify the correctness of the algorithm.

THEOREM 1. *For all $k = 0, 1, \dots, n$ and $\eta = 0, 1, \dots, e-1$, $A_\eta^{(k)}$ has minimal length in $\mathcal{E}_\eta^{(k)}$.*

Proof. The proof is by induction on k . The induction hypothesis is that, when beginning Step k ,

properties P_0, P_1, \dots, P_{k-1} hold (see (4)-(7)); and

$A_g^{(r)}$ has minimal length in $\mathcal{E}_g^{(r)}$ for $0 \leq r \leq k$, $0 \leq g < e$.

In Step k we compute $u_{\eta k}$ etc. from (9) and form $A_\eta^{(k+1)}$. To establish the induction we must show that, at the end of Step k , property P_k holds, i.e.,

(P_k) For all $0 \leq \eta < e$, either

$$(26) \quad L(A_\eta^{(k+1)}) = L(A_\eta^{(k)})$$

or else

$$(27) \quad \eta + u_{gk} < e,$$

$$(28) \quad L(A_\eta^{(k+1)}) = k + 1 - L(A_g^{(k)}),$$

$$(29) \quad L(A_\eta^{(k+1)}) > L(A_\eta^{(k)});$$

and that

$A_\eta^{(k+1)}$ has minimal length in $\mathcal{E}_\eta^{(k+1)}$ for $0 \leq \eta < e$.

The initialization, proving P_0 and the minimality of $A_\eta^{(0)}$ and $A_\eta^{(1)}$, is straightforward and we omit the details.

Suppose we are in Step k , and Case I obtains. Then (26) holds, and $A_\eta^{(k+1)}$ has minimal length by induction.

Suppose we have Case IIa. We first establish P_k . We may assume (26) does not hold. Then

$$L(A_g^{(k)}) = 0, \quad A_g^{(k)} = (p^g, 0),$$

and, from (9),

$$S(x)p^g \equiv \theta_{gk} p^{u_{gk}} x^k \pmod{x^{k+1}}.$$

This implies that $p^{e-g} = p^{1+u_{\eta k}}$ divides each of S_0, \dots, S_{k-1} and $S_k = \theta p^{u_{gk}-g}$ for some $\theta \in R^*$. Let $S_i = p^{1+u_{\eta k}} S_i^*$ for $i < k$. Using (9) again, and remembering that $a_\eta^{(k)}(0) = p^\eta$, we have

$$(30) \quad \begin{aligned} & \{ p^{1+u_{\eta k}} (S_0^* + \dots + S_{k-1}^* x^{k-1}) + \theta p^{u_{gk}-g} x^k \} \cdot \{ p^\eta + \dots \} \\ & \equiv b_\eta^{(k)}(x) + \theta_{\eta k} p^{u_{\eta k}} x^k \pmod{x^{k+1}}. \end{aligned}$$

Since $L(A_\eta^{(k+1)}) = k+1 > L(A_\eta^{(k)})$,

$$\deg b_\eta^{(k)}(x) \leq k-1.$$

Equating coefficients of x^k in (30), and using (10), we obtain

$$\alpha p^{1+u_{\eta k}} + \theta p^{u_{gk} + \eta - e + 1 + u_{\eta k}} = \theta_{\eta k} p^{u_{\eta k}}$$

for some $\alpha \in R$. Since $\theta_{\eta k}$ is a unit, it follows that p does not divide $p^{u_{gk} + \eta - e + 1}$, i.e., $\eta + u_{gk} < e$, which is (27).

Next we show that (28) follows from (27). In fact we shall show that (27) implies

$$(31) \quad L(A_\eta^{(k+1)}) = \max \{ L(A_\eta^{(k)}), k+1 - L(A_g^{(k)}) \}.$$

From (16), (17), (14) we have

$$\begin{aligned} L(A_\eta^{(k+1)}) & \leq \max \{ L(A_\eta^{(k)}), k-r + L(A_h^{(r)}) \} \\ & = \max \{ L(A_\eta^{(k)}), k+1 - L(A_g^{(k)}) \}. \end{aligned}$$

But the reverse inequality follows from Lemma 1, using (22), and establishes (31). The minimality of $A_\eta^{(k+1)}$ now follows from (27), (28) and Lemma 2.

Finally, suppose Case IIb obtains. To establish P_k we may assume (26) does not hold, and so, from (16), (17),

$$k-r + L(A_h^{(r)}) > L(A_\eta^{(k)}),$$

i.e.

$$(32) \quad k+1 > L(A_\eta^{(k)}) + L(A_g^{(k)}),$$

using (14). Consider the polynomial

$$(33) \quad q(x) = a_\eta^{(k)}(x) \{ S(x) a_g^{(k)}(x) - b_g^{(k)}(x) \} - a_g^{(k)}(x) \{ S(x) a_\eta^{(k)}(x) - b_\eta^{(k)}(x) \}$$

$$(34) \quad = a_g^{(k)}(x) b_\eta^{(k)}(x) - a_\eta^{(k)}(x) b_g^{(k)}(x).$$

Then, just as in (25),

$$\deg q(x) \leq L(A_\eta^{(k)}) + L(A_g^{(k)}) - 1 < k, \quad \text{by (32)}.$$

On the other hand, from (33),

$$(35) \quad q(x) = (p^\eta + \dots)(\theta_{gk} p^{u_{gk}} x^k + \dots) - (p^g + \dots)(\theta_{\eta k} p^{u_{\eta k}} x^k + \dots),$$

containing only terms of degree $\geq k$. Therefore $q(x)$ is identically zero. But the coefficient of x^k in (35) is

$$\theta_{gk} p^{\eta+u_{gk}} - \theta_{\eta k} p^{g+u_{\eta k}}$$

and so

$$(36) \quad \eta + u_{gk} = g + u_{\eta k}.$$

Equation (27) now follows from (10). The remainder of the proof is the same as in Case IIa.

6. An example. As an example we find the shortest linear recurrence that can produce the sequence $S_0 = 6, S_1 = 3, S_2 = 1, S_3 = 5, S_4 = 6$ modulo 9. The computation is displayed in a pair of tables indexed by $(k, \eta), k = 0, 1, \dots, 5$ and $\eta = 0, 1$. The first table shows the pairs $(a_\eta^{(k)}(x), b_\eta^{(k)}(x))$. The second table shows the quintuples $(l, u_{\eta k}, \theta_{\eta k}, f(\eta, k), \text{Case})$, where $l = L(a_\eta^{(k)}(x), b_\eta^{(k)}(x))$, and Case is one of I, IIa, or IIb, indicating which case holds in the computation of $(a_\eta^{(k+1)}(x), b_\eta^{(k+1)}(x))$. From the last line of Table 1, the shortest recurrence satisfied by the sequence is

$$S_n + 4S_{n-1} + 7S_{n-2} + S_{n-3} = 0, \quad n \geq 3.$$

TABLE 1
 $(a_\eta^{(k)}(x), b_\eta^{(k)}(x))$

| | $\eta = 0$ | $\eta = 1$ |
|---------|---|--|
| $k = 0$ | (1, 0) | (3, 0) |
| 1 | (1, 6) | (3, 0) |
| 2 | (1 + 4x, 6) | (3, 0) |
| 3 | (1 + 4x, 6 + 4x ²) | (3 + 4x ² , 0) |
| 4 | (1 + 4x, 6 + 4x ²) | (3 + 4x ² , 0) |
| 5 | (1 + 4x + 7x ² + x ³ , 6 + x ²) | (3 + 3x ² + 5x ³ , 3x ²) |

TABLE 2
 $(l, u_{\eta k}, \theta_{\eta k}, f(\eta, k), \text{case})$

| | $\eta = 0$ | $\eta = 1$ |
|---------|-------------------|-------------------|
| $k = 0$ | (0, 1, 2, 0, *) | (0, 2, 1, 0, *) |
| 1 | (1, 1, 1, 0, IIb) | (0, 2, 1, *, I) |
| 2 | (1, 0, 4, 1, IIa) | (0, 1, 1, 0, IIb) |
| 3 | (3, 2, 1, *, 1) | (2, 2, 1, *, 1) |
| 4 | (3, 0, 8, 1, IIb) | (2, 0, 4, 1, IIb) |
| 5 | (3, *, *, *, *) | (3, *, *, *, *) |

* Means "does not apply".

Acknowledgment. We thank the referee for some very helpful comments.

REFERENCES

[1] E. R. BERLEKAMP, *Algebraic Coding Theory*, McGraw-Hill, New York, 1968.
 [2] E. R. BERLEKAMP, E. M. FREDRICKSEN AND R. C. PROTO, *Minimum conditions for uniquely determining the generator of a linear sequence*, Utilitas Math., 5 (1974), pp. 305-315.

- [3] I. F. BLAKE, *Codes over certain rings*, Inform. Control, 20 (1972), pp. 396–404.
- [4] ———, *Codes over integer residue rings*, Inform. Control, 29 (1975), pp. 295–300.
- [5] W. A. BLANKINSHIP, *Solution of simultaneous linear diophantine equations*, Algorithm 288 in Collected Algorithms from ACM, 2 Vols., Association for Computing Machinery, New York, 1978.
- [6] R. P. BRENT, F. G. GUSTAVSON AND D. 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 nonnormal Padé tables*, SIAM J. Appl. Math., 39 (1980), pp. 106–118.
- [8] P. H. CHEN, *Multisequence linear shift register synthesis and its application to BCH decoding*, IEEE Trans. Commun., COM-24 (1976), pp. 438–440.
- [9] B. W. DICKINSON, M. MORF AND T. KAILATH, *A minimal realization algorithm for matrix sequences*, IEEE Trans. Automat. Control, AC-19 (1974), pp. 31–38.
- [10] J. F. DILLON AND R. A. MORRIS, *On a paper of Berlekamp, Fredricksen and Proto*, Utilitas Math., 5 (1974), pp. 317–321.
- [11] R. A. GAMES AND A. H. CHAN, *A fast algorithm for determining the complexity of a binary sequence with period 2^n* , IEEE Trans. Inform. Theory, IT-29 (1983), pp. 144–146.
- [12] W. B. GRAGG, *The Padé table and its relation to certain algorithms of numerical analysis*, SIAM Rev., 14 (1972), pp. 1–62.
- [13] F. G. GUSTAVSON, *Analysis of the Berlekamp–Massey feedback shift-register synthesis algorithm*, IBM J. Res. Dev., 20 (1976), pp. 204–212.
- [14] P. HENRICI, *Quotient-difference algorithms*, in Mathematical Methods for Digital Computers II, A. Ralston and H. Wilf, eds., John Wiley, New York, pp. 35–62.
- [15] W. B. JONES AND W. J. THRON, *Continued Fractions: Analytic Theory and Applications*, Addison-Wesley, Reading, MA, 1980.
- [16] S. LANG, *Algebra*, Addison-Wesley, Reading, MA, 1971.
- [17] W. F. LUNNON, *Linear recurring sequences over the complex numbers*, unpublished manuscript, 1970.
- [18] R. J. MCELIECE, *The Theory of Information and Coding*, Addison-Wesley, Reading, MA, 1977.
- [19] F. J. MACWILLIAMS AND N. J. A. SLOANE, *The Theory of Error-Correcting Codes*, North-Holland, Amsterdam, 1977.
- [20] D. M. MANDELBAUM, *A method for decoding of generalized Goppa codes*, IEEE Trans. Inform. Theory, IT-23 (1977), pp. 137–140.
- [21] J. L. MASSEY, *Shift-register synthesis and BCH decoding*, IEEE Trans. Inform. Theory, IT-15 (1969), pp. 122–127.
- [22] W. M. MILLS, *Continued fractions and linear recurrences*, Math. Comp., 29 (1975), pp. 173–180.
- [23] N. J. PATTERSON, *The algebraic decoding of Goppa codes*, IEEE Trans. Inform. Theory, IT-21 (1975), pp. 203–207.
- [24] J. B. PLUMSTEAD, *Inferring a sequence generated by a linear congruence*, in 23rd Annual Symposium on Foundations of Computer Science, IEEE Press, New York, 1982, pp. 153–159.
- [25] ———, *Inferring sequences produced by pseudo-random number generators*, Ph.D. dissertation, Computer Science Dept., Univ. California, Berkeley, 1983.
- [26] M. K. SAIN, *Minimal torsion spaces and the partial input/output problem*, Inform. Control, 29 (1975), pp. 103–124.
- [27] D. V. SARWATE, *On the complexity of decoding Goppa codes*, IEEE Trans. Inform. Theory, IT-23 (1977), pp. 515–516.
- [28] P. SHANKAR, *On BCH codes over arbitrary integer rings*, IEEE Trans. Inform. Theory, IT-25 (1979), pp. 480–483.
- [29] N. J. A. SLOANE, *Encrypting by random rotations*, pp. 71–128 of Cryptography (Proc. Workshop on Cryptography, Burg Feuerstein, Germany, March 29–April 2, 1982), Lecture Notes in Computer Science 149, Springer-Verlag, New York, 1983.
- [30] E. SPIEGEL, *Codes over Z_m* , Inform. Control, 35 (1977), pp. 48–51.
- [31] ———, *Codes over Z_m , revisited*, Inform. Control, 37 (1978), pp. 100–104.
- [32] Y. SUGIYAMA, M. KASAHARA, S. HIRASAWA AND T. NAMEKAWA, *A method for solving key equation for decoding Goppa codes*, Inform. Control, 27 (1975), pp. 87–99.
- [33] K. K. TZENG AND G. L. FENG, *Shift-register synthesis for t sequences*, preprint.
- [34] L. R. WELCH AND R. A. SCHOLTZ, *Continued fractions and Berlekamp's algorithm*, IEEE Trans. Inform. Theory, IT-25 (1979), pp. 19–27.
- [35] N. ZIERLER, *Linear recurring sequences and error-correcting codes*, in Error Correcting Codes, H. B. Mann, ed., John Wiley, New York, 1969, pp. 47–59.

ALPHABETIC MINIMAX TREES*

DAVID G. KIRKPATRICK† AND MARIA M. KLAWE‡

Abstract. This paper concerns the following problem. Given vertices v_1, \dots, v_n with weights w_1, \dots, w_n , construct a t -ary tree with leaves v_1, \dots, v_n in left to right order, such that if l_i denotes the length of the path from v_i to the root for each i , the maximum of $w_i + l_i$ is minimized. A linear algorithm is presented for the case where all the weights are integers, and this is used to obtain an $O(n \log n)$ algorithm for the case of general weights. Moreover it is shown that the minimax value obtained is bounded above by $2 + \log_t (\sum t^{(w_i)})$. This result has applications in the study of the effect of fan-out constraints in logical circuits.

Key words. optimal weighted tree, minimax tree, alphabetic tree, t -ary tree, linear algorithm, upper bound, fanout reduction in logical circuits

1. Introduction. A rooted tree is called a t -ary tree if every internal vertex has exactly t sons. It is easy to see that the number of leaves in a t -ary tree must equal $1 \pmod{t-1}$. In this paper we deal with the following problem, which we will refer to as the *alphabetic minimax problem*. Given vertices v_1, \dots, v_n with weights w_1, \dots, w_n for n an integer equal to $1 \pmod{t-1}$, construct a t -ary tree with leaves v_1, \dots, v_n in left to right order, such that if l_i denotes the length of the path from v_i to the root for each i , the maximum of $w_i + l_i$ is minimized. We call this value the minimax value, and denote it by $P(w_1, \dots, w_n)$. An equivalent version of the problem is to consider weighted t -ary trees in which the weight of each internal vertex is $1 +$ the maximum of the weights of its sons. In this formulation we are trying to construct a weighted t -ary tree with leaves v_1, \dots, v_n in left to right order, such that the weight of the root is minimized. As well as developing an efficient algorithm for constructing the optimal t -ary tree, we will establish tight upper bounds for the minimax value in terms of w_1, \dots, w_n .

One practical situation where this problem arises is where w_i represents the height of the vertex v_i , i.e., the length of the longest path leaving v_i , in an acyclic directed graph. In this case the minimax value obtained is the minimum possible height of a vertex v in the acyclic directed graph if v is the root of a t -ary tree which has v_1, \dots, v_n as its leaves in that order. Via this interpretation, our results on alphabetic minimax trees can be applied to obtain a bounding fan-out algorithm for circuits which preserves edge crossing constraints, while not increasing size and depth by more than constant multiplicative factors. We will describe this application in more detail towards the end of this section.

Similar problems concerning constructing t -ary trees which are optimum under various criteria have arisen in searching, sorting, coding theory and many other fields. Most of the research has concentrated on the problem of minimizing weighted path length, i.e., $\sum w_i l_i$, for both the unordered case where the leaves v_i are allowed to occur in any order, and the alphabetic case, where, as in our problem, the left to right order of the leaves is specified. In 1952 Huffman [10] gave a simple $O(n \log n)$ algorithm for constructing a t -ary tree which minimizes weighted path length in the unordered case. For the alphabetic case of weighted path length, in 1959 Gilbert and Moore [3] gave an $O(n^3)$ dynamic programming algorithm for constructing binary trees. A more

* Received by the editors July 6, 1982, and in revised form March 13, 1984.

† Department of Computer Science, University of British Columbia, Vancouver, British Columbia, Canada V6T 1W5.

‡ Department of Computer Science, IBM Research, San Jose, California 95193.

efficient dynamic programming algorithm using only $O(n^2)$ time was presented by Knuth [13] in 1971, and later that year Hu and Tucker [9] gave an algorithm which can be implemented to run in $O(n \log n)$ time.

The unordered version of the minimax problem was first considered by Golubic [4] in 1976, who showed that an adaptation of Huffman's algorithm constructs the optimum minimax tree. For the alphabetic minimax case, it is not hard to see that dynamic programming will work for any t , though it may be $O(n^3)$ at worst. For $t = 2$ or 3, the existence of an $O(n \log n)$ algorithm follows as a consequence of the extension of the Hu-Tucker algorithm to a more general setting by Hu, Kleitman and Tamaki [8].

In this paper we give an algorithm for the alphabetic minimax problem which runs in linear time when all of the weights are integers. We are also able to reduce the problem of general weights to at most $\log n$ integer problems, thus obtaining an $O(n \log n)$ algorithm for the alphabetic minimax problem for any t . Moreover, if desired, by solving k versions of the integer problem one can approximate the solution to a general alphabetic problem with error at most $1/2^{k-1}$. The integer version of the algorithm for binary trees is very simple, and easy to prove correct. In contrast, the Hu-Tucker algorithm, though not as complex as it appears at first glance, is quite unintuitive, and even after various simplifications [7], [2], [8], its proof remains far from obvious. Generalizing our integer algorithm to t -ary trees inevitably introduces some additional complexity in the proof of correctness, but conceptually, the simplicity of the binary case is retained.

In addition to the development of algorithms for the weighted path length and minimax problems, significant attention has been paid to establishing upper bounds on the optimal values in terms of the weights w_1, \dots, w_n . Upper bounds for the unordered and alphabetic weighted path length problems are due to Shannon [15] and Gilbert and Moore [3] respectively. A slightly different and tighter type of bound for the binary case of the latter problem was later obtained by Kleitman and Saks [12], as a consequence of determining which order of weights results in the greatest weighted path length for alphabetic binary trees. Golubic [4] proved that the optimal value for the unordered minimax problem is strictly less than $1 + \log_t (\sum t^{w_i})$ given that all the w_i are integers. A proof of the same bound for general weights can be found in Hoover, Klawe and Pippenger [6]. In this paper we prove that the alphabetic minimax value is strictly less than $2 + \log_t (\sum t^{w_i})$. Combining this with the Kraft inequality (i.e., the (easily proved) lower bound of $\log_t (\sum t^{w_i})$ for the unordered minimax value) shows that the difference between the alphabetic and unordered minimax value is at most 1. We are also able to prove a slightly tighter bound of the Kleitman and Saks variety.

In [6] Hoover, Klawe and Pippenger exploit Golubic's result for the unordered minimax problem to obtain a bounding fanout algorithm, referred to here as the HKP algorithm, with the following properties. Given any acyclic directed graph G with fan-in bounded by s , p inputs and q outputs, the HKP algorithm constructs a functionally equivalent graph G' , which has fan-out bounded by t as well as fan-in bounded by s , p inputs and q outputs. Moreover G' satisfies $\text{Size}(G') \leq (1 + (s-1)/(t-1)) \text{Size}(G) + (q-1)/(t-1)$ and $\text{Depth}(G') \leq (1 + \log_t s) \text{Depth}(G) + \log_t q$, where the size of a graph is defined to be its number of vertices and the depth is the length of its longest path from an input to an output. Viewing an acyclic directed graph as a model of a logic circuit, where vertices represent gates and edges represent the wires connecting gates, size corresponds to the cost of building a circuit and depth to the circuit's time delay. The motivation for the HKP algorithm is that since different technologies impose widely differing constraints on fan-out, it is of interest to know that any logic circuit

can be converted to one of bounded fan-out without multiplying either size or depth by more than a constant factor. Unfortunately, using the HKP algorithm, even if G is planar the new graph G' may be very far from planar. Since circuits, although not necessarily planar, have highly constrained edge crossings, it seems desirable to have a bounding fan-out algorithm which will preserve these constraints. The HKP algorithm uses the Golombic tree construction to replace vertices with too high fan-out by t -ary trees. By using our algorithm to construct alphabetic minimax trees instead, one can obtain an algorithm preserving crossing constraints. In particular, if G is planar then G' will be planar also. The bound on the increase in size is as before, and the bound on depth becomes $\text{Depth}(G') \leq (2 + \log_t s) \text{Depth}(G) + \log_t q$. Thus by preserving crossing constraints one pays a price of at most 1 in the multiplicative depth factor. Proofs of these bounds are entirely analogous to those for the HKP algorithm in [6].

In the application above, and in applications in general, the number n of weights will not necessarily be equal to $1 \pmod{t-1}$. By adding the appropriate number of dummy weights equal to $-\infty$ to the end of the list of weights, our algorithm can be applied without affecting the upper bound on the minimax value, but as we will illustrate in § 5 this technique does not necessarily construct the optimal tree when $n \not\equiv 1 \pmod{t-1}$. Let us define a sub- t -ary tree to be a rooted tree in which every vertex has at most t sons. Motivated by our results for t -ary trees, Coppersmith, Klawe and Pippenger [1] obtained similar results for sub- t -ary trees. In addition to giving a linear algorithm for integer weights, they prove that the sub- t -ary alphabetic minimax value is bounded by $1 + \log_t (2 \sum t^{(w_i)})$. In turn, inspired by some of their ideas, we were able to simplify our algorithm and the proof of our upper bound (Corollary 3.1.3) for integer weights.

In the next section we present local criteria which can be used to choose a set of t adjacent leaves to be siblings in a tree without increasing the minimax value. Section 3 exploits these criteria to obtain the integer algorithm and proof of the upper bound. In § 4 we extend these results to general weights, and in § 5 we present examples indicating that our results are best possible in various ways.

2. Right locally minimal t -tuples. In this section we introduce the concept of a right locally minimal t -tuple, and prove that if the weights of t adjacent leaf vertices form a right locally minimal t -tuple then there is some optimal minimax tree in which these vertices are siblings. In the next section we will use this result to obtain an algorithm which constructs optimal minimax trees for the case of integer weights. Right locally minimal t -tuples will also be essential in the proof of upper bounds on the minimax value.

In general it will be more convenient to deal directly with an ordered list of weights rather than an ordered list of weighted vertices. We remind the reader that for the entirety of this paper, except where explicitly noted, we assume that the number n of weights (or leaf vertices) in the initial list is equal to $1 \pmod{t-1}$. If $W = w_1, \dots, w_n$ is such a list of weights, we will use both $P(W)$ and $P(w_1, \dots, w_n)$ to denote the alphabetic minimax value for this list of weights. Thus, in other words, $P(W)$ is the minimum value of the root of a weighted tree, whose sequence of leaf weights is w_1, \dots, w_n . Moreover, we say that a tree T is an *optimal minimax tree for W* if the root of T has weight $P(W)$, and the sequence of leaf weights of T is W . We use the term *t -tuple* to refer to a sequence of t consecutive weights, and we speak of *merging* a t -tuple to form a new weight w to refer to the introduction of a new vertex v with weight w , such that the sons of v are the vertices whose weights form that t -tuple. Let $W = w_1, \dots, w_n$ be a list of weights. For $1 \leq i \leq n - t + 1$ we define $r \max(W, i) =$

$\max \{w_{i+j}; 0 \leq j \leq t-1\}$. With this definition we are now ready to define a *right locally minimal t -tuple* which we abbreviate by r.l.m. t -tuple. We say that $w_i, w_{i+1}, \dots, w_{i+t-1}$ are a r.l.m. t -tuple in W if the following three criteria are satisfied:

- (1) $1 \leq i \leq n-t+1$.
- (2) For each k such that $\max \{1, i-t+1\} \leq k \leq i-1$ we have $r \max(W, k) \geq r \max(W, i)$.
- (3) If $i \leq n-t$ then $w_{i+t} \geq 1+r \max(W, i)$.

Thus a r.l.m. t -tuple is just a t -tuple of weights whose collective maximum forms a local minimum among the t -tuples which intersect it, and moreover the collective maximum of every intersecting t -tuple to its right is greater by at least 1. Notice that if $n \geq t$ then it is always possible to find a r.l.m. t -tuple in a list of n integer weights, since it is easy to check that the rightmost t -tuple which has the minimum collective maximum over all t -tuples in the list must be a r.l.m. t -tuple. This is not true for general weights as can be seen by considering the list $1, \frac{1}{2}, \frac{1}{2}, 1$ for $t=2$.

The next theorem indicates the importance of right locally minimal t -tuples in constructing optimal minimax trees. We say that a subtree S of a tree T is maximal if the root of S is a son of the root of T .

THEOREM 2.1. *Let v_1, \dots, v_n be a list of vertices whose weights form the list $W = w_1, \dots, w_n$, and suppose w_i, \dots, w_{i+t-1} is a r.l.m. t -tuple in W . Then there exists an optimal minimax tree for W in which v_i, \dots, v_{i+t-1} are siblings.*

Proof. The proof is by induction on n . The case $n=t$ is obvious since then $i=1$ and clearly v_1, \dots, v_t must be the sons of the root of any minimax tree for W . Thus assume $n > t$ and that the theorem holds for any list of weights of size less than n . Let T be an optimal tree for W with leaves v_1, \dots, v_n , let S be the maximal subtree of T with v_i as a leaf, and let j be maximal such that v_j is a leaf of S . If $j \geq i+t-1$ then we can apply the inductive hypothesis to the leaves of S to obtain an optimal tree S' on the same leaves, such that v_i, \dots, v_{i+t-1} are siblings in S' . Now by replacing S by S' in T we obtain the desired optimal tree. Thus we may assume $j < i+t-1$. Let R be the maximal subtree containing v_{i+t-1} as a leaf, and let m be minimal such that v_m is a leaf of R .

We first consider the case that both S and R have at least t leaves. Let $h(S)$ [$h(R)$] be the maximum of the weights of the t rightmost [leftmost] leaves of S [R]. We will assume that $h(S) \geq h(R)$; a symmetric argument, with v_{i+t-1} playing the role of v_i , applies to the case $h(R) > h(S)$. We construct a new optimal tree T' as follows. First merge v_i, \dots, v_{i+t-1} as the sons of a new vertex v which takes v_i 's position in T . Now for $i+1 \leq k \leq m-1$ place v_{k+t-1} in v_k 's position in T . Finally, the subtree R is replaced by an optimal subtree R' with the same leaves as R except that the leftmost $t-1$ leaves have been removed. Figure 1 illustrates the construction of T' from T .

We now show that if u is a father of v_k in T for any k with $i \leq k \leq m-1$, then the weight of u in T' is no greater than the weight of u in T . To do this it suffices to show that $w(u)$, the weight of u in T , is at least one greater than the weight of the vertex which replaces v_k in T' . First suppose that u is the father of v_i . It is not hard to see that since w_i, \dots, w_{i+t-1} is a r.l.m. t -tuple, we must have $w(u) \geq 1+h(S)$. Now since $h(S) \geq h(R) \geq w_{i+t} \geq 1+r \max(W, i)$, and $w(v) = 1+r \max(W, i)$, this shows that $w(u) \geq 1+w(v)$. If u is the father of v_k , where $i+1 \leq k \leq j$, then again it is easy to see that $w(u) \geq 1+h(S) \geq 1+h(R) \geq 1+w_{k+t-1}$. Finally, if u is the father of v_k , where $j+1 \leq k \leq m-1$ then v_k must be a maximal subtree of T . In this case u must be the root of T and so obviously $w(u) \geq 1+w_{k+t-1}$.

If S [R] has less than t leaves then v_i [v_{i+t-1}] must be a son of the root of T . In this case it is even easier to see that T can be restructured so that v_i, \dots, v_{i+t-1} are siblings without increasing the weight of the root. \square

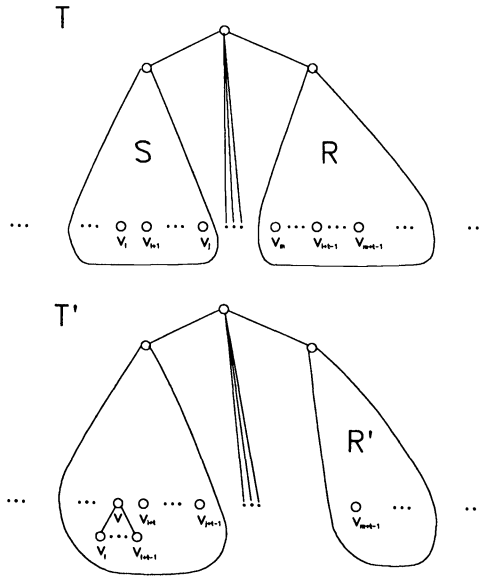


FIG. 1

3. Integer weights. In the previous section we observed that every list of at least t integer weights has at least one right locally minimal t -tuple. Thus for any list of integer weights W we can construct a tree as follows. Find a r.l.m. t -tuple w_i, \dots, w_{i+t-1} , replace this t -tuple by the weight $1 + r \max(W, i)$, and continue recursively. Theorem 2.1 guarantees that any tree constructed in this manner will be optimal for W . Since an integer list may have many different r.l.m. t -tuples, it is sometimes possible to construct many different optimal minimax trees by this technique, but in any case we see that r.l.m. t -tuples do yield an algorithm in the case of integer weights. By choosing the r.l.m. t -tuples appropriately we will see that the algorithm can be implemented to run in linear time. Before introducing our algorithm, we show that r.l.m. t -tuples can be used effectively to prove upper bounds on $P(W)$.

3.1. Upper bounds. For any weighted list $W = w_1, \dots, w_n$ where $n \geq t$, we define $G(W) = \sum_{1 \leq j \leq n-t+1} t^{r \max(W, j)}$.

LEMMA 3.1.1. *Let w_i, \dots, w_{i+t-1} be a r.l.m. t -tuple in a list $W = w_1, \dots, w_n$ with $n \geq 2t - 1$, and let X be the list obtained by deleting w_i, \dots, w_{i+t-1} from W and inserting (in w_i 's position) a new weight $w = 1 + r \max(W, i)$. Thus X is the list x_1, \dots, x_{n-t+1} , where $x_k = w_k$ for $1 \leq k \leq i - 1$, $x_i = w$, and $x_k = w_{k+t-1}$ for $i + 1 \leq k \leq n - t + 1$. Then $G(X) \leq G(W)$.*

Proof. We first observe that without loss of generality we may assume that $t \leq i \leq n - 2t + 2$, since if not, let $M = 1 + \max\{w_i : 1 \leq i \leq n\}$ and consider the lists W' and X' formed by adding $t - 1$ new weights of size M to each end of W and X respectively. Then w_i, \dots, w_{i+t-1} is a r.l.m. t -tuple in W' , and $G(X') \leq G(W')$ implies $G(X) \leq G(W)$ since $G(W') - G(W) = G(X') - G(X) = (2t - 2)t^M$. Since $t \leq i \leq n - 2t + 2$ it is straightforward to confirm that

$$G(X) = G(W) - \sum_{i-t+1 \leq k \leq i+t-1} t^{r \max(W, k)} + \sum_{i-t+1 \leq k \leq i} t^{r \max(X, k)}$$

As $w_{i+t} \geq w$, it follows that $r \max(X, i) = r \max(W, i + t - 1)$ and hence it suffices to show that

$$\sum_{i-t+1 \leq k \leq i+t-2} t^{r \max(W, k)} \geq \sum_{i-t+1 \leq k \leq i-1} t^{r \max(X, k)}$$

By the definition of a r.l.m. t -tuple, we know that $r \max(W, i) \leq \min \{r \max(W, k), r \max(W, k+t-1)\}$ for $i-t+1 \leq k \leq i-1$. Moreover, since $r \max(W, j) \geq w_{i+t} \geq w$ for $i+1 \leq j \leq i+t-1$, it follows that $r \max(X, k) = \max \{r \max(W, k), r \max(W, k+t-1)\}$ for $i-t+2 \leq k \leq i-1$. Hence we have $t^{r \max(W, k)} + t^{r \max(W, k+t-1)} \geq t^{r \max(X, k)} + t^{r \max(W, i)}$ for $i-t+2 \leq k \leq i-1$. Finally note that $r \max(X, i-t+1) = \max \{r \max(W, i-t+1), r \max(W, i)+1\}$ and hence $t^{r \max(W, i-t+1)} + t^{r \max(W, i)} \geq t^{r \max(X, i-t+1)} + t^{r \max(W, i)}$. Combining all this we have

$$\begin{aligned} \sum_{i-t+1 \leq k \leq i-t-2} t^{r \max(W, k)} &\geq t^{r \max(W, i-t+1)} + t^{r \max(W, i)} \\ &+ \sum_{i-t+2 \leq k \leq i-1} t^{r \max(X, k)} + (t-2)t^{r \max(W, i)} \\ &\geq \sum_{i-t+1 \leq k \leq i-1} t^{r \max(X, k)} \end{aligned}$$

as desired. \square

COROLLARY 3.1.2. *If W is a list of integer weights of length at least t then $t^{P(W)} \leq tG(W)$.*

Proof. Let T be a tree with leaf weight sequence W which is formed by repeatedly merging r.l.m. t -tuples, let r be the root of T , and let x_1, \dots, x_t be the sons of r in left to right order. Applying Lemma 3.1.1 sufficiently often shows that $t^{\max\{w(x_j): 1 \leq j \leq t\}} = G(w(x_1), \dots, w(x_t)) \leq G(W)$. The proof is completed by observing that $P(W) \leq w(r) = 1 + \max \{w(x_j): 1 \leq j \leq t\}$. \square

COROLLARY 3.1.3. *If w_1, \dots, w_n are integers then $P(w_1, \dots, w_n) < 2 + \log_t (\sum_{1 \leq j \leq n} t^{(w_j)})$.*

Proof. It is easy to see that $\sum_{1 \leq j \leq n-t+1} t^{r \max(W, j)} < t \sum_{1 \leq j \leq n} t^{(w_j)}$. Combining this with Corollary 3.1.2 and taking logarithms with respect to t yields the desired upper bound on $P(w_1, \dots, w_n)$. \square

Corollary 3.1.2 actually yields a slightly stronger upper bound which we present in the next corollary.

COROLLARY 3.1.4. *Let $\omega_1, \dots, \omega_n$ be the integer weights w_1, \dots, w_n rearranged into descending order. Then $P(w_1, \dots, w_n) \leq 2 + \log_t (\sum_{1 \leq i \leq \eta} t^{(\omega_i)})$, where $\eta = \lceil (n-t+1)/t \rceil$.*

Proof. It is not hard to see that if $W' = w'_1, \dots, w'_n$ is any rearrangement of the weight list $W = w_1, \dots, w_n$ such that $w'_{jt} = w_j$ for $1 \leq j \leq \lfloor (n-t+1)/t \rfloor$ and $w'_n = w_n$, then $G(W) \leq G(W') \leq t \sum_{1 \leq i \leq \eta} t^{(\omega_i)}$. Combining this with Corollary 3.1.2 completes the proof. \square

3.2. The algorithm. We now concentrate on presenting a linear algorithm for constructing an optimal minimax tree for a list of integer weights. The basic idea in our original algorithm was to perform a right to left scan on the initial list of weights until a r.l.m. t -tuple was encountered, merge the weights in the r.l.m. t -tuple to form the appropriate new weight, back up sufficiently far to ensure that no new r.l.m. t -tuple would be missed, and continue the right to left scan. By proving that one never had to back up more than $2t$ weights, we were able to bound the running time by kn , where k is a constant independent of t . Inspired by an idea in an algorithm for a related problem due to Coppersmith, Klawe and Pippenger [1], and changing our direction of scan from left to right, we were able to modify our algorithm slightly so that back-ups are unnecessary. As a result the proofs of correctness and the bound on the running time are somewhat simpler. The idea which simplified our algorithm is that it is possible to "fill in valleys" in a sequence of weights without increasing the minimax value. More precisely, we have the following lemma.

LEMMA 3.2.1. *Let $W = w_1, \dots, w_n$ and suppose there exist $0 < s < t$ and $i \leq n - s - 1$ such that*

$$\max \{w_{i+j} : 1 \leq j \leq s\} \leq \min \{w_i, w_{i+s+1}\}.$$

Let W' be the list $w_1, \dots, w_i, z_{i+1}, \dots, z_{i+s}, w_{i+s+1}, \dots, w_n$, where

$$z_{i+j} = \min \{w_i, w_{i+s+1}\} \quad \text{for } 1 \leq j \leq s.$$

Then $P(W') = P(W)$.

Proof. Let T be an optimal minimax tree for W , and suppose u is the father of a leaf with weight w_{i+j} for some j with $1 \leq j \leq s$. Since $s < t$ either the leaf with weight w_i or the leaf with weight w_{i+s+1} must be a descendant of u , and hence increasing w_{i+j} to $\min \{w_i, w_{i+s+1}\}$ cannot increase the weight of u . \square

Before presenting the actual implementation we give an intuitive description of how the algorithm functions. We maintain a list of the weights of the current set of vertices which have been created, but whose fathers have not been created yet. A pointer divides this list into two sublists, those weights which have been processed (to the left of the pointer), and those which have not (the pointer points to the head of this sublist).

We use Lemma 3.2.1 to ensure that the list of processed weights forms a nonincreasing sequence. Thus when we examine a new weight to be processed, if it is greater than the last processed weight we attempt to fill in a valley, raising as many processed weights as necessary to the new weight. If this should turn out to be more than $t - 1$, of course Lemma 3.2.1 does not apply, but in this case, because all weights are integers, it is easy to see that the last t processed weights must be a r.l.m. t -tuple. Thus in this case we merge them to form a new weight which is placed at the head of the unprocessed weight sublist. If on the other hand, the new weight is not greater than the last processed weight, we simply move the new weight to the end of the processed weight sublist by moving the pointer to the right.

Each time Lemma 3.2.1 is used, as many as $t - 1$ weights might have to be updated, and if we actually did this, although the algorithm would run in linear time, the constant would depend on t . However, since the sequence of processed weights form a descending step function, it is only necessary to store the positions and values of the weights where steps occur. Moreover, since an update at a step means that that step is permanently obliterated, each weight can represent a step at most once, and hence the number of changes made is bounded by $n + (n - 1)/(t - 1)$, the total number of vertices in the tree.

3.2.2. Implementation. To avoid handling special cases we add a dummy weight $= \infty$ to each end of the list of weights w_1, \dots, w_n and initialize a doubly linked list called LIST to hold this extended list. We use the variable PTR to represent the pointer into the doubly linked list of weights, and use $W(\text{PTR})$ to denote the weight in the list to which the pointer is pointing. The integer variables TOTAL and COUNT keep track of the total number of nondummy weights in the list and processed weight sublist respectively. We initialize PTR to point to w_1 , the first non dummy weight, and initialize TOTAL and COUNT to n and 0 respectively. The descending step function of weights is stored in two arrays, WT and POS, forming a double stack which is accessed by an integer variable, TOP. The bottom entry of the (WT, POS) stack contains the leftmost weight and position of the step function, and the remaining entries from bottom to top store the weights and positions of the steps from left to right. The (WT, POS) stack is initialized to hold the entry $(\infty, 0)$.

INTEGER WEIGHT ALGORITHM

```

/* initialization */
LIST:=∞, w1, . . . , wn, ∞;
PTR points to w1;
TOTAL:= n;
COUNT:= 0;
TOP:= 1;
WT (TOP):= ∞;
POS (TOP):= 0;
WHILE TOTAL> 1 DO;
  IF W (PTR)< WT (TOP) THEN DO;
    /* pointer weight is processed and step function is extended to include
    this weight and position */
    TOP:= TOP+ 1;
    WT (TOP):= W (PTR);
    POS (TOP):= COUNT+ 1;
    move PTR right;
    COUNT:= COUNT+ 1;
  END;
  ELSE DO;
    /* check whether last t processed weights are a r.l.m. t-tuple */
    WHILE (WT (TOP)< W (PTR) AND POS (TOP)>COUNT- t
      + 1) DO;
      LASTPOS:= POS (TOP);
      TOP:= TOP- 1;
    END;
    /* if not, pointer weight is processed and step function is updated */
    IF WT(TOP)≥ W(PTR) THEN DO;
      IF WT(TOP)> W (PTR) THEN DO;
        TOP:= TOP+ 1;
        WT (TOP):= W (PTR);
        POS (TOP):= LASTPOS;
      END;
      move PTR right;
      COUNT:= COUNT+ 1;
    END;
    /* if so, merge the last t processed vertices to form a new weight */
    ELSE DO;
      /* delete the last t processed weights from LIST */
      /* and update the step function if necessary */
      DELETE weights in positions COUNT- t+ 1, . . . , COUNT from
      LIST;
      IF POS (TOP)= COUNT- t+ 1 THEN TOP:= TOP- 1;
      COUNT:= COUNT- t;
      /* add a new weight which is the weight of the father */
      /* of the vertices whose weights have just been deleted, */
      /* and point PTR at it */
      INSERT WT (TOP)+ 1 in LIST on the left of PTR;
      move PTR left;
      TOTAL:= TOTAL- t+ 1;
    END;
  END;
END;
/* output the minimax value P(w1, . . . , wn) */
OUTPUT W(PTR);

```

3.2.3. Correctness. It is easy to check that the variables COUNT and TOTAL function as they are supposed to, and that the value of TOP is always at least 1. Also because of the dummy ∞ weights at the head and tail of the doubly linked list, it is not hard to verify that PTR is never moved right when it points at the tail (thus the tail dummy weight is never processed), and the head dummy weight is never deleted. By induction one can easily prove that at all times we have $\infty = \text{WT}(1) > \text{WT}(2) > \dots > \text{WT}(\text{TOP})$, $0 = \text{POS}(1)$, and $1 = \text{POS}(2) < \dots < \text{POS}(\text{TOP}) \leq \text{COUNT}$ whenever $\text{COUNT} \geq 1$. For current values of weights on the left of PTR (i.e. processed weights) we use the (WT, POS) stack as follows. If $0 \leq j \leq \text{COUNT}$, let $I(j) = \max\{i: 1 \leq i \leq \text{TOP}, \text{POS}(i) \leq j\}$. Then the weight in position j is $\text{WT}(I(j))$. To see that the output of the algorithm is indeed $P(w_1, \dots, w_n)$, it suffices to note that whenever the values of processed weights are increased, the minimax value is not increased (Lemma 3.2.1), and whenever t weights are merged, those t weights form a r.l.m. t -tuple and hence by Theorem 2.1 the merge cannot increase the minimax value either. As a final observation we note that the two statements involving the temporary variable LASTPOS can be deleted without affecting the execution of the algorithm. Their only purpose is to make the algorithm easier to follow.

3.2.4. Timing analysis. In any pass through the main while loop, either the pointer moves right or a t -tuple of weights is merged to create a new weight. During the execution of the algorithm exactly $(n-1)/(t-1)$ new weights are created, and the pointer moves right past a weight at most once. This shows that the total number of passes through the main while loop is bounded by $n + 2(n-1)/(t-1)$. Each pass through the inner while loop causes an entry to be popped from the (WT, POS) stack. Since each weight can be an entry in the (WT, POS) stack at most once, the total number of passes through the inner while loop is bounded by $n + (n-1)/(t-1)$. Finally, it is obvious that the number of deletions must be bounded by the total number of weights, i.e., $n + (n-1)/(t-1)$. Combining all this, and observing that all other operations are done at most once during each pass through the main while loop, it is obvious that the algorithm runs in linear time, and the constant is independent of t .

4. General weights. For $W = w_1, \dots, w_n$ and a any real number, let $W(a)$ be the list $\lceil w_1 - a \rceil, \dots, \lceil w_n - a \rceil$. Moreover for $1 \leq i \leq n$ let $a_i = w_i - \lfloor w_i \rfloor$. Since $\lceil w - a \rceil + a \geq w$ for any a and w , it is clear that $P(W(a_i)) + a_i \geq P(W)$ for each i .

LEMMA 4.1. *Let j such that $P(W(a_j)) + a_j = \min\{P(W(a_i)) + a_i: 1 \leq i \leq n\}$. Then $P(W) = P(W(a_j)) + a_j$ and the optimal tree for $W(a_j)$ yields an optimal tree for W .*

Proof. First we show that $P(W(a_i)) + a_i \leq P(W)$ for some i . By the remark preceding the lemma, this implies that $P(W) = P(W(a_j)) + a_j$. By definition $P(W) = \max\{w_i + l_i: 1 \leq i \leq n\}$, where l_i is the length of the path from the leaf with weight w_i to the root in a tree T which is optimal for W . Choose i such that $P(W) = w_i + l_i$. Let T' be the weighted tree obtained from T by changing each leaf weight w_k to $\lceil w_k - a_i \rceil + a_i$ and adjusting the weights of internal vertices as necessary, and let w' be the weight of the root of T' . Clearly $P(W(a_i)) + a_i \leq w' = \max\{\lceil w_k - a_i \rceil + a_i + l_k\} \leq w_i + l_i = P(W)$, since for each k , we have $\lceil w_k - a_i \rceil + a_i + l_k = \lceil w_k + l_k - a_i \rceil + a_i \leq \lceil w_i + l_i - a_i \rceil + a_i = w_i + l_i$.

Now suppose S is any optimal tree for $W(a_j)$ and let S' be the tree obtained by replacing each leaf weight $\lceil w_k - a_j \rceil$ by w_k and adjusting internal weights as necessary. To see that S' is optimal for W it suffices to show that its root has weight at most $P(W(a_j)) + a_j$, but this is obvious since we increased each leaf weight by at most a_j . \square

The preceding lemma shows that the problem reduces to finding the value of i for which $P(W(a_i)) + a_i$ is minimal, and solving the problem for the (integer) weight list $W(a_i)$.

Let b_1, \dots, b_n be the numbers a_i rearranged into ascending order (this can be done in $O(n \log n)$ time), and let $b_0 = b_n - 1$. It is easy to verify that $P(W(b_0)) \geq P(W(b_1)) \geq P(W(b_2)) \geq \dots \geq P(W(b_n))$ and that $P(W(b_0)) - P(W(b_n)) = 1$. Let j be the smallest i such that $P(W(b_i)) = P(W(b_0)) - 1$. Since $P(W(b_i))$ is an integer for each i , it is easy to see that $P(W(b_j)) + b_j$ is the smallest of the values $P(W(b_i)) + b_i$. Using binary search to find j , one solves at most $O(\log n)$ integer weight problems, thus yielding an $O(n \log n)$ algorithm for constructing an optimal t -ary tree given general weights.

The same technique can be used to obtain arbitrarily close approximations to $P(W)$. Let W_k be the list $\lceil 2^k w_1 \rceil / 2^k, \dots, \lceil 2^k w_n \rceil / 2^k$. Obviously $P(W_k) \geq P(W) \geq P(W_k) - 1/2^k$, and since there are at most 2^k distinct values in the sequence $\lceil 2^k w_i \rceil / 2^k - \lfloor \lceil 2^k w_i \rceil / 2^k \rfloor$, one can find $P(W_k)$ by solving at most $k + 1$ integer weight subproblems.

We now extend the proofs of the upper bounds on $P(w_1, \dots, w_n)$ to general weights. As above, let j be the integer such that $P(W(b_j)) + b_j$ is the smallest of the values $P(W(b_i)) + b_i$. Upper bounds on $P(W(b_{j-1}))$ imply corresponding upper bounds on $P(W)$. We illustrate this in the proof of the following proposition. The same technique can be used to extend Corollary 3.1.4 to general weights but we leave the details to the reader.

PROPOSITION 4.2. *For any list $W = w_1, \dots, w_n$ we have*

$$P(W) < 2 + \log_t \left(\sum_{1 \leq i \leq n} t^{(w_i)} \right).$$

Proof. If $a_i \leq b_{j-1}$ then $\lceil w_i - b_{j-1} \rceil = \lfloor w_i \rfloor$. Moreover, since b_1, \dots, b_n is a rearrangement of a_1, \dots, a_n into ascending order, if $a_i > b_{j-1}$ then in fact $a_i \geq b_j$ and hence $\lceil w_i - b_{j-1} \rceil = \lfloor w_i \rfloor + 1 = w_i - a_i + 1 \leq w_i - b_j + 1$. From this it is clear that for any i we have $\lceil w_i - b_{j-1} \rceil - 1 + b_j \leq w_i$. Now

$$t^{P(W)} = t^{P(W(b_j)) + b_j} = t^{P(W(b_{j-1})) - 1 + b_j} < t^2 \sum t^{(\lceil w_i - b_{j-1} \rceil - 1 + b_j)}$$

by Corollary 3.1.3. Combining all this yields $t^{P(W)} < t^2 \sum t^{(w_i)}$ as desired.

5. Examples. In this section we present examples which show that our results are in some sense best possible. The first example shows that the upper bounds on $P(w_1, \dots, w_n)$ given in Corollaries 3.1.2, 3.1.3 and 3.1.4 are tight. The second example illustrates the limitations of local criteria in determining which t -tuples should be merged to form an optimal tree in the case of general weights. One of the surprising things about this second example is that it suffices to consider lists of weights in which all but t of the weights are integers, and the other t weights are multiples of $\frac{1}{2}$. Our final example illustrates the problems which occur when $n \not\equiv 1 \pmod{t-1}$.

Example 5.1. For M, N and k positive integers we define $W(M, N, k)$ to be the list of weights w_1, \dots, w_n , where $n = t^k + t - 1$, $w_i = M$ if $i \equiv 0 \pmod t$, and $w_i = -N$ otherwise. It is not hard to see that $P(W(M, N, k)) = M + k + 1$. Clearly $G(W(M, N, k)) = t^k t^M = t^{M+k}$ showing that Corollary 3.1.2 is tight, and $\sum_{1 \leq i \leq n} t^{(w_i)} = t^{k-1} t^M + (t-1)(t^{k-1} + 1)t^{-N}$. Thus for any $\epsilon > 0$, if N is large enough we have $\sum_{1 \leq i \leq n} t^{(w_i)} < t^{M+k-1} + \epsilon$, which shows that Corollary 3.1.3 cannot be improved. Since $\eta = t^{k-1}$ and $\omega_i = M$ for $1 \leq i \leq \eta$, obviously Corollary 3.1.4 is tight also.

Example 5.2. By a local merging criterion we mean a set of conditions concerning a sequence of weights, in which a particular t -tuple is specified with the following properties. The length of the sequence is a function of t , and whenever a list of weights has a sequence satisfying the conditions then there is some tree which is optimal for that list in which the specified t -tuple of the sequence is merged. Thus the concept of right locally minimal t -tuple is a local merging criterion. The next example will show that there is no local merging criterion such that every list of general weights has a subsequence satisfying the local merging criterion. More precisely, the following example gives two arbitrarily long lists of weights which are locally indistinguishable, yet have fundamentally different optimal tree structures.

For $n \geq t$ let $W(n)$ be the list w_1, \dots, w_n such that $w_i = \frac{3}{2}$ for $1 \leq i \leq t-1$, $w_i = 1$ for $t \leq i \leq n-1$, and $w_n = \frac{3}{2}$. Let us consider the lists $W(t^k)$ and $W(t^k - t(t-1))$ for any integer $k \geq 3$. Using the techniques of § 4 it can be shown that for each of these lists, the optimal tree is unique. However, in the optimal tree for $W(t^k)$ the weights w_1, \dots, w_t must be merged, whereas in the optimal tree for $W(t^k - t(t-1))$ they are not. Figure 2 shows the optimal trees for $W(t^k)$ and $W(t^k - t(t-1))$ when $t = k = 3$.

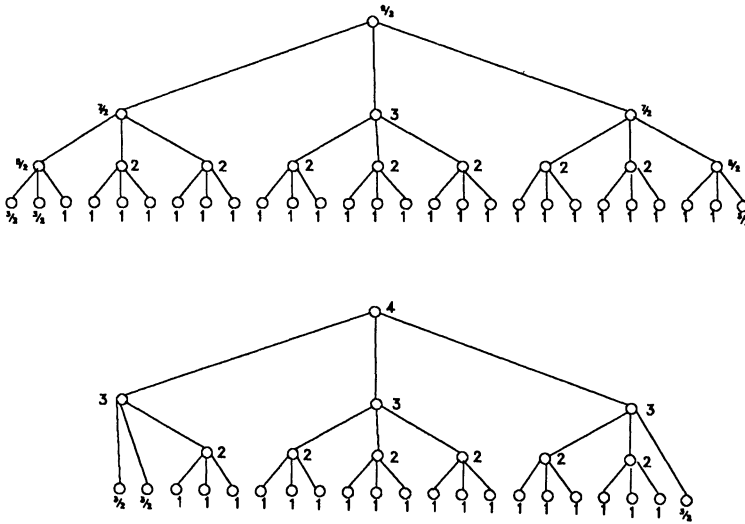


FIG. 2

Example 5.3. We conclude with some remarks concerning the case when the number of weights, n , is not equal to $1 \pmod{t-1}$. There are (at least) two possible ways to extend the definition of t -ary tree to this case. One is that at most one internal vertex does not have exactly t sons, and the other is that the number of internal vertices in the tree is exactly $\lceil (n-1)/(t-1) \rceil$. In general these definitions lead to different values of $P(W)$ as is illustrated in Fig. 3 for the list $1, 1, 1, 2, 1, 1, 1, 2$ with $t = 4$. One can observe that no matter which definition is used, there is an optimal tree in which every internal vertex with fewer than t leaves has only leaves as sons. In spite of this it does not seem to be easy to decide which sets of leaves should have a father with fewer than t sons. For example the list $2, 1, 1, 2$ with $t = 3$ shows that adding the appropriate number of dummy $-\infty$ weights at one end will not succeed in general. This is illustrated in Fig. 4.

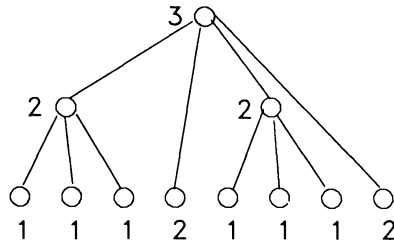
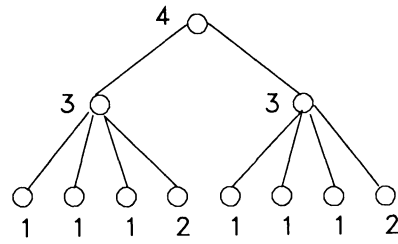


FIG. 3

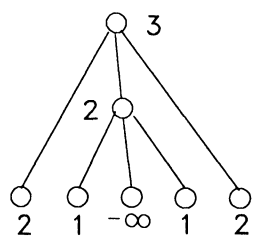
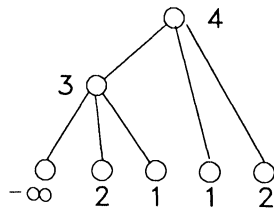


FIG. 4

REFERENCES

[1] D. COPPERSMITH, M. KLAWE AND N. PIPPENGER, *Alphabetic minimax trees of degree at most t*, this Journal, 14 (1985), to appear.

[2] A. M. GARSIA AND M. L. WACHS, *A new algorithm for minimal binary search trees*, this Journal, 6 (1977), pp. 622-642.

[3] E. N. GILBERT AND E. F. MOORE, *Variable length binary encodings*, Bell System Tech. J., 38 (1959), pp. 933-968.

[4] M. C. GOLUMBIC, *Combinatorial merging*, IEEE Trans. Comput. (1976), pp. 1164-1167.

[5] L. GOTTLIEB, *Optimal multi-way search trees*, this Journal, 10 (1981), pp. 422-433.

[6] H. J. HOOVER, M. M. KLAWE AND N. J. PIPPENGER, *Bounding fan-out in logical networks*, J. Assoc. Comput. Mach., 31 (1984), pp. 13-18.

- [7] T. C. HU, *A new proof of T-C algorithms*, SIAM J. Appl. Math., 25 (1973), pp. 83–94.
- [8] T. C. HU, D. J. KLEITMAN AND J. K. TAMAKI, *Binary trees optimum under various criteria*, SIAM J. Appl. Math., 37 (1979), pp. 246–256.
- [9] T. C. HU AND A. C. TUCKER, *Optimal computer search trees and variable-length alphabetical codes*, SIAM J. Appl. Math., 21 (1971), pp. 514–532.
- [10] D. A. HUFFMAN, *A method for the construction of minimum redundancy codes*, Proc. IRE, 40 (1952), pp. 1098–1101.
- [11] A. ITAI, *Optimal alphabetic trees*, this Journal, 5 (1976), pp. 9–18.
- [12] D. J. KLEITMAN AND M. E. SAKS, *Set orderings requiring costliest alphabetic binary trees*, SIAM J. Alg. Disc. Meth., 2 (1981), pp. 142–146.
- [13] D. E. KNUTH, *Optimum binary search trees*, Acta. Informat., 1 (1971), 14–25.
- [14] ———, *The Art of Computer Programming*, Vol. 3, Addison-Wesley, Reading, MA, 1973.
- [15] C. E. SHANNON, *A mathematical theory of communication*, Bell System Tech. J., 27 (1948), pp. 379–423.

SOLVING UNDIRECTED GRAPH PROBLEMS ON VLSI*

SUSANNE E. HAMBRUSCH† AND JANOS SIMON‡

Abstract. We study VLSI solutions to the connected component problem on networks that have area too small to store all the edges of the graph for the entire computation. We give lower bounds on the time needed to solve this problem on such networks, as well as an optimal algorithm. The lower bounds use a new proof technique combining adversary strategy, information flow, and Kolmogorov complexity arguments. The lower bounds obtained for the connected components problem hold for a number of other undirected graph problems.

Key words. VLSI complexity, lower bound techniques, information theoretic arguments, graph problems

1. Introduction. The potential use of VLSI technology for direct hardware implementation of algorithms has motivated much recent research in parallel computation and in the design of special purpose chips tailored to a particular problem [AA], [BK], [GKT], [LS], [T]. In this paper we study to the *connected component problem* (ccp), which is a paradigm for many other graph problems. We consider solutions on chips of small area, i.e., area too small to store all the edges of the graph explicitly within the chip for the entire computation. The assumption is interesting for both upper and lower bounds: for algorithms because small area may mean that actual implementation of the design could be attempted; and for lower bounds, because our solution yielded a new proof technique.

Many lower bounds on the quantity AT^2 for VLSI chips have been obtained, using information flow arguments [BK], [T], [V]. Our results are also obtained by measuring information transfer across a cut in the chip. However, previous arguments [AA], [LS], [T], [V] are essentially static: by considering the location of the initial data on the chip and the structure of the problem, the bound on the information transfer can be obtained, either by counting the number of input-output pairs [AA], [T], [V], or by counting the number of such pairs together with the number of machine configurations (as in the crossing sequence proofs of [LS]). In both cases, the counting is done at the *beginning* of the computation. We note that there are proofs using similar static localized information arguments that yield lower bounds on the complexity of on-line computations [P], [PSS]. It would be desirable to extend these techniques to the dynamic case, and to be able to talk about localized information *during* the computation. This could yield more precise lower bounds, lower bounds for unrestricted computations, and, in general, a handle on intermediate states of computation. Our results are a first step in this direction. We consider information transmission in chips of small area, that cannot contain all the information (or all the input) at any given time.

Applying the information transfer technique to undirected graph problems yields $AT^2 = \Omega(n^2(\log n)^2)$ [J]. This lower bound is tight within factors of $\log n$ for chips of $\Omega(n^2)$ area: [H1] presents an algorithm for the ccp that achieves $AT^2 = O(n^2(\log n)^8)$ on a network of $O(n^2(\log n)^2)$ area. (A similar result can also be found in [NMB].) For chips of small area the lower bound of $AT^2 = \Omega(n^2(\log n)^2)$ is far from optimal. For example, on networks of $O(n)$ area, the time required to read n^2 inputs is $\Omega(n)$, which gives $AT^2 = \Omega(n^3)$.

* Received by the editors February 19, 1982, and in revised form March 12, 1984.

† Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907.

‡ Department of Computer Science, Pennsylvania State University, University Park, Pennsylvania 16802.

The research of this author was supported in part by the National Science Foundation under grant MCS81-04876 and the U.S. Army Research Office under contract DAAG29-82-K-0110.

Our lower bound technique combines adversary, Kolmogorov complexity, and information flow arguments. We obtain lower bounds for verification problems (i.e., given a solution to the problem and a graph, verify that the solution is the correct solution for the given graph). In general it seems to be harder to prove lower bounds for verification problems than for problems where the solution must be computed and multiple outputs are produced. We show that the lower bounds for the verification problem also hold for the original problem.

We examine several forms of input for a graph: adjacency matrix, adjacency lists, and an unordered sequence of edges. We show that the form of input is crucial for the performance of the algorithm. In this paper we only consider *when oblivious* chips; i.e., the environment, rather than the chip, determines the moments in time at which the input is given to the chip. Networks of area large enough to store all the inputs before and during the actual computation are not sensitive to the when obliviousness of the model. This does not hold for all problems solved on networks of small area: while the performance of the algorithms presented in [KL] does not depend on when obliviousness, efficient algorithms for undirected graph problems seem to be harder to find in the when oblivious model. Furthermore, for undirected graph problems we prove stronger lower bounds for the when oblivious model. Algorithms for the ccp and other graph problems on both VLSI models can be found in [AK], [GKT], [H1], [NMB] and [LV].

For every when oblivious chip of $o(n^2)$ area, we prove that there exists a graph in the class of adversary graphs for which we can obtain a lower bound on the time needed to process this graph. We show that a certain amount of time must elapse between successive input sequences, because there will always be a segment on the chip that has an information deficiency about the input supplied to it. The next wave of input must then wait until enough information flows into the deficient segment. For networks of $O(n)$ area having constant width and input in the form of adjacency lists or edges, we prove that for some n vertex graph, the time elapsing between two consecutive input sequences is at least $\Omega(n)$. Since there may be n input sequences, we have $T = \Omega(n^2)$, and $AT^2 = \Omega(n^5)$, which can be achieved by the algorithm presented in this paper. We first apply our lower bound technique to chips of $O(n)$ area, and then show how to extend the results to chips of $O(nm)$ area, $m = o(n)$.

The paper is organized as follows. In § 2 we describe the VLSI model and the different forms of input for graphs. Section 3 contains the lower bound results. We first prove the lower bounds for the verification problem of the ccp and show that a lower bound on the time needed to solve the verification problem of the ccp on a network of $O(n)$ area is $\Omega(n^2/k)$, when the network can be circumscribed by a rectangle of size $k \times n/k$, $1 \leq k \leq n^{1/2}$. We generalize this technique to obtain lower bounds on the time for networks of $O(nm)$ area. We then show that the connected component problem is at least as hard as its verification problem. Thus a lower bound on the verification problem is also a lower bound for the ccp. We conclude this section with lower bound results for an approximate verification problem. In § 4 we present an optimal algorithm for the ccp.

2. Model of computation and forms of input. In a number of recent papers ([BK], [CM], [LS], [T]), parallel models suitable for VLSI implementation have been developed and refined. Our model follows Thompson's grid model [T].

(1) Each *processing element* (PE) of the chip contains r registers, r constant, and is able to execute a simple set of instructions. Each register consists of $\log n$ bits, where n is the number of vertices in the graph.

(2) A PE is connected to a constant number of other PE's. The PE's operate synchronously.

(3) The chip communicates with its environment through the input/output ports (I/O ports). Since we are interested in the information flow requirements of the problem, and not in the information flow requirements based on the I/O restrictions of the chip, we assume each PE on the chip can read input and produce output.

(4) Each input is read once, and each output is generated once.

(5) The chips are *when and where oblivious*. A chip is when and where oblivious when the time and locations at which the inputs arrive (and the outputs are generated) are independent of the input data. On a when oblivious chip the time elapsing between the reading of two successive input waves is fixed by the environment of the chip and is the same for all successive input waves.

(6) In one *time unit* a PE can either transmit the content of one of its registers to an adjacent PE, or it can perform an operation on its registers. When computing the *area* of a chip, we consider a PE to occupy unit area, and a wire to have unit width.

Within the VLSI literature two approaches, the bit- and the word-oriented approach, have been used when defining time and area. We next discuss these and justify our assumptions. Parallel time is the number of *steps* required to generate all the outputs. In the *bit-oriented* model one step is a time pulse of length τ , in which one bit can be sent across a wire of width λ , and where τ and λ are technology dependent constants. This definition is used in a number of papers dealing with lower bounds and with problems that require operations on bits [BK], [T], [V].

This paper uses the *word-oriented* model, where one step is a *cycle*. In one cycle $\log n$ bits can be sent from one PE to another PE or a local computation within a PE can take place. In our model a PE has length l_v and width l_h , where $l_v \leq l_h$ and $l_v \cdot l_h = r \log n$. Each connection consists of at most l_v bit-carrying wires in the vertical, and at most l_h bit-carrying wires in the horizontal dimension. Hence, a connection between two PE's in the vertical dimension contains l_v/λ bit-carrying wires, $\lambda \leq l_v \leq (r \log n)^{1/2}$, and a cycle consists of $\tau \lambda \log n/l_v$ time pulses. The assumption that the PE's operate synchronously prevents the PE's from sending the content of a register faster along the horizontal dimension than the vertical one. Word-oriented models are used in most algorithms for VLSI networks that perform operations on entire registers rather than on bits of the registers [AK], [GKT], [H2], [KL], [LV].

All the lower bounds developed in this paper measure time in terms of cycles. This makes it easier to compare our results to existing upper bounds, and it eliminates implementation-dependent factors of $\log n$'s. The issue of whether time is proportional to the wire length (synchronous versus diffusion model) is avoided. We count one cycle for the communication between PE's, and the lower bounds hold in both models.

The area is the space necessary to lay out the PE's with their interconnections on a small, constant number of parallel layers. Strictly speaking, a PE has $O(\log n)$ area, interconnections may have $O(\log n)$ width, and transmitting the contents of a register may take several time pulses. In order to achieve a cleaner presentation, we look at cycles instead of time pulses, and consider the area of a register, not that of a bit, as a unit. Thus, processors have $O(1)$ area, and wires have unit width. This is analogous to the count of register operations in the sequential model, and makes it possible to compare parallel and sequential performance.

Three common representations for a graph are: adjacency matrix, adjacency lists and unordered edge list. When the graph is given to the chip in form of an *adjacency matrix* it reads the n rows of the matrix. Let PE_1, \dots, PE_m be the m processing elements receiving inputs, V be the vertex set, $V = \{1, 2, \dots, n\}$, and E be the set of

edges. If $(i, j) \in E, i < j$. When the graph is given in the form of *adjacency lists* the edges are read in lexicographic order; i.e., if PE_i reads (u_i, v_i) and PE_{i+1} reads (u_{i+1}, v_{i+1}) , then $u_i < u_{i+1}$, or, $u_i = u_{i+1}$ and $v_i < v_{i+1}$. Thus PE_i reads the $((j-1)m + i)$ th edge of the list of edges in lexicographic order in the j th input sequence, $1 \leq i < m, 1 \leq j \leq e/m$, where e is the number of edges in the graph. When the graph is given in the form of an *unordered edge list* (hereafter called input in the form of edges) the edges (i, j) are read into the chip in arbitrary order. See Fig. 2.1. In this paper we primarily discuss input in the form of adjacency lists and edges. We refer to [H1] for algorithms with input in the form of an adjacency matrix.

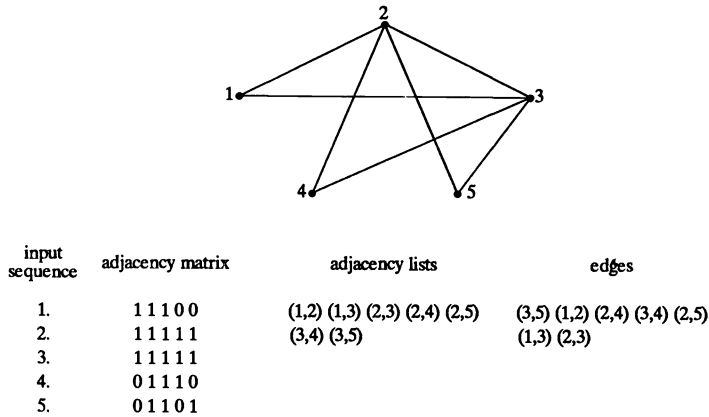


FIG. 2.1. The input sequences for the 3 forms of input ($m = 5$).

A chip receiving the input in the form of adjacency lists or edges reads only e inputs, as compared with n^2 inputs for the adjacency matrix. Every graph represented in the form of a matrix or lists has a unique description, while different input sequences can describe the same graph when represented in the form of edges. Graphs given in the form of an adjacency matrix allow the PE's of the chip to know before the computation what inputs they will read (e.g. PE_i reads the i th column). This is no longer possible when the graph is given in the form of lists or edges, since the inputs PE_i receives depend on the graph in the first case and are arbitrary in the second case.

3. Lower bound results. We prove lower bounds on the time needed to solve undirected graph problems on when oblivious chips of small— $o(n^2)$ —area, when the graph is given in the form of adjacency lists or edges. In this paper we only deal with the connected component problem (i.e., two vertices are in the same connected component if and only if there is a path between them), but the results extend to many other problems (e.g., biconnectivity, bridgeconnectivity, minimum-cost spanning tree). The proof technique combines adversary, Kolmogorov complexity, and information flow arguments, and the lower bounds obtained hold also for verification problems.

3.1. The verification problem. In the *verification problem* of the ccp we are given as input an encoding of a solution to the ccp and a description of a graph G . We have to determine whether or not the connected components of graph G are those specified in the solution and produce a 0/1 answer. The minimal length of an encoding describing the solutions to the ccp is $n \log n$, and $n \log n$ bits are sufficient (e.g., describing the components by component numbers, as in § 4, requires $n \log n$ bits).

Verification is easy when input is given in the form of an adjacency matrix. In [H1] we show how to verify in time $O(n)$ on a network of $O(n)$ area. This time is optimal for a network of $O(n)$ area. We show that the verification of the ccp is harder when input is given in the form of adjacency lists or edges. We first define the class of adversary graphs used in the proofs.

An *adversary graph* $G^* = (V^*, E^*)$ consists of $n/6 + 1$ connected components, where $n/6$ components consists of a single edge, and one component is a connected subgraph on the remaining $2n/3$ vertices, which we term the *filler graph*. See Fig. 3.1.

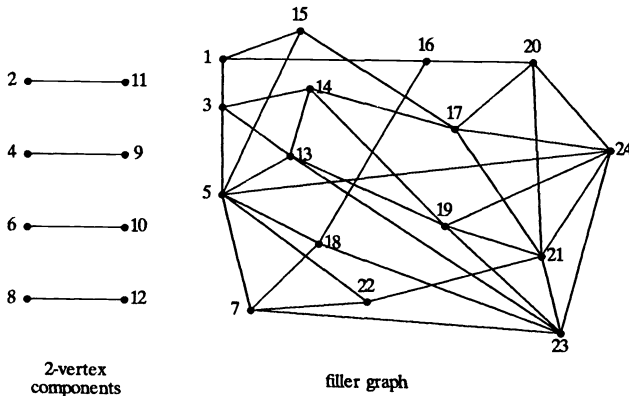


FIG. 3.1. An adversary graph with 24 vertices.

Let the vertex set $V^* = \{1, 2, \dots, n\}$. Each one of the $n/6$ edges representing a 2-vertex-component is incident to an even numbered vertex between 2 and $n/3$ and a vertex between $n/3 + 1$ and $n/2$. The first $n/3$ odd numbered vertices and the vertices between $n/2 + 1$ and n are the vertices of the filler graph, which is one connected component. Formally, E^* is defined as $\{(2i, x_i) | 1 \leq i \leq n/6, n/3 + 1 \leq x_i \leq n/2, x_i \neq x_j \text{ for } i \neq j\} \cup E'$. Let $V_x = \{2i + 1 | 0 \leq i < n/6\} \cup \{n/2 + 1, \dots, n\}$. Then E' , the edges of the filler graph, are a subset of $V_x \times V_x$ so that the subgraph induced by E' is connected. The *filler edges* will be used to “fill up” input sequences, i.e., make them sufficiently long for the lower bound argument. The edges representing 2-vertex-components are the ones of interest for the adversary.

Kolmogorov complexity, also known under the term *descriptive complexity*, measures the minimal number of bits needed for descriptions. We refer to [PSS] for formal definitions. We use Kolmogorov complexity to measure the length of the minimal encoding describing the connected components of the adversary graphs. The next two lemmas formalize the intuitively obvious fact that at least $\alpha n \log n$ bits must be used to encode an adversary graph, for some $\alpha > 0$.

LEMMA 3.1. *There are at least $2^{\alpha n \log n}$ different adversary graphs G^* , for some constant α , and thus at least $\alpha n \log n$ bits are needed to encode all different adversary graphs.*

Proof. There are $(n/6)!$ different ways to choose the $n/3$ 2-vertex-components, which is at least $2^{\alpha n \log n}$ for some constant α . Thus encoding all adversary graphs with different 2-vertex-components (without counting the bits needed to encode the filler graph) requires at least $\alpha n \log n$ bits, and the lemma follows. \square

It does not immediately follow that the chip will need $\alpha n \log n$ bits of its registers for the encoding, since the chip may use the fixed network configuration in the encoding of the solution (e.g. use the indices of the PE's or the interconnection pattern or the

fact that a given register is not used in the encoding). While an encoding of the solution of length $\alpha n \log n$ is input, the chip is allowed to modify (and, possibly compress) the solution before the graph is being input. But it still must be able to distinguish between the $2^{\alpha n \log n}$ different solutions, each one describing an adversary graph with different connected components. In Lemma 3.2 we show that for chips of $O(n)$ area there cannot be sufficiently clever encodings of the input solution on the chip: at least $\alpha' n \log n$ bits of the registers are needed to distinguish between all different solutions, for some constant α' .

Our notion of an “encoding” is a very generous one: an encoding is setting some registers of some PE’s to a given value. The length of the encoding is the sum, over the PE’s used, of the number of bits used in their registers. Thus, we do not count unused bit portions in registers, or PE’s unaffected by the encoding process. This implies that we can distinguish whether a PE is used or not in the encoding without using a bit to specify this fact. (One may think, for example, of a situation where all registers are initially set to 0, then the encoding is given to the selected PE’s. All PE’s can remember, in their control, whether or not they received a part of the encoding.) Thus, less than n bits of the “encoding” may specify 2^n configurations—if there are 2^n processors, a single bit will do. While this may be unrealistic (after all, a finite control does use area), we are proving lower bounds, and our results show that the recognition problem is hard, even if such unnatural encoding tricks are used.

LEMMA 3.2. *Let N be a chip of $O(n)$ area that reads a minimal length encoding. Let the length of the encoding be $\alpha n \log n$. Then in order to be able to distinguish between all $2^{\alpha n \log n}$ different encodings, at least $\alpha' n \log n$ bits of the registers of the PE’s of the chip are needed, for some constant α' .*

Proof. For simplicity assume the chip has n PE’s and each PE has one register ($r = 1$). If the network has cn PE’s, for some constant c , or $r > 1$, the results will hold with different constants.

Assume the chip needs at most s bits to store the (possibly modified) encoding on the chip. The number of different configurations specified by i bits when the locations of which i bits are to be used can be changed is $\binom{n \log n}{i} 2^i$. Thus, the total number of configurations when at most s bits are allowed to be used is $\sum_{i=0}^s \binom{n \log n}{i} 2^i$, which is less than $\binom{n \log n}{s} 2^{s+1}$ when $s \leq (n \log n)/2$. (If $s > (n \log n)/2$ the lemma is trivially true.) In order to satisfy $\binom{n \log n}{s} 2^{s+1} \geq 2^{\alpha n \log n}$, we need $s \geq \alpha' n \log n$, for some constant α' . \square

3.2. Networks of $O(n)$ area. We will now give the lower bound for networks of $O(n)$ area that solve the verification problem of the ccp. In the lower bound proofs we make a constant number of cuts through the chip and then choose a particular section S of the chip. After having stored (and possibly modified) the solution read as input, we start reading the edges of the adversary graph in the form of adjacency lists or edges. The adversary graph will be such that all the edges representing 2-vertex-components are fed into section S in a constant number of input sequences. Since section S can hold at most half of the information needed by the 2-vertex-components during the verification process, an information exchange between the PE’s in section S and the PE’s outside section S has to occur. In order to be able to continue reading the edges of the filler graph (after having read all 2-vertex-components) and to correctly verify the connected components, a certain time has to elapse between the reading of two consecutive input sequences. This will give the lower bound on the time.

We first present the bound for input given in the form of edges. Note that in the lower bound arguments we measure the exchange of information in bits, but in the final results we express the time in terms of cycles (as described in § 2).

THEOREM 3.3. *Let N be a network of $O(n)$ area that can be circumscribed by a rectangle of size $k \times n/k$, $1 \leq k \leq n^{1/2}$. Then solving the verification problem of the ccp on N with edges as input requires time $\Omega(n^2/k)$, or, expressed in terms of the edges of the graph, time $\Omega(e/k)$.*

Proof. W.l.o.g. assume the network has n PE's. (If the network has cn PE's, for some constant c , the proof below will hold, provided we multiply the appropriate constants by c .) Consider the adversary graphs G^* whose minimal length encoding of the input solution describing the $n/6$ 2-vertex-components requires $\alpha n \log n$ bits on the chip, $0 < \alpha < 1$. According to Lemma 3.2 such graphs exist. Assume the input solution has been read and stored in the chip using $\alpha n \log n$ bits.

Let r be the constant denoting the number of registers in each PE. Make $\max(2r/\alpha - 1, 12r - 1)$ cuts through the network such that each cut is of length $O(k)$ and each section between two cuts contains $\min(\alpha/2r \cdot n, 1/12r \cdot n)$ PE's. See Fig. 3.2.

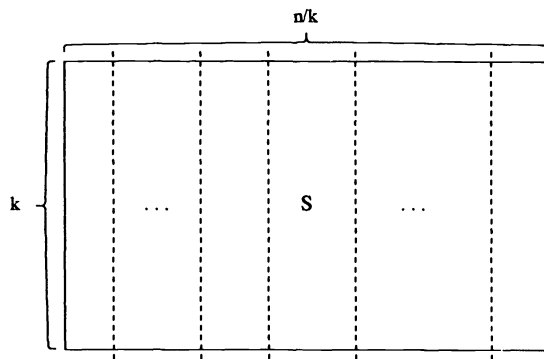


FIG. 3.2. The cuts through the chip.

It is always possible to make such a cut: PE's lying on the cut are counted either to the section to the left or to the right of the cut depending on the number of PE's in the section left of the cut. Each section between two cuts can hold a total of $\min(\alpha/2 \cdot n \log n, 1/12 \cdot n \log n)$ bits. Let S be an arbitrary section.

Consider first the case when $2r/\alpha > 12r$. Then $\alpha < 1/6$, and section S has $\alpha/2r \cdot n$ PE's and can hold a total of $\alpha/2 \cdot n \log n$ bits. In the extreme case all the $\alpha/2 \cdot n \log n$ bits available in section S contain bits of the encoding describing the input solution, which contains the connected components of the edges representing 2-vertex-components. Thus at least $\alpha/2 \cdot n \log n$ bits of the encoding describing the input solution are stored outside section S .

In a constant number of input sequences all the edges representing 2-vertex-components are read into section S , while the PE's outside section S will read edges of the filler graph. Since there are $n/6$ 2-vertex-components and section S contains $\alpha/2r \cdot n$ PE's, $r/3\alpha$ input sequences are needed to read all the 2-vertex-components into section S .

We first give the intuition behind the proof. During the entire verification process an encoding of the input solution has to be compared with an encoding of the connected components of the graph read as input. We expect that while for some of the edges representing 2-vertex-components and read into section S verification might be easy (i.e., their verification can be done without knowing anything about the part of the input solution stored outside section S), other edges will need to know something

about the part of the encoding stored outside section S . We are able to verify at most half of the 2-vertex-components without knowing anything about the encoding stored outside section S . The communication needed between section S and the rest of the chip will cause the delay. More formally, we argue as follows.

There exist adversary graphs G^* such that their 2-vertex-components generate in section S at least $\alpha/2 \cdot n \log n$ bits that have to be compared with the $\alpha/2 \cdot n \log n$ bits stored outside section S , which are part of the encoding of the input solution. Assume by contradiction that fewer than $\alpha/2 \cdot n \log n$ bits are generated from section S . Then there exist two adversary graphs G_1^* and G_2^* with different 2-vertex-components where the sequence of bits that flows out of section S or into section S is the same. Thus the chip cannot distinguish between G_1^* and G_2^* and will produce the wrong answer for one of the graphs.

The verification process of the 2-vertex-components does not have to be finished by the time the $(r/3\alpha + 1)$ st input sequence is read. This input sequence (and all subsequent input sequences) consists of edges of the filler graph only. But at the time the $(r/3\alpha + 1)$ st input sequence is read, section S has to contain $\alpha/2r \cdot n \log n$ "free" bits; i.e., bits that can be used to store the edges of this input sequence.

Recall that k is the width of the circumscribing rectangle. Thus, at most k connections can cross each cut, and in one cycle at most $2k \log n$ bits can leave or enter section S ($k \log n$ bits for each side). Let t be the uniform delay between two successive input sequences. Altogether, the first $r/3\alpha$ input sequences generate $\alpha/2 \cdot n \log n$ bits initially stored in section S . Those bits either have to leave section S , or wait for the corresponding bits from outside section S before they can be reused. Thus, at the time the $(r/3\alpha + 1)$ st input sequence is read at most $2r/3\alpha \cdot k \cdot t \cdot \log n$ of the $\alpha/2 \cdot n \log n$ bits generated in section S can be reused. Section S contains only $\alpha/2 \cdot n \log n$ bits, and in order to have $\alpha/2r \cdot n \log n$ "free" bits to store the new input sequence we need:

$$\frac{\alpha}{2} n \log n - 2 \frac{r}{3\alpha} k t \log n + \frac{\alpha}{2r} n \log n \leq \frac{\alpha}{2} n \log n.$$

The only variable in the inequality that is not fixed is t (α and r are constants, and k is fixed by the network). When distributing the amount of time it takes to "free" $\alpha/2r \cdot n \log n$ bits evenly among the $r/3\alpha$ time intervals (each one of length t) elapsing between the reading of two input sequences, we obtain a lower bound on t . We need $\alpha/2r \cdot n \leq 2r/3\alpha \cdot k \cdot t$ in order to be able to read the $(r/3\alpha + 1)$ st input sequence, which gives $t = \Omega(n/k)$. Thus $T = \Omega(n^2/k)$, or $\Omega(e/k)$. This concludes the proof in the case when $2r/\alpha > 12r$.

Consider now the case when $12r \geq 2r/\alpha$. Then $\alpha \geq 1/6$ and section S contains $1/12r \cdot n$ PE's and can hold a total of $1/12 \cdot n \log n$ bits. In this case we need to read $2r$ input sequences to feed all the 2-vertex-components into section S . Since $1/12 \cdot n \log n < \alpha/2 \cdot n \log n$, more than half of the bits of the encoding of the input solution describing the connected components of the 2-vertex-components are stored outside section S . By an argument similar to the one used in the first case we obtain $T = \Omega(n^2/k)$, or $\Omega(e/k)$. \square

Our argument used the fact that the algorithm was where oblivious in a crucial way. One could dislike the idea of using where obliviousness in such an inimical fashion: one can easily agree with the idea of the chip's environment having control of where the input will be read, but one does not expect the environment to act as an adversary. The lower bound, however, does not depend on a possibly bad definition. Consider a more realistic model: the input given in the form of adjacency lists, with

the edges arriving into the input ports in lexicographic order. Now, clearly, the environment is not an adversary, yet the same bounds hold, as we show below.

THEOREM 3.4. *Let N be a network of $O(n)$ area that can be circumscribed by a rectangle of size $k \times n/k$, $1 \leq k \leq n^{1/2}$. Then solving the verification problem of the ccp on N with input given in the form of adjacency lists requires time $\Omega(n^2/k)$, or, expressed in terms of edges $\Omega(e/k)$.*

Proof. We will show how to find a section S and an adversary graph G^* so that all the edges representing 2-vertex-components are fed into section S in a constant number of input sequences. Again, as in the proof of Theorem 3.3, the filler edges will be used to fill up input sequences and are read by the PE's outside section S . One has to be more careful with the selection of the filler edges and the order in which all the edges are input, since the lexicographic ordering of all the edges has to be preserved.

Make the cuts through the network as in Theorem 3.3. Let S be the section containing $PE_{n/2}$. Let $i_1, i_2, \dots, i_{\delta n}$ be the indices of the PE's in section S , $\delta = \min(\alpha/2r, 1/12r)$. Thus section S contains $PE_{i_1}, \dots, PE_{n/2}, \dots, PE_{i_{\delta n}}$. In the first input sequence PE_{i_1} reads the edge $(2, x_1)$, PE_{i_2} the edge $(4, x_2), \dots, PE_{i_{\delta n}}$ reads the edge $(2\delta n, x_{\delta n})$. The edges $(2i, x_i)$ are the first δn edges of the adjacency list representing 2-vertex-components. See Fig. 3.3, where section S contains $PE_1, PE_5, PE_7, PE_{n/2-1}, PE_{n/2}, \dots$, and the input for the first $n/2$ PE's in the first input sequence is given. ($PE_i \leftarrow (x, y)$ stands for PE_i reads the edge (x, y) .)

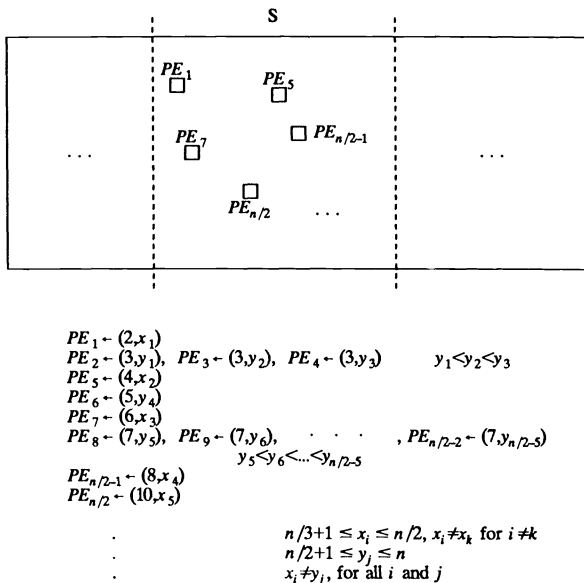


FIG. 3.3. Section S for input in the form of adjacency lists.

Let i_p and i_{p+1} be two consecutive indices of PE's in section S ; i.e., all PE_k with $i_p < k < i_{p+1}$ are outside section S , $1 \leq p \leq \delta n - 1$. If $i_p < i_{p+1} - 1$, $PE_{i_{p+1}}, \dots, PE_{i_{p+1}-1}$, which are the PE's outside section S , have to read edges of the filler graph. They read $i_{p+1} - i_p - 1$ edges adjacent to vertex $2p + 1$. (Recall PE_{i_p} reads the edge $(2p, x_p)$, and $PE_{i_{p+1}}$ reads the edge $(2(p + 1), x_{p+1})$.) Note that $i_{p+1} - i_p - 1 \leq n/2$. Hence, vertex $2p + 1$ of the adversary graph may need to be adjacent to $n/2$ vertices y_i with $y_i > 2p + 1$. The adversary graph allows us to supply the PE's outside section S with the right number of filler edges adjacent to vertex $2p + 1$.

If $i_p = i_{p+1} - 1$, vertex $2p + 1$ is adjacent to no other vertex. In this situation we actually need to change our definition of the class of adversary graphs. The filler graph consists then of a number of singleton vertices and one big connected component. Nevertheless, the same result holds. Note that PE_1, \dots, PE_{i_p-1} read the edges adjacent to vertex 1, and that $PE_{i_{p+1}}, \dots, PE_n$ read the edges adjacent to vertex $2\delta n + 1$.

The succeeding input sequences are formed in the same way: each time the next δn edges of the adjacency list representing 2-vertex-components are read into section S , while the PE's outside section S read the edges of the filler graph as described above.

After all the 2-vertex-components have been read into section S , the $\Omega(n^2)$ edges (i, x_i) of the filler graph, $i > n/3$, $i < x_i$, remain to be read. The same argument as used in the proof of Theorem 3.3 yields $T = \Omega(n^2/k)$ for chips of $O(n)$ area with input given in the form of adjacency lists. \square

3.3. Networks of $O(nm)$ area. The result of the previous theorems can be generalized to networks of $O(nm)$ area that have $O(np)$ PE's. The area determines the amount of information that can flow in and out of section S in one time pulse, the number of PE's determines the number of input sequences required to read n^2 inputs.

THEOREM 3.5. *Let N be a network of $O(nm)$ area containing $O(np)$ PE's, that can be circumscribed by a rectangle of size $k \times nm/k$, $1 \leq k \leq (nm)^{1/2}$, $m = o(n)$, $p = o(n)$, $m \geq p$. Then solving the verification problem of the ccp on N with input given in the form of edges requires time $\Omega(n^2/pk)$. When input is given in the form of adjacency lists the time required is $\Omega(n^2/(p^2k) + n/p)$.*

Proof. For chips containing $O(np)$ PE's, $p > o(1)$, it is no longer true in our model that an encoding of the solution on the chip requires $\alpha n \log n$ bits of the registers of the PE's. (The use of the indices of the PE's in the encoding or the use of the fact that some bits are not used in the encoding allows the chip to encode with fewer bits.)

When the graph G^* is given in the form of edges consider a section of the chip containing $\alpha/2r \cdot n$ PE's. Such a section S contains $\alpha/2 \cdot n \log n$ bits and by Lemma 3.2 at most half of the information about the encoding of the solution can be stored in S . The "other half" of the encoding is stored outside section S . Even when the "other half" is encoded by the PE's outside section S using $o(\alpha/2 \cdot n \log n)$ bits (using "unnatural" encodings), at least $\alpha/2 \cdot n \log n$ bits must cross the boundary of section S . Section S is obtained by making $\max(2pr/\alpha - 1, 12pr - 1)$ cuts through the network so that each section between two cuts contains $\min(\alpha/2r \cdot n, 1/12r \cdot n)$ PE's. Input all the edges representing 2-vertex-components into section S in a constant number of input sequences. Choose G^* as in the proof of Theorem 3.3 and obtain $T = \Omega(n^2/np \cdot n/k) = \Omega(n^2/pk)$, $1 \leq k \leq (nm)^{1/2}$, when the input is given in the form of edges.

Consider now the case when the graph G^* is given in the form of adjacency lists. Again, as in the proof of Theorem 3.4, it must be possible to feed the PE's outside a section S with filler edges while preserving the lexicographic ordering of the input edges. Let PE_{i_k} and $PE_{i_{k+1}}$ be two succeeding PE's in section S , $i_k < i_{k+1}$ and for no i_p , $i_k < i_p < i_{k+1}$, PE_{i_p} is in section S . In Theorem 3.4 we never needed to fill in more than $n/2$ edges adjacent to one vertex between PE_{i_k} and $PE_{i_{k+1}}$. Assume $PE_{np/2}$ to be in section S , then we must now be able to fill in $np/2$ edges between PE_{i_k} and $PE_{i_{k+1}}$. Since it is not possible to have $np/2$ edges adjacent to one vertex, we define the modified adversary graph G_m^* .

G_m^* consists of $n/6p$ 2-vertex-components and a filler graph on $n - 2n/6p = n(1 - 1/3p)$ vertices. $E_m^* = \{(i(p+1), x_i) \mid 1 \leq i \leq n/6p, n(p+1)/6p < x_i \leq n\} \cup E'$. Let V_x be the set of vertices greater than $n(p+1)/6p$ not used in the 2-vertex components.

Then $E' \subseteq \{((i-1)p+i+j, x_{ij}) \mid 1 \leq i \leq n/6p, 0 \leq j \leq p-1, x_{ij} \in V_x\} \cup V_x \times V_x$. Between two succeeding 2-vertex-components $(i(p+1), x_i)$ and $((i+1)(p+1), x_{i+1})$ in the adjacency list we can have the edges adjacent to $(i+1)(p+1) - i(p+1) - 1 = p$ vertices. This allows us to fill up the input sequences correctly (each one of the p vertices can be adjacent to at least $n/2$ vertices with higher index not used in the 2-vertex-components). It can be shown that $\alpha/p \cdot n \log n$ bits are needed for the encoding of the 2-vertex-components of G_m^* .

Make cuts through the network, so that each section between two cuts contains $\alpha/2pr \cdot n$ PE's. Choose as section S the section containing $PE_{np/2}$. By applying the ideas of Theorems 3.3 and 3.4 we obtain $t = \Omega(n/kp)$. Since n/p time pulses are needed to read all the input sequences $T = \Omega(n^2/np \cdot n/kp + n/p) = \Omega(n^2/p^2k + n/p)$, which proves Theorem 3.5. \square

Note that for $p > O(1)$ the lower bound achieved for input in the form of adjacency lists is weaker than the one for edges. But we do not know of an algorithm that achieves a better time when the input is in the form of adjacency lists than when input is in the form of edges.

The reader may want to compare our results with those reported in [J], where a different technique, based on "mutual information transmission" arguments (as in [JK] and [Y]), was used to obtain lower bounds for the ccp on a different VLSI model. The important difference between the models used in [J] and ours is that the model in [J] assumes that the information about the previous input sequence is completely processed at the time a new input sequence is read. Under this assumption our lower bounds hold also for input in the form of an adjacency matrix. In our model, a different approach is necessary for a lower bound proof for input in the form of an adjacency matrix. In [H1] we show that verification for input in the form of an adjacency matrix can be done in $O(n)$ time on a binary tree network of $O(n)$ area.

3.4. Verifying is no harder than computing. In a sequential model of computation, verification of problems with unique solutions is never harder than computing, since one can always verify by computing the actual solution and comparing it with the given input solution. In a parallel setting, this argument does not hold: both the input solution and the computed solution will be scattered among different processing elements. Nevertheless, we show that the lower bounds obtained for the verification problem of the ccp hold also for the ccp; i.e., computing the connected components is at least as hard as verifying them. For all networks solving the ccp we assume that the solution describing the connected components can be stored entirely within the network; i.e., length of the solution $\leq A \cdot \log n$, where A is the area of the network. This seems to be a reasonable assumption. There are graphs where the connected components cannot be determined until all the input has been read, and a description of the connected components formed so far has to be stored in the network in order to solve the ccp correctly. For example, networks of $O(n)$ area that output the connected components in form of an $n \times n$ adjacency matrix have to keep a description of the connected components of length $O(n \log n)$ bits in the network before generating the matrix.

Assume network N computes the connected components. We show how to construct a new network N_v that verifies the connected components and whose time and area requirements differ from network N only by a constant factor. Network N_v uses N as a subnetwork.

THEOREM 3.6. *Let N be a network of area A , $A = \Omega(n)$, that solves the ccp in time T . Then there exists a network N_v of area $\Theta(A)$ solving the verification problem of the ccp in time $\Theta(T)$.*

Proof. Network N has area A and produces the solution to the ccp in form of an encoding E in time T . Since $A = \Omega(n)$ we can assume that one PE produces at most $\log n$ bits of the solution.

Let $\mathcal{N}(A, T, E_v)$ be the class of networks solving the verification problem of the ccp in time $O(T)$ on a network of area $O(A)$ when the input solution is given in the form of the encoding E_v . We show that this class is not empty and outline how to construct a network $N_v \in \mathcal{N}$ that uses network N as a subnetwork.

Network N_v contains the same number of PE's and the same interconnections as network N , and each PE contains an additional register. Let the encoding of the solution read by the network N_v be of the same form as the encoding of the solution generated by the network N ; i.e., $E_v = E$. Input the bits of E into those PE's of the network N that will contain the corresponding output bits of the solution of the ccp generated by the algorithm for the ccp. After storing the input solution in the additional register run the algorithm for the ccp. When all the output bits have been generated by the algorithm compare in each PE the generated encoding with the encoding given as input and set a flag in the PE in case of inequality.

Assume w.l.o.g. that PE_1 produces the 0/1 answer to the verification problem. Then PE_1 produces a 0 if one (or more) flags have been set to 0. Let each PE that contains a set flag send the value of its flag to PE_1 . If two or more flags meet at a PE on the path to PE_1 , the "and" of all the incoming flags is produced and send to PE_1 . (Since each PE is connected only to a constant number of other PE's, this step takes constant time.) It can easily be shown that in any network N_v , where each PE is needed at some point during the computation, the number of steps required to generate the connected components is larger than the longest shortest distance between any two PE's. Thus the area (and the time) to determine the answer to the verification problem does not increase the time (and area) needed to solve the ccp by more than a constant factor, and the theorem follows. \square

3.5. Approximate verification. We show that the lower bounds proven hold even if the verification is allowed to be approximate: given as input an encoding of the solution and a graph G the answer to the verification problem is 1 if and only if G contains at most δn vertices that are in different connected components than the ones specified in the solution (i.e., a 1 is produced if the connected components of G are "close" to the ones of the given solution). For input in the form of edges we have $0 \leq \delta < 1$, and for input in the form of adjacency lists we have $0 \leq \delta < 2/3$. We present the lower bound proof for networks of $O(n)$ area (the extension for networks of $O(nm)$ area is straightforward).

In the proof we consider first the adversary graphs G^* containing $n/6$ 2-vertex-components as defined in § 3.1, and assume $\delta < 1/3$. We then outline how to modify the adversary graphs so that the lower bounds hold for $\delta \geq 1/3$. For each adversary graph G_i^* define a class of approximate graphs H_i^* as follows. Let V_i^* be the set of vertices used in the 2-vertex-components of G_i^* . Then $H_i^* = \{G_i^j \mid G_i^j \text{ contains at most } \delta n \text{ vertices } x_k, x_k \in V_i^*, \text{ in different connected components as in } G_i^*\}$. Thus the connected components of $G_i^j \in H_i^*$ differ from the ones of G_i^* only in the 2-vertex-components.

THEOREM 3.7. *Solving the approximate verification problem of the ccp on a network of $O(n)$ area, which can be circumscribed by a rectangle of size $k \times n/k$, $1 \leq k \leq n^{1/2}$, requires time $\Omega(n^2/k)$ when the input is given in the form of edges or adjacency lists.*

Proof. Assume G_i^* is an adversary graph with $n/6$ 2-vertex-components and $\delta < 1/3$. Graph G_i^j contains at most δn vertices that are in 2-vertex-components in G_i^* , and in different connected components than in G_i^* . Thus G_i^j and G_i^* agree on

at least $n/6 - \delta/2 \cdot n = n(1/6 - \delta/2)$ 2-vertex-components. Let $\lambda = (1/6 - \delta/2)$. Then there exist G_i^j and G_i^* such that an encoding of the λn 2-vertex-components contained in both graphs requires $\lambda' n \log n$ bits, for some constant λ' . Make a constant number of cuts through the chip so that each section contains $\lambda'/2 \cdot n \log n$ bits. Choose an appropriate section S and input all the λn 2-vertex-components that are in G_i^j as well as G_i^* into section S in a constant number of input sequences. Again, as in the proof of Theorems 3.3 and 3.4, at least half of the bits of the encoding describing the input solution (i.e., the connected components of G_i^j) of the λn 2-vertex-components are stored outside section S . Using the same technique as in the proof of Theorems 3.3 and 3.4 we obtain $T = \Omega(n^2/k)$ when $\delta < 1/3$.

When using adversary graphs with $n/6$ 2-vertex-components we need $\delta < 1/3$ in order to have enough 2-vertex-components that are in G_i^j as well as in G_i^* . We have to modify the adversary graphs to achieve the same lower bounds for $\delta \geq 1/3$. Consider first input in the form of edges. Each graph consisting of $k/(2k+1) \cdot n$ 2-vertex-components and a filler graph of $1/(2k+1) \cdot n$ vertices can be used as an adversary graph, $k \geq 1$. As long as $\delta < 2k/(2k+1)$ we obtain the same lower bounds, and for large enough k we can get arbitrarily close to 1.

When the input is given in the form of adjacency lists a slightly more complicated adversary graph is needed. The filler graph has to contain edges that can be interleaved with the 2-vertex-components in order to maintain the lexicographic ordering. (Recall the odd/even construction in § 3.1 and how it is used in Theorem 3.4.) Furthermore the filler graph has to contain $\Omega(n)$ vertices not used in the interleaving process. A graph consisting of $k/(3k+1) \cdot n$ 2-vertex-components, which has $k/(3k+1) \cdot n$ vertices in the filler graph used in the interleaving process and $1/(3k+1) \cdot n$ vertices in the filler graph used to make up a large number of input sequences, can also be used as an adversary graph, $k \geq 1$. As long as $\delta < 2k/(3k+1)$ we obtain the same lower bounds, and for large enough k we can get arbitrarily close to $2/3$. \square

4. An optimal linear array algorithm. In § 3.2 we showed that a lower bound on the time needed to solve the ccp with a when oblivious algorithm on a network of $O(n)$ area with constant width is $\Omega(n^2)$, or, in terms of the number of edges in the graph, $\Omega(e)$. This lower bound can be achieved: We present an algorithm solving the ccp on a linear array of n PE's in $O(e)$ time when the input is given in the form of adjacency lists or edges. The algorithm produces output in the form of a vector C of size n . C_i , the *component number* of vertex i , is the smallest vertex in the connected component containing vertex i , $i \in \{1, 2, \dots, n\}$. The basic idea of the algorithm is similar to the ones used in ([H1], [LV], [S]).

In the linear array PE_i is connected to PE_{i-1} and PE_{i+1} , provided they exist, $1 \leq i \leq n$. Each PE contains 7 registers: Register C of PE_i holds the current component number of vertex i ; register X and Y hold initially an input edge (x, y) , later the current component number of vertex x and vertex y , respectively. The remaining 4 registers are used as auxiliary registers during the algorithm. The network is initialized with every vertex being a component by itself: i.e., the value of register C of PE_i is initialized to i .

After reading an input sequence of n edges, each PE contains an edge (x, y) . Algorithm CONNECT-LA then determines the current component number of the vertices x and y . Edges with $C_x = C_y$ are deleted. If $C_x \neq C_y$ and $C_x < C_y$, the algorithm merges the components C_x and C_y by replacing every occurrence of C_y by C_x . We denote this by $C_y \rightarrow C_x$ and call it a *change* (similar for $C_x > C_y$). In order to merge components simultaneously we need to update the change in PE_i before sending it

through the linear array, $1 \leq i \leq n$; i.e., the component number of the vertices x and y in PE_i is updated to the component number the vertices x and y have after the changes in PE_1, \dots, PE_{i-1} have been processed.

Algorithm CONNECT-LA avoids problems of congestion on the communication paths by letting all PE's of the linear array send information either to the PE to the right or to the PE to the left.

Since parallel algorithms are harder to read and understand than sequential ones, the algorithms should be designed such that one gets an understanding of what goes on in the algorithm, at a sufficiently high level of abstraction, without getting lost in implementation details, such as the exact information flow in the network. The following algorithm is written as a sequence of *basic building blocks* (bbb's), where each bbb can be thought of as a subroutine call. The primitive bbb's are a parallel assignment statement and information transmission statements.

Each bbb selects a subset of the PE's of the network to execute the next action, and specifies the type of action to be executed. A program written as a sequence of basic building blocks looks very much like a sequential program. We give a very informal description of the bbb's needed in the algorithm, and refer to [H1] for details and more complex bbb's.

A bbb is either a *condition* followed by a *command* and its *arguments*, or a *condition* followed by the delimiter **begin**, a number of bbb's and the delimiter **end**. The condition selects the PE's of the network that will execute the statements between the **begin** and **end** delimiter of the bbb, or the single command that follows the condition. Its function is similar to the boolean guards of Dijkstra's guarded commands [D]. Each PE has a unique *processor number* (PEnr), which is known to itself. (We also assume that each PE knows the PEnr of the PE's to which it is directly connected.) The condition selects PE's by describing the range of their PEnr, but it may also contain boolean expressions containing register names of the PE's; (e.g., $(1 \leq PEnr \leq n) \ \& \ (X = Y)$ selects all PE's with a PEnr between 1 and n where register X and Y have the same value).

The SET command is the parallel assignment statement. For example, $(1 \leq PEnr \leq n)$ SET $((X, Y), (0, 0))$ sets in each PE with PEnr between 1 and n registers X and Y to 0. In the following algorithm we only need a simple form of information transmission: sending the content of registers of a PE to its right or left neighbor PE. For example, in $(1 \leq PEnr \leq n-1)$ SEND $((X, Y), PEnr+1)$ the PE's with the PEnr between 1 and $n-1$ send the content of register X and Y to their right neighbor, and it is stored there in register X and Y . The DCL command used in the algorithm is a nonexecutable command, which lists the registers of the PE's used during the algorithm.

Algorithm CONNECT-LA;

```
//CONNECT-LA solves the ccp on a linear array of  $n$  PE's; register  $C$  holds the
//current component number, registers  $X$  and  $Y$  hold the current input edge //
 $(1 \leq PEnr \leq n)$  DCL ( $C, X, Y$ );
1. Initialization
   //every vertex is a component by itself //
    $(1 \leq PEnr \leq n)$  SET ( $C, PEnr$ );
2. repeat until all edges have been read
   2.1. //read the next  $n$  edges //
        $(1 \leq PEnr \leq n)$  READ ( $X, Y$ );
   2.2. //determine the current component number of the vertices //
       //and the changes //
       DET-COMPNR-LA;
```

```

//remove the edges that do not merge components and let //
//register  $X$  contain the larger of the two component numbers //
( $1 \leq PEnr \leq n$ ) begin
  ( $X = Y$ ) SET (( $X, Y$ ), (0, 0));
  ( $X < Y$ ) SET (( $X, Y$ ), ( $Y, X$ ));
  end;
2.3. //update the changes //
  RIPPLE (( $X, Y$ ),  $1 \leq PEnr \leq n$ );
2.4. //merge components //
  MERGE-LA;
endrepeat
end CONNECT-LA;

```

Algorithm DET-COMPNER-LA determines the current component number of vertex X and vertex Y . The current component number of each vertex is sent through the linear array, and a PE containing vertex k records the current component number of vertex k when it passes through.

```

Algorithm DET-COMPNER-LA;
//each PE sends the current component number of the vertex associated //
//with the PE to all the other PEs in the linear array //
( $1 \leq PEnr \leq n$ ) begin
  DCL ( $C, X, Y$ );
  DCL LOCAL ( $FX, FCX, BX, BCX$ );
  end;
//store each vertex and its current component number in the auxiliary //
//registers; the values in ( $BX, BCX$ ) get sent backwards, the values //
//in ( $FX, FCX$ ) get sent forwards //
( $1 \leq PEnr \leq n$ ) begin
  SET (( $FX, FCX$ ), ( $PEnr, C$ ));
  SET (( $BX, BCX$ ), ( $PEnr, C$ ));
  end;
for  $i = 1$  to  $n$  do
  ( $1 \leq PEnr \leq n$ ) begin
    ( $X = BX$ ) SET ( $X, BCX$ ); //record the current component //
    ( $X = FX$ ) SET ( $X, FCX$ ); //number of the vertex when the //
    ( $Y = BX$ ) SET ( $Y, BCX$ ); //appropriate entry passes through //
    ( $Y = FX$ ) SET ( $Y, FCX$ );
    end;
  //send the entries forwards and backwards in the linear array //
  ( $1 \leq PEnr \leq n - 1$ ) SEND (( $FX, FCX$ ),  $PEnr + 1$ );
  ( $2 \leq PEnr \leq n$ ) SEND (( $BX, BCX$ ),  $PEnr - 1$ );
endfor;
end DET-COMPNER-LA;

```

After each PE knows the current component number of the vertices of its input edge, it decides whether or not it contains a change. In order to send the changes through the network and merge components simultaneously, the changes have to be updated first: When merging the components the i th change $x \rightarrow y$ of PE_i will arrive at all PE's after the changes $1, \dots, i-1$. Thus if the changes $1, \dots, i-1$ change the component number of vertex x to x' , $x' < x$, we need to update in change i x to x' ,

before sending out change i . (Similar for vertex y .) The updating is done by letting the changes “ripple” to the right in the linear array and is described in Algorithm RIPPLE. For example, the changes $3 \rightarrow 2$, $4 \rightarrow 3$, $5 \rightarrow 4$, $5 \rightarrow 2$, $2 \rightarrow 1$, and $5 \rightarrow 3$ stored in PE_1 , PE_2 , PE_3 , PE_4 , PE_5 , and PE_6 , respectively, are updated to $3 \rightarrow 2$, $4 \rightarrow 2$, $5 \rightarrow 2$, $2 \rightarrow 2$, $2 \rightarrow 1$, and $1 \rightarrow 1$. More details about the RIPPLE routine can be found in [H2].

```

//Update X and Y of the ith change to the index of the component //
//X and Y are in after the i-1 preceding changes have been processed //
//let U and V be two auxiliary registers //
Algorithm RIPPLE ((X, Y),  $1 \leq PEnr \leq s$ ); ( $1 \leq PEnr \leq n$ ) begin
  DCL (X, Y);
  DCL LOCAL (U, V);
  end;
( $1 \leq PEnr \leq n$ ) SET ((U, V), (X, Y)); for  $i = 1$  to  $n$  do
  //the ith change is updated, continue updating: change  $i + 1, \dots, n$  //
  ( $1 \leq PEnr \leq n - 1$ ) SEND ((U, V),  $PEnr + 1$ ); //
  ( $i + 1 \leq PEnr \leq n$ ) begin
    //the jth change merges two components already //
    //merged by the preceding changes //
    case of:
      ( $U = X$ ) & ( $V = Y$ ) SET (X, Y) //change  $X \rightarrow Y$  into  $Y \rightarrow Y$  //
      //in the changes  $j + 1, \dots, n$  it is already recorded that X //
      //and Y get merged; every occurrence of X is changes into Y //
      ( $V = X$ ) SET (V, Y) //change  $U \rightarrow V$  into  $U \rightarrow Y$  //
      ( $V = Y$ ) NULL;
      ( $U = Y$ ) SET (Y, V) //change  $X \rightarrow Y$  into  $X \rightarrow V$  //
      ( $U = X$ ) & ( $V > Y$ ) SET ((U, V, X, Y), (V, Y, V, Y)) //
      //change  $U \rightarrow V$  and  $X \rightarrow Y$  into  $V \rightarrow Y$  //
      ( $U = X$ ) & ( $V < Y$ ) SET ((U, V, X, Y), (Y, V, Y, V)) //
      //change  $U \rightarrow V$  and  $X \rightarrow Y$  into  $Y \rightarrow V$  //
    end case;
  end; endfor; end RIPPLE;

```

After the updating the changes are sent out to all the PE's in the network such that change i arrives at each PE after change $i - 1$, and we merge the components. Assume change $x \rightarrow y$ arrives at PE_j . If the content of register C of PE_j equals x , it gets changed to y .

Algorithm MERGE-LA;

```

//send the ith change from  $PE_i$  to  $PE_1$ ; activate change  $i$  when it //
//arrives at  $PE_1$  and send it through the linear array to merge components //
( $1 \leq PEnr \leq n$ ) begin
  DCL (C, X, Y);
  DCL LOCAL (AX, AY);
  end;
( $1 \leq PEnr \leq n$ ) SET ((AX, AY), (0, 0));
for  $i = 1$  to  $2n$  do
  2.1. //activate the change currently at  $PE_i$  //
      ( $PEnr = 1$ ) & ( $i \leq n$ ) SET ((AX, AY), (X, Y)); //
  2.2. //test if a component number needs to get changed and //
      //send the activated change to the next PE //

```



```

(1 ≤ PEnr ≤ n) & (AX ≠ 0) begin
  (C = AX) SET (C, AY);
  SEND ((AX, AY), PEnr + 1);
end;
2.3. //send the changes not yet activated towards PE1 //
      (2 ≤ PEnr ≤ n - i + 1) SEND ((X, Y), PEnr - 1);
endfor;
end MERGE-LA;

```

THEOREM 4.1. *Algorithm CONNECT-LA solves the ccp in time $O(e)$ on a linear array of n PE's when the graph is given in the form of adjacency lists or edges, which is optimal.*

Proof. When Algorithm CONNECT-LA reads the i th input sequence it has computed the connected components given by the previously read $(i-1)n$ edges. It merged two components C' and C'' if and only if it had read an edge (x, y) such that $C_x = C'$ and $C_y = C''$ right before the merge was performed. Thus Algorithm CONNECT-LA computes the connected components correctly.

Each one of the steps 2.1–2.5 of the algorithm can be done in $O(n)$ time. It can be shown by induction that Algorithm RIPPLE correctly updates the changes: i and j of the change $i \rightarrow j$ of PE_k are updated to the component number of the component they are in after the $k-1$ preceding changes have been performed. Since at most e/n input sequences are read, and the time between the reading of two input sequences is $O(n)$, the $O(e)$ time bound follows. \square

Acknowledgments. We would like to thank Helmut Alt, Thomas Lengauer, and Joseph Ja'Ja' for reading the early version of this paper and for making valuable suggestions, and one referee for his helpful comments.

REFERENCES

- [AA] H. ABELSON AND P. ANDREA, *Information transfer and area-time tradeoffs for VLSI multiplication*, Comm. ACM, 23 (1980), pp. 20–23.
- [AK] M. J. ATALLAH AND S. R. KOSARAJU, *Graph problems on a mesh-connected processor array*, Proc. 14th Annual ACM Symposium on Theory of Computing, 1982, pp. 345–353.
- [BK] R. P. BRENT AND H. T. KUNG, *The area-time complexity of binary multiplication*, J. Assoc. Comput. Mach., 28 (1981), pp. 521–534.
- [CM] B. CHAZELLE AND L. MONIER, *A model of computation for VLSI with related complexity results*, Proc. 13th Annual ACM Symposium on Theory of Computing, 1981, pp. 318–325.
- [D] E. W. DIJKSTRA, *Guarded commands, nondeterminacy and formal derivation of programs*, Comm. ACM, 18 (1979), pp. 453–457.
- [GKT] L. J. GUIBAS, H. T. KUNG AND C. D. THOMPSON, *Direct VLSI implementations for combinatorial algorithms*, Proc. Conference on VLSI Technology, Design and Fabrication, California Institute of Technology, Pasadena, 1979, pp. 509–525.
- [H1] S. E. HAMBRUSCH, *The complexity of graph problems on VLSI*, Ph.D. thesis, Pennsylvania State Univ., University Park, 1982.
- [H2] ———, *VLSI algorithms for the connected components problem*, this Journal, 12 (1983), pp. 354–365.
- [J] J. JA'JA', *The VLSI complexity of graph problems*, Tech. Rep. CS81-25, Pennsylvania State Univ., University Park, October 1981.
- [JK] J. JA'JA' AND V. K. KUMAR, *Information transfer in distributed computing with applications to VLSI*, J. Assoc. Comput. Mach., 31 (1984), pp. 150–162.
- [KL] H. T. KUNG AND C. E. LEISERSON, *Systolic arrays for VLSI*, in Introduction to VLSI Systems, C. Mead and L. Conway, eds., Addison-Wesley, Reading, MA, 1980, pp. 260–292.
- [LS] R. J. LIPTON AND R. S. SEDGEWICK, *Lower Bounds for VLSI*, Proc. 13th Annual ACM Symposium on Theory of Computing, 1981, pp. 300–307.

- [LV] R. J. LIPTON AND J. VALDES, *Census function: An approach to VLSI upper bounds*, Proc. 22nd Annual Symposium on Foundations of Computer Science, 1981, pp. 13-22.
- [MC] C. MEAD AND L. CONWAY, *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA, 1980.
- [NMB] D. NATH, S. N. MAHESHWARI AND P. C. P. BHATT, *Efficient VLSI networks for parallel processing based on orthogonal trees*, IEEE Trans. Comput., C-32 (1983), pp. 569-581.
- [P] W. PAUL, *On heads versus tapes*, Proc. 22nd Annual Symposium on Foundations of Computer Science, 1981, pp. 68-73.
- [PSS] W. PAUL, J. SEIFERAS AND J. SIMON, *An information theoretic approach to time bounds on on-line computation*, Proc. 12th Annual ACM Symposium on Theory of Computing, 1980, pp. 357-367.
- [PV] F. P. PREPARATA AND J. E. VUILLEMIN, *The cube connected cycles: a versatile network for parallel computation*, Comm. ACM, 24 (1981), pp. 300-309.
- [S] C. SAVAGE, *A systolic data structure chip for connectivity problems*, Proc. CMU Conference on VLSI Systems and Computing, 1981, pp. 296-300.
- [T] C. D. THOMPSON, *Area-time complexity for VLSI*, Proc. 11th Annual ACM Symposium on Theory of Computing, 1979, pp. 81-88.
- [V] J. E. VUILLEMIN, *A combinatorial limit to the computing power of VLSI circuits*, IEEE Trans. Comput., C-32 (1983), pp. 294-300.
- [Y] A. YAO, *Some complexity questions related to distributed computing*, Proc. 11th Annual ACM Symposium on Theory of Computing, 1979, pp. 209-213.

BIASED SEARCH TREES*

SAMUEL W. BENT†, DANIEL D. SLEATOR‡ AND ROBERT E. TARJAN‡

Abstract. We consider the problem of storing items from a totally ordered set in a search tree so that the access time for a given item depends on a known estimate of the access frequency of the item. We describe two related classes of *biased search trees* whose average access time is within a constant factor of the minimum and that are easy to update under insertions, deletions and more radical update operations. We present and analyze efficient update algorithms for biased search trees. We list several applications of such trees.

Key words. optimum search trees, binary search trees, data structures, algorithms, complexity

1. Introduction. The following problem, which we shall call the *dictionary problem*, occurs frequently in computer science. We are given a totally ordered universe U . We wish to represent sets $S \subseteq U$ in such a way that the following access and update operations are efficient:

- access* (i, S): If item i is an element of set S , return a pointer to its location. Otherwise return a special **null** pointer.
- insert* (i, S): Insert item i into set S . We allow an insertion to take place only if i is not initially an element of S .
- delete* (i, S): Delete item i from set S . We allow a deletion only if i is initially in S .

In addition to *access*, *insert* and *delete*, the following more radical update operations are often useful:

- join* (R, S): (two-way join). Return the set consisting of the union of R and S . This operation destroys R and S , and is allowed only if every item in R is less than every item in S . (Thus a join can be regarded as the concatenation of the sorted sets R and S .)
- join* (R, i, S): (three-way join). Return the set consisting of the union of R , $\{i\}$ and S . This operation destroys R and S , and is allowed only if every item in R is less than i and every item in S is greater than i .
- split* (i, S): Split S into three sets: P , containing all items in S less than i ; Q , containing i if $i \in S$ (three-way split) and nothing if $i \notin S$ (two-way split); and R , containing all elements in S greater than i . This operation destroys S .

One kind of data structure that efficiently supports these operations is a *search tree*. A search tree is an ordered tree (Appendix A contains our tree terminology) containing the items of a set in its leaves, in left-to-right order, one item per leaf. In order to facilitate the access operation, we must also store auxiliary items, called *keys*, in the internal nodes. (Appendix B discusses the placement, use and updating of keys.) To access an item, we start at the root of the tree, compare the item being accessed with the key(s) in this node, go to a child determined by the outcome of the comparison(s), and continue in this way until reaching a leaf. This leaf contains the item if it is in the tree. With this access method, the time to access an item is proportional

* Received by the editors March 15, 1983, and in revised form March 7, 1984. Research partially supported by the National Science Foundation under grant MCS 82-03238.

† Computer Science Department, University of Wisconsin, Madison, Wisconsin 53706.

‡ AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

to the depth of the leaf containing it, if we assume a fixed upper bound on the degree of a node.

If the search tree satisfies an appropriate balance condition, then the height of the tree, and hence the worst-case access time, is $O(\log n)^1$, where n is the number of items in the set. For a comparison-based model of computation, one can prove a lower bound of $\Omega(\log n)$ on the worst-case access time; thus balanced trees have a worst-case access time within a constant factor of the minimum. Kinds of balanced trees include height-balanced trees [2], [20], two-three trees [3], B -trees [6], weight-balanced trees [27], red-black trees [12] and many others. These kinds of trees all have the additional property that each of the update operations can be carried out in $O(\log n)$ time.

In many applications of search trees, the access frequencies are different for different items. In such a situation we would like to *bias* the search tree, so that the more-frequently needed items can be accessed faster than the less-frequently needed ones. In order to treat this problem formally let us assume that each item i has a known *weight* $w_i > 0$ representing the access frequency. A measure of the average access time is

$$\sum_{i \in S} w_i \frac{(d_i + 1)}{W},$$

where $W = \sum_{i \in S} w_i$ is the sum of the weights of the items in the set and d_i is the depth in the search tree of the node containing item i . Our goal is to make the total weighted depth $\sum_{i \in S} w_i d_i$ as small as possible while preserving the ability to update the search tree rapidly. We call this the *biased dictionary problem*. It is natural in this problem to allow an additional operation for changing the weight of an item:

reweight (i, w, S): Redefine the weight of item i in set S to be w .

In this paper we shall propose two kinds of trees, which we call *biased search trees*, for solving the biased dictionary problem. Our results provide not only specific kinds of search trees, but also a general methodology for converting almost any class of balanced search trees into a similar but more general class of biased search trees. In addition to developing new ideas, the paper extends and refines ideas first presented earlier in preliminary form [7], [8].

The paper contains five sections. Section 2 reviews relevant previous work. Sections 3 and 4 define and analyze the properties of two kinds of biased search trees. Section 3 describes *biased 2, b trees*, which are analogous to 2, 3 trees and B -trees. Section 4 describes *biased binary trees*, which generalize a particular kind of red-black tree [12] sometimes called a symmetric binary B -tree [5]. Section 5 summarizes our results and discusses several applications and related work. Appendix A contains our tree terminology. Appendix B describes the arrangement of the keys in a search tree, their use for search and their updating.

2. Previous research. Several known results bear on the biased dictionary problem. The first and most important is a standard theorem of information theory.

THEOREM A [1]. *Consider any search tree T for a set S . If every node of T has at most b children, then the total weighted depth $\sum_{i \in S} w_i d_i$ is at least $W \sum_{i \in S} p_i \log_b (1/p_i)$, where $p_i = w_i/W$.*

In light of Theorem A, our goal is to devise classes of search trees with the property that $d_i = O(\log(W/w_i))$ for each item i , since any such search tree has minimum average

¹ If f and g are functions of a nonnegative real number x , we write " $f(x)$ is $O(g(x))$ " if there are positive constants c_1 and c_2 such that $f(x) \leq c_1 g(x) + c_2$ for all x . We write " $f(x)$ is $\Omega(g(x))$ " if $g(x)$ is $O(f(x))$.

access time to within a constant factor. We shall call $O(\log(W/w_i))$ the *ideal access time* of item i . If all item weights are equal the ideal access time for every item is $O(\log n)$ and any balanced search tree has ideal access time for all items.

Much previous work deals with the case of a *static* search tree. Suppose we are given a fixed set S whose items have known weights and we want to construct a search tree of exactly minimum total weighted depth. We call such a search tree *optimum*. Knuth [19] (see also Yao [33]) has given a dynamic programming algorithm that computes an optimum search tree among binary trees containing one item per internal node (instead of one item per leaf). Knuth's algorithm runs in $O(n^2)$ time and allows for the possibility that weights are given not only for the items in the set but also for the gaps between items; these gaps correspond to the possible ways to search for an item not in the set.

An algorithm of Hu and Tucker [17] (see also Garsia and Wachs [11], Hu, Kleitman and Tamaki [16], and Hu [15]) finds an optimum search tree among binary trees with one item per leaf, assuming no weights are given for the gaps. (This is a special case of the problem solved by Knuth.) The Hu-Tucker algorithm runs in $O(n \log n)$ time and resembles Huffman's algorithm [18] for computing an optimum binary prefix code. Fredman [10], Mehlhorn [24] and Korsch [21] have proposed $O(n)$ -time algorithms that construct binary search trees whose total weighted depth is within a constant factor of minimum.

None of these algorithms is satisfactory in the dynamic case, since they require completely restructuring the tree (spending $\Omega(n)$ time) each time an update occurs. Several authors have proposed classes of biased search trees that are easier to update. Baer [4] gave a heuristic for rebalancing biased weight-balanced search trees, but he gave no theoretical results, and indeed his trees do not have ideal access time in the worst case. Unterauer [32] described a class of biased weight-balanced trees that have ideal access time, but he did not analyze the worst-case time required for updates. Mehlhorn [26] described a class of biased search trees based on weight-balanced trees, called *D-trees*. D-trees have ideal worst-case access time and require $O(\min\{n, \log(W'/w_0)\})$ time for insertion, where w_0 is the weight of the smallest item and W' is the total weight of all the items in the set after the insertion. Mehlhorn [25] subsequently showed that a weight change in a D-tree can be performed in $O(\log(\max\{W, W'\}/\min\{w_i, w'_i\}))$ time, where w_i, w'_i, W, W' are the old weight of the reweighted item, the new weight of this item, the old total weight and the new total weight, respectively. Kriegel and Vaishnavi [22] proposed another version of biased search trees with time bounds similar to those of Mehlhorn.

The kinds of biased search trees we propose here are simpler than those proposed by Mehlhorn and Kriegel and Vaishnavi. They also have faster running times for insertion, deletion, join and split. (Mehlhorn does not consider join and split; Kriegel and Vaishnavi's bound for split is the same as ours but their bound for join is worse.)

3. Biased 2, b trees. Our first class of biased search trees generalizes 2, 3 trees [3]. A *2, b tree* is a search tree each of whose internal nodes has at least 2 and at most b children, where b is any fixed integer greater than two. We define the *rank* $s(x)$ of a node x in a 2, b tree recursively by

$$s(x) = \begin{cases} \lfloor \lg w_i \rfloor & \text{if } x \text{ is a leaf containing item } i,^2 \\ 1 + \max\{s(y) \mid y \text{ is a child of } x\} & \text{if } x \text{ is an internal node.} \end{cases}$$

² We denote $\log_2 x$ by $\lg x$.

This definition implies that if y is a node other than the tree root and x is its parent, then $s(y) \leq s(x) - 1$. We call y *major* if $s(y) = s(x) - 1$ and *minor* if $s(y) < s(x) - 1$. By convention the root is major. A *locally biased 2, b tree* is a 2, b tree satisfying the following property, which constrains the environment of minor nodes (see Fig. 1).

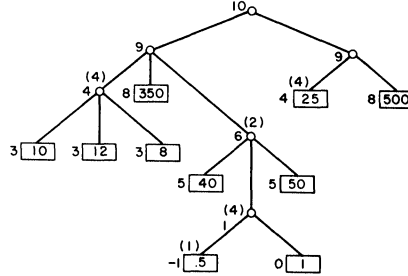


FIG. 1. A locally biased 2, 3 tree. Leaves are rectangles, internal nodes are circles. Numbers inside leaves are weights. Numbers to the left of nodes are ranks; those above nodes (in parentheses) are credit counts. For clarity the items in nodes are omitted.

Local bias. Any neighboring sibling of a minor node is a major leaf. (We say the tree is *locally biased* at the minor node.)

A 2, b tree is *balanced* if every leaf has the same depth. (Our definition of a balanced 2, 3 tree coincides with the usual definition of a 2, 3 tree.) In the case of equal-weight items, a 2, b tree is balanced if and only if it is biased; if all weights are one, the rank of a node is its height in the tree.

Our first results show that biased 2, b trees have ideal access time for all items. If x is a node, let $w(x)$ be the total weight of the items in leaves that are descendants of x ; that is, $w(x)$ equals w_i if x is a leaf containing item i , and $w(x)$ equals $\sum \{w(y) \mid y \text{ is a child of } x\}$ if x is an internal node.

LEMMA 1. For any node x , $2^{s(x)-1} \leq w(x)$. If x is a leaf, $2^{s(x)} \leq w(x) < 2^{s(x)+1}$.

Proof. By induction on $s(x)$. If x is a leaf, the definition $s(x) = \lfloor \lg w(x) \rfloor$ implies $2^{s(x)} \leq w(x) < 2^{s(x)+1}$. If x is an internal node with a minor child, x has a major child, say y , that is a leaf, and $2^{s(x)-1} = 2^{s(y)} \leq w(y) \leq w(x)$. If x is an internal node with no minor children, x has at least two major children, say y and z , and $2^{s(x)-1} = 2^{s(y)-1} + 2^{s(z)-1} \leq w(y) + w(z) \leq w(x)$. \square

LEMMA 2. If x is a leaf of depth d containing item i , $d < \lg(W/w_i) + 2$.

Proof. Let r be the root of the tree. Since the rank increases by at least one from child to parent, $d \leq s(r) - s(x)$. By Lemma 1, $\lg W = \lg w(r) \geq s(r) - 1$ and $\lg w(x) < s(x) + 1$. Combining inequalities gives the lemma. \square

THEOREM 1. A biased 2, b tree has ideal access time for all items.

Proof. Immediate from Lemma 2. \square

In our analysis of the running times of the update operations on biased 2, b trees, we shall use amortization. That is, we shall average the running time of individual update operations over a (worst-case) sequence of updates. In order to make the analysis as concrete as possible, we introduce the concept of *credits* (called *chips* in [7], [8]). A credit represents one unit of computing time. To perform an update operation, we are given a certain number of credits. Spending one credit allows us to perform $O(1)$ computational steps. If we complete the operation before running out of credits, we can save the extra credits to use on future operations. If we run out of credits before completing the operation, we can spend previously saved credits. If we can perform a sequence of operations without running out of credits during the process,

then the number of credits allocated to each operation gives an upper bound on its amortized running time. We call this upper bound the *amortized time* of the operation.

Notice that if this analytical technique is successful, then any sequence of update operations requires actual time at most a constant times the sum of the amortized times of the individual operations; a later operation may require more actual time than its amortized time, but only if a corresponding amount of time was saved in earlier operations. Some algorithms, such as the path compression method of maintaining disjoint sets [31], exhibit the opposite behavior: early operations can be slower-than-average but only if the time lost is made up in later operations.

In the case of biased 2, b trees, we shall keep track of credits saved from previous operations by storing them in the trees. In particular, we say a biased 2, b tree satisfies the *credit invariant* if every minor node y with parent x contains $s(x) - s(y) - 1$ credits. (See Fig. 1.) Note that this definition is consistent with a major node having no credits; in particular a single-node tree needs no credits. For each update operation we shall give an upper bound on the number of credits needed to perform the operation, assuming that the initial trees satisfy the credit invariant and requiring that the final trees satisfy the invariant. It is important to remember that the credits in a tree are only a conceptual device to aid in the running time analysis and neither appear in the data structure nor affect the actual implementation of the update algorithms.

We first consider (two-way) join, since all the other update operations can be defined in terms of this one. We shall describe an algorithm that joins two trees with roots x and y and returns the root of the resulting tree. The algorithm is recursive and consists of three main cases, two of which are symmetric (see Fig. 2).

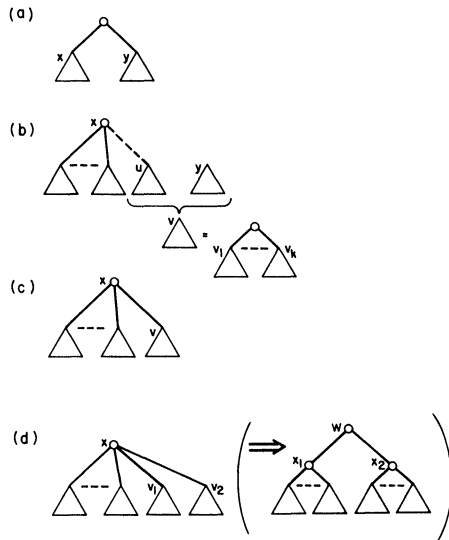


FIG. 2. Join algorithm for locally biased 2, b trees. Triangles denote subtrees. (a) Case 1 (terminate). (b) Case 2 (recurse). Join right child u of x to y , forming v . (c) Subcase 2a (v has rank less than x). Attach v as right child of x in place of u . (d) Subcase 2b (v has same rank as x). Attach children of v as children of x . Split x if it has more than b children.

Case 1. $s(x) = s(y)$, or $s(x) > s(y)$ and x is a leaf, or $s(x) < s(y)$ and y is a leaf. Create and return a new node with nodes x and y as its two children.

Case 2. $s(x) > s(y)$ and x is not a leaf. Let u be the right child of x . Remove u as a child of x and recursively join the trees with roots u and y , producing a single tree, say with root v .

Subcase 2a. $s(v) \leq s(x) - 1$. Attach v as the right child of x and return x .

Subcase 2b. $s(v) = s(x)$. In this case v has exactly two children. Attach these as children of x (to the right of the other children of x) and destroy v . Node x has now gained a child. If x now has no more than b children, return x . Otherwise split x into two nodes with $\lfloor (b+1)/2 \rfloor$ and $\lceil (b+1)/2 \rceil$ children, respectively. Create a new node w with these two nodes as children and return w . The two nodes resulting from the split have the same rank as x ; the rank of w is one greater.

Case 3. $s(x) < s(y)$ and y is not a leaf. This case is symmetric to Case 2.

Note. When node x is split in Subcase 2b, it is not necessary to divide its children approximately fifty-fifty; it is sufficient that each new node have at least two children.

The first thing we must verify about this algorithm is its correctness. An easy induction argument shows that the algorithm produces a 2, b tree whose root has rank $\max\{s(x), s(y)\}$ or $\max\{s(x), s(y)\} + 1$; in the latter case the root has exactly two children. This verifies the assertion at the beginning of Subcase 2b. A similar induction based on the following observations shows that the algorithm produces a biased 2, b tree given two biased 2, b trees as input:

Case 1. If x is minor in the new tree, y is a leaf, which means that the new tree is locally biased at x ; similarly if y is minor.

Subcase 2a. If v is minor in the new tree, u was minor in the old tree and the old left sibling of u , which is now the left sibling of v , is a major leaf, giving local bias at v . On the other hand, if the left sibling of v is minor in the new tree, it must be the case that u is a leaf of rank $s(x) - 1$. But then Subcase 2b would have occurred instead of Subcase 2a. Thus the new tree is locally biased.

Subcase 2b. If the left child of v is minor, the right child of v is a leaf, which can only happen if the children of v are u and y and the latter is a leaf. If the left sibling of u in the old tree is minor, u is a leaf of rank $s(x) - 1$, and in this case also the children of v are u and y . It follows that the tree existing after x gains a child but before x splits is locally biased. Splitting preserves local bias, which means that the final tree is locally biased.

Thus the join algorithm is correct. We shall prove by a similar case analysis that if we allocate $|s(x) - s(y)| + 1$ credits to the join, we can perform the join while preserving the credit invariant. Thus the join requires $O(|s(x) - s(y)|)$ amortized time. To carry out the analysis, we assume $s(x) \geq s(y)$. (The case $s(x) < s(y)$ is symmetric.) We begin the join with $s(x) - s(y) + 1$ credits in hand.

Case 1. We need one credit to build the new tree and $s(x) - s(y)$ credits to establish the credit invariant on y , for a total of $s(x) - s(y) + 1$.

Case 2. We acquire $s(x) - s(u) - 1$ credits from u , giving us a total of $2s(x) - s(y) - s(u)$. We need $\max\{s(u), s(y)\} - \min\{s(u), s(y)\} + 1$ to recursively join the trees with roots u and y .

Subcase 2a. We need one credit to build the new tree and $s(x) - s(v) - 1$ to place on v . If $s(y) \geq s(u)$, we use a total of $s(y) - s(u) + s(x) - s(v) + 1 \leq 2s(x) - s(y) - s(u)$, since $s(v) \geq s(y)$ and $s(x) > s(y)$. If $s(y) < s(u)$, we use $s(u) - s(y) + s(x) - s(v) + 1 \leq 2s(x) - s(y) - s(u)$, since $s(x) > s(u)$ and $s(v) \geq s(u)$.

Subcase 2b. We need one credit to build the new tree. We need no credits to place on the new children of x , since as children of v they already have the proper number of credits. Splitting x preserves the credit invariant. (Local bias implies that both nodes resulting from the split have rank $s(x)$.) Thus, we use a total of $\max\{s(u), s(y)\} - \min\{s(u), s(y)\} + 2$ credits. The analysis in Subcase 2a shows that this is at most $2s(x) - s(y) - s(u)$.

Summarizing our analysis, we have the following theorem:

THEOREM 2. *The two-way join algorithm is correct and runs in $O(|s(x) - s(y)|)$ amortized time on two trees with roots x and y .*

There is a useful alternative formulation of this theorem. We say a tree with root x is *cast to rank k* if it satisfies the credit invariant and has $k - s(x)$ additional credits on its root. If x and y are the roots of two trees cast to a rank $k > \max\{s(x), s(y)\}$, then Theorem 2 implies that we can join these trees using no extra credits, producing a single tree cast to rank k .

We can describe the behavior of the join algorithm as follows (see Fig. 3): Traverse the right path of the tree rooted at x and the left path of the tree rooted at y concurrently, descending rank-by-rank, until arriving at a leaf in one path or at two nodes of equal rank, one in each path. Merge the traversed parts of the paths, ordering nodes in decreasing order by rank. If the traversal stops at two nodes of equal rank, say k ,

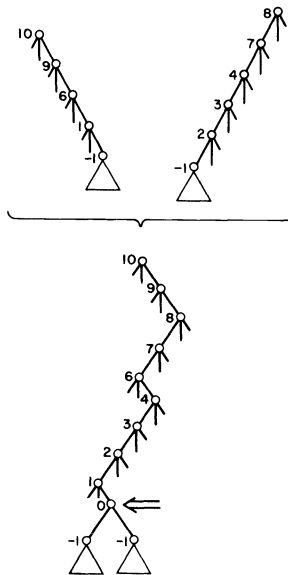


FIG. 3. A join. Only right path of left tree, left path of right tree are shown. Numbers beside nodes are ranks. Node indicated by arrow is created by join.

either make them both children of the previous node on the merged path, if the previous node has rank $k + 1$, or else make them children of a new node with rank $k + 1$, whose parent is the previous node. Starting from the bottom of the merged path and working up toward the root, split nodes as necessary until reaching a node with no more than b children. This description implies the following worst-case time bound for join:

THEOREM 3. *Consider a join of two trees with roots x and y such that the rightmost leaf descendant of x is u and the leftmost leaf descendant of y is v . The worst-case join time is $O(\max\{s(x), s(y)\} - \max\{s(u), s(v)\}) = O(\log(W/(w_- + w_+)))$, where W is the total weight of the items in the new tree, $w_- = w(u)$, and $w_+ = w(v)$.*

Note. The worst-case time for a join can be either larger or smaller than its amortized time.

Let us now consider the other update operations, beginning with three-way join. We can implement a three-way join as two successive two-way joins. Theorems 2 and 3 give the following time bounds:

THEOREM 4. Consider the three-way join of a tree with root x to a leaf y and to a tree with root z . The amortized time for the join is $O(\max \{s(x), s(y), s(z)\} - \min \{s(x), s(y), s(z)\})$. The worst-case time for the join is $O(\max \{s(x), s(y), s(z)\} - s(y)) = O(\log (W/w_i))$, where W is the total weight of the items in the new tree and i is the item in node y .

Note. The worst-case join time is the same as the access time for item i in the new tree and never exceeds the amortized join time.

A split can be implemented as a sequence of (two-way) joins. Let us first consider splitting at an item i already in the tree. Let x be the root of the tree to be split and y the leaf containing item i . The split will proceed up the path from y to x , accumulating a *left tree* of items less than i and a *right tree* of items greater than i . Initially y is the *previous node*, the parent of y is the *current node*, and the left and right trees are empty. The split consists of repeating the following general step until the root is the previous node (see Fig. 4):

General step. Delete every child of the current node to the left of the previous node. If there is one such child, join it to the left tree; if there are two or more such children, give them a new common parent and join the resulting tree to the left tree. Repeat this process with the children to the right of the previous node, joining the resulting tree to the right tree. Remove the previous node as a child of the current node and destroy it if it is not y . Make the current node the new previous node and its parent the new current node.

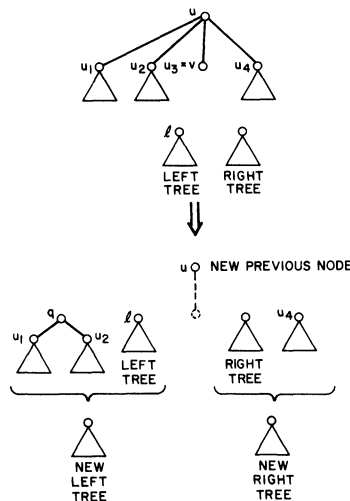


FIG. 4. One step of split algorithm. Node u is the current node, v the previous node, u_1, u_2, \dots the children of u , q the root of the tree that is joined to the left tree and l the root of the left tree.

This algorithm is obviously correct. To establish its amortized time, let u be the current node, v the previous node, q the root of the tree containing children of u that is joined to the left tree, and l the root of the left tree. An easy induction shows that $s(v) \geq s(l)$. Suppose we begin the current execution of the general step with both the left and the right tree cast to rank $s(v) + 1$. The following argument shows that with $2(s(u) - s(v)) + 5$ additional credits, we can carry out the general step and finish with both the left and the right tree cast to rank $s(u) + 1$. If we place at most two new credits on q and $s(u) - s(v)$ new credits on l , we can join q and l to produce a new left tree cast to rank $s(u) + 1$, since either $s(q) = s(u)$ (if u has two more children to the left

of v) or $s(q) \leq s(u) - 1$ and q has $s(u) - s(q) - 1$ credits on it already (if u has one child to the left of v). Similarly we need $s(u) - s(v) + 2$ credits for the right join. One new credit accounts for the $O(1)$ time required for the rest of the general step, giving a total credit count of $2(s(u) - s(v)) + 5$.

Summing over all executions of the general step, we obtain:

THEOREM 5. *The amortized time to split a tree with root x at leaf y is $O(s(x) - s(y)) = O(\log(W/w_i))$, where W is the weight of all the items in the tree and i is the item in leaf y . Each of the (up to) three resulting trees is cast to rank $s(x) + 1$.*

Splitting at an item not in the tree is just like splitting at an item in the tree, except that the initial execution of the general step is slightly different. Let x be the root of the tree, i an item not in the tree, i^- and i^+ the largest item in the tree less than i and the smallest item in the tree greater than i , respectively. And let y be the *handle* of i , which is defined to be the nearest common ancestor of the leaves containing i^- and i^+ . To split the tree, we combine all children of y containing items smaller than i ; the result becomes the original left tree. We combine the remaining children of y (those containing items greater than i) to form the original right tree. Then we make y the previous node and its parent the new current node and repeat the general step as before.

THEOREM 6. *The amortized time to split a tree with root x at an item i not in the tree is $O(s(x) - s(y)) = O(\log(W/(w_{i^-} + w_{i^+})))$, where y is the handle of i , and i^- , i^+ are as defined above. Each of the (up to) two resulting trees is cast to rank $s(x) + 1$.*

Unlike join, split does not have a logarithmic bound on its worst-case running time. However, as we shall see at the end of this section, we can get a good bound on the worst-case split time by strengthening the bias property and changing the implementation of join to maintain this stronger property.

We can implement each of the remaining update operations as a combination of a split and a join: an insertion is a two-way split followed by a three-way join, a deletion is a three-way split followed by a two-way join, and a weight change is a three-way split followed by a three-way join. The next theorem gives the amortized time of these operations.

THEOREM 7. *The amortized time to perform an insertion of item i into a tree is $O(\log(W'/\min\{w_{i^-} + w_{i^+}, w_i\}))$, where W' is the weight of the tree after the insertion and i^- and i^+ are the largest item smaller than i and the smallest item larger than i , respectively. The amortized time to perform a deletion of item i from a tree is $O(\log(W/w_i))$, where W is the weight of the tree before the deletion. The amortized time to perform a weight change on item i in a tree is $O(\log(\max\{W, W'\}/\min\{w_i, w'_i\}))$, where W, W', w_i, w'_i are the weights of the tree before and after the update and the weights of i before and after the update, respectively.*

Proof. Consider an insertion. The two-way split takes amortized time $O(\log(W/(w_{i^-} + w_{i^+})))$, where W is the weight of the original tree, and produces trees cast to a rank of at least $\lceil \lg(\max\{w_{i^-}, w_{i^+}\}) \rceil$. The three-way join thus requires $O(\log(W'/\min\{w_{i^-} + w_{i^+}, w_i\}))$ additional amortized time. This gives the bound for insertion, since $W \leq W'$. The three-way split beginning a deletion requires $O(\log W/w_i)$ time; the two-way join completing it takes $O(1)$ additional amortized time since the trees resulting from the split are cast to the same rank. This gives the bound for deletion. The three-way split beginning a weight change also requires $O(\log W/w_i)$ amortized time and produces trees cast to a rank of at least $\lceil \lg w_i \rceil$. The three-way join completing the weight change thus requires $O(\log(\max\{W, W'\}/\min\{w_i, w'_i\}))$ additional amortized time. This gives the bound for weight change. \square

Remark. In practice it may be useful to design customized implementations of insert, delete and weight change, rather than expressing them in terms of join and

split. We leave this as a (nontrivial) exercise; the algorithms so obtained are more complicated than those using join and split.

The data structure we have described and analyzed is a good one if amortized running time is the complexity measure of interest. We shall now describe a modification appropriate if worst-case per-operation running time is important. A *globally biased 2, b tree* is a 2, b tree with the following property, which is stronger than local bias (see Fig. 5):

Global bias. Any neighboring leaf of a minor node y with parent x has rank at least $s(x) - 1$. (We say the tree is *globally biased* at y .)

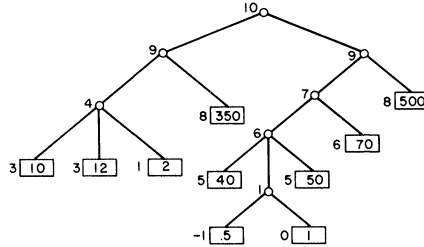


FIG. 5. A globally biased 2, 3 tree. Weights are inside leaves, ranks are to left of nodes.

Since any globally biased 2, b tree is locally biased, globally biased 2, b trees have ideal access time. The following version of the join algorithm will join two globally biased 2, b trees with roots x and y into a single globally biased 2, b tree and return the root of the new tree. (See Figs. 2 and 6.)

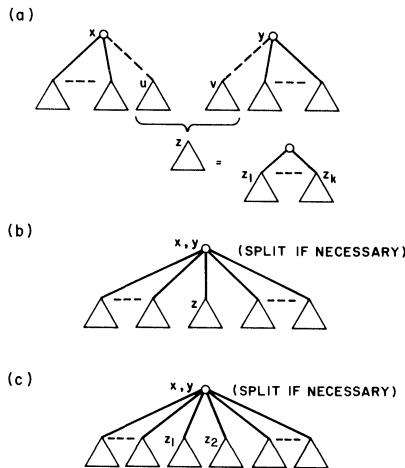


FIG. 6. Case 4 of global join algorithm (equal ranks, nonterminating). (a) Right child of x , left child of y joined to form z . (b) Rank of z less than x . Attach z as right child of x , fuse x and y , split if necessary. (c) Ranks of z and x equal. Fuse x, z, y ; split if necessary.

Case 1. $s(x) \geq s(y)$ and x is a leaf, or $s(x) \leq s(y)$ and y is a leaf. Create a new node u with nodes x and y as its two children and return u .

Case 2. $s(x) > s(y)$ and x is not a leaf. Proceed as in Case 2 of the join algorithm for locally biased trees.

Case 3. $s(x) < s(y)$ and y is not a leaf. Symmetric to Case 2.

Case 4. $s(x) = s(y)$ and neither x nor y is a leaf. Let u be the right child of x and v the left child of y . Remove u as a child of x and v as a child of y . Recursively join the trees rooted at u and v , producing a single tree, say with root z . If $s(z) < s(x)$, attach z as the right child of x ; otherwise ($s(z) = s(x)$) attach the two children of z as (right-most) children of x and destroy z . Fuse x and y into a single node. If this new node has no more than b children, return it; otherwise split it and return a new node whose two children are the results of the split.

We shall refer to the join algorithm for locally biased trees as *local join* and to the version for globally biased trees as *global join*. We can describe a global join iteratively as follows (see Fig. 7): Traverse the right path of the left tree and the left

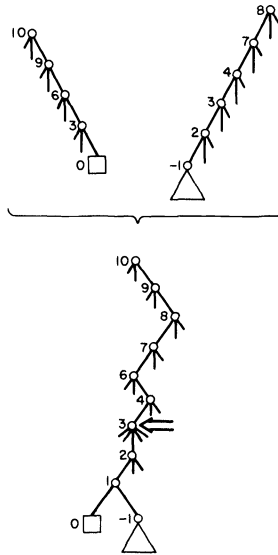


FIG. 7. A global join. Nodes on spliced path must be split if necessary, working bottom-up. Node indicated by arrow is formed by fusing two nodes, one from each tree.

path of the right tree concurrently, descending rank-by-rank until reaching a leaf in one of the paths. Merge the traversed parts of the paths, ordering nodes in decreasing order by rank and fusing any two nodes of equal rank. If the traversal stops at two leaves of equal rank, say k , do not fuse them but instead either make them children of the previous node on the spliced path, if the previous node has rank $k + 1$, or else make them children of a new node with rank $k + 1$ whose parent is the previous node. Proceed back up the merged path, splitting every node with more than b children.

As with a local join, a global join of trees with roots x and y produces a 2, b tree whose root has rank $\max\{s(x), s(y)\}$ or $\max\{s(x), s(y)\} + 1$; in the latter case the root has exactly two children. The following discussion shows that the new tree is globally biased. Let the *left tree* and the *right tree* be the original trees with roots x and y , respectively. Consider the tree produced by applying the global join algorithm without doing any splitting. We call this the *fused tree*. The only possible nodes at which the fused tree might not be globally biased are minor nodes along the spliced path; let v with parent u be such a node. Node v has leaf descendants in either the left tree, the right tree, or both; v is either a node, say q , in the left tree, a node, say r , in the right tree, a node produced by fusing two nodes, say q from the left tree and r from the right tree, or a new node with two children (at least one of which is a leaf),

say q from the left tree and r from the right tree. If q exists, its left neighboring leaf in the left tree has rank at least $s(u) - 1$ and is the left neighboring leaf of v in the fused tree. Similarly if r exists, its right neighboring leaf in the right tree has rank at least $s(u) - 1$ and is the right neighboring leaf of v in the fused tree. Suppose q does not exist. Then v is a minor node in the right tree. Let g with parent f be the node on the right path of the left tree such that $s(g) \leq s(v) < s(f)$. (If g does not exist the fused tree is globally biased at v .) Then $s(u) \leq s(f)$, which when combined with the fact that v is minor ($s(v) + 1 < s(u)$) implies that g is minor in the left tree. Thus the neighboring leaf of g , which is the neighboring leaf of v in the fused tree, has rank at least $s(u) - 1$. A similar argument applies if r does not exist. Thus in any case the fused tree is globally biased at v . Splitting preserves global bias; thus the tree resulting from the join is globally biased.

THEOREM 8. *The global join algorithm is correct. Furthermore the worst-case time bound given in Theorem 3 for local join holds also for global join. Thus a global join runs in $O(\log(W/(w_- + w_+)))$ worst-case time, where W is the total weight of both trees, w_- is the weight of the rightmost item in the left tree, and w_+ is the weight of the leftmost item in the right tree.*

Proof. The discussion above verifies correctness; the time bound follows immediately. \square

THEOREM 9. *A three-way join of globally biased 2, b trees, implemented as two successive global joins, has the same worst-case time bound as given in Theorem 4 for three-way local join. Thus a three-way join takes $O(\log(W/w_i))$ worst-case time, where W is the total weight of the joined tree and w_i is the weight of the item inserted between the two trees.*

Proof. Immediate from Theorem 8. \square

We can split a globally biased 2, b tree exactly as we did a locally biased 2, b tree, using local joins rather than global joins to build up the left tree and the right tree generated by the split. Below we shall verify that this method results in a globally biased tree, and also get a bound on the running time of the operation. Let u be the current node, v the previous node, q the root of the tree containing the children of u that are to be joined to the left tree, and l the root of the left tree. (See Fig. 4.) The analysis is simplified by the assumption that the subtrees rooted at q and l are not empty and v is not a leaf in the original tree.

We want to verify by induction that after the entire split the resulting left and right trees are globally biased. The induction hypothesis is that the leftmost path descending from l is in the original tree (except possibly for l itself), and that the tree rooted at l is globally biased. The tree with root q is globally biased by construction, and its rightmost path (possibly excluding the node q itself) is a path in the original tree. Since v is not a leaf in the original tree, its left sibling cannot be minor. This implies that $s(q) = s(u) - 1$ or $s(q) = s(u)$. We know by the earlier discussion of split that $s(l) \leq s(v) < s(u)$. Combining these inequalities gives $s(l) \leq s(q)$. The join of the trees with roots q and l proceeds down the rightmost path from q until reaching the first node t such that t is a leaf or $s(t) \leq s(l)$. Because of global bias, each node above t is major, and t must also be major unless l is a leaf. Thus the rank decreases by 1 each step down the tree, and either $s(l) = s(t)$ or one of l or t is a leaf. (The important point is that the join does not continue down the left-most path of l as it normally might.) At this point l and t become siblings, and the join terminates (after splitting nodes back up the merged path). We can now verify that our induction holds for the new left tree just created. The leftmost path of this tree is that of the original tree except possibly for the top node (which is new only if $s(q) = s(r)$). We have already

said that the nodes above t must be major. The other nodes on the original rightmost path descending from q also have global bias because they have the same adjacent leaf on the right that they used to have, namely the leftmost leaf of l . This shows that the new left tree is globally biased.

It also follows from this discussion that the number of steps taken by the join is $O(s(q) - s(l))$, because (as mentioned above) the join only traverses down the right path of q until it reaches t . (It does not then propagate down the left path of l .)

To obtain a time bound for split, let us consider the joins that form the left tree. Let q_1, q_2, \dots, q_k be the roots of the successive trees joined into the left tree, let u_i for $1 \leq i \leq k$ be the current node when the tree with root q_i is joined with the left tree, let l_i for $1 \leq i \leq k$ be root of the left tree after the tree with root q_i is joined (thus $l_1 = q_1$), and finally let x be the root of the tree to be split and y the node at which the split starts. The discussion above implies that $s(q_i) \leq s(l_i) \leq s(u_i) \leq s(u_{i+1}) - 1 \leq s(q_{i+1}) \leq s(u_{i+1})$ for $1 \leq i \leq k-1$ and that the join of the trees with roots q_{i+1} and l_i takes $O(s(q_{i+1}) - s(l_i))$ time for $1 \leq i \leq k-1$. For $2 \leq i \leq k-1$, this bound is $O(s(u_{i+1}) - s(u_i))$ by the inequalities above. Consider the case $i=1$. We have $l_1 = q_1$. If $s(q_1) \geq s(u_1) - 1$, then $s(q_2) - s(l_1) \leq s(u_2) - s(u_1) + 1$. If $s(q_1) < s(u_1) - 1$, then q_1 is a minor child of u_1 , and the rightmost leaf descendant of q_2 has rank at least $s(u_1) - 1$. Thus the join of trees with roots q_1 and q_2 takes $O(s(q_2) - s(u_1)) = O(s(u_2) - s(u_1))$ time. We conclude that for $1 \leq i \leq k-1$, the join of trees with roots l_i and q_{i+1} takes $O(s(u_{i+1}) - s(u_i))$ time. Summing over i , we obtain a bound of $O(s(x) - s(y))$ on the total time to form the left tree. The same argument applies to the right tree. Thus we have the following theorem:

THEOREM 10. *The worst-case time to split a globally biased 2, b tree rooted at x , starting at a node y , is $O(s(x) - s(y))$. If y is a leaf containing item i , the time is $O(\log(W/w_i))$. If y is an internal node, the time is $O(\log(W/(w_- + w_+)))$, where w_- and w_+ are the weights of the items in the left and right neighboring leaves of y , respectively.*

If we implement each of the operations insert, delete and reweight as a split followed by a global join, we obtain from Theorems 8–10 the following time bounds:

THEOREM 11. *The worst-case time to insert an item i into a globally biased 2, b tree is $O(\log(W/(w_- + w_+)) + \log(W'/w_i))$, where the various parameters are defined as in Theorem 7. The time to delete an item i is $O(\log(W/w_i) + \log(W'/(w_- + w_+)))$. The time to change the weight of an item i is $O(\log(W/w_i) + \log(W'/w'_i))$.*

Remark. Based on the time bounds we have derived, the choice between locally and globally biased trees does not seem to be clear-cut but depends upon the application.

We conclude this section by describing a way to build a globally biased 2, b tree of n items in $O(n)$ time. The idea is to form n single-leaf trees, one per item, and join them one-at-a-time, left-to-right, into a large tree, initially empty. To join a single leaf x with the current tree, we use a bottom-up method. We start at the rightmost leaf of the tree and walk up until finding the maximum-rank node, say v , such that $s(v) \leq s(x)$. If v is the tree root, we create a new root with two children, v and x , and stop. If v is not the tree root, we compare $s(p(v))$ to $s(x)$. If $s(p(v)) = s(x) + 1$, we make x the rightmost child of $p(v)$ and split nodes up the right path as necessary. If $s(p(v)) > s(x) + 1$, we create a new node with two children, v and x , and replace v as a child of its old parent by the new node. (See Fig. 8.)

This bottom-up join method obviously maintains global bias. To obtain a bound on the total time for all $n-1$ joins, we note that, except for $O(1)$ time per join, each step taken by a join either decreases the number of nodes on the rightmost path of the current tree or splits a b -node (a node with b children), thereby reducing the number of b -nodes by one. A single join can only increase the number of nodes on

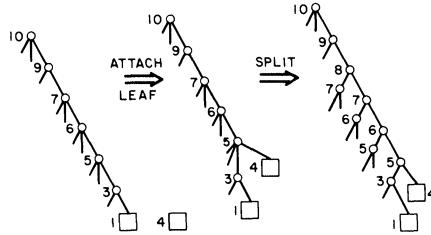


FIG. 8. Bottom-up join of a single item with a globally biased 2, 3 tree.

the rightmost path by one and can only increase the number of b -nodes by one. It follows that the $n - 1$ joins require a total of $O(n)$ time, and we have the following theorem:

THEOREM 12. *Repeated single-node, bottom-up joins will construct a globally biased 2, b tree in $O(n)$ actual time.*

Note. Theorem 12 does *not* include time corresponding to the credits necessary to establish the credit invariant on the constructed tree (if we are using locally biased trees); the number of credits needed depends upon the relative weight of the items and is not bounded by any function of n . (Consider a tree containing two items with weights 1 and 2^k for k arbitrarily large.)

Remark. Local join and global join as we have described them each consist of a top-down pass (for merging) followed by a bottom-up pass (for splitting). However, if $b \geq 4$ either form of join can be implemented in a one-pass, purely top-down fashion by preemptively splitting nodes with b or more children during merging.

4. Biased binary trees. In practice, implementations of balanced tree data structures are plagued by a multiplicity of cases, making the resulting code lengthy, opaque and hard to prove correct. In this section we shall describe a class of biased search trees whose update algorithms are relatively easy to program and have a manageable number of cases. We shall only sketch proofs of algorithm correctness and time bounds, since the proofs use exactly the same techniques as in § 3.

A *locally biased binary search tree* is a full binary search tree (every internal node has exactly two children), each of whose nodes x has an integer rank $s(x)$, such that the ranks have the following properties:

- (i) If x is a leaf, then $s(x) = \lfloor \lg w(x) \rfloor$.
- (ii) If node y has parent x , $s(y) \leq s(x)$; if y is a leaf, $s(y) \leq s(x) - 1$.
- (iii) If node y has grandparent x , then $s(y) \leq s(x) - 1$.
- (iv) *Local bias.* A node is *minor* if the rank of its parent is at least two greater than its own rank and *major* otherwise. Let y be a minor node with parent x . If y is the left child of x , either the sibling of y or the left child of that sibling is a leaf of rank $s(x) - 1$. If, in addition, x is the right child of its parent and has the same rank as its parent, then either the sibling of x or the right child of that sibling is a leaf of rank $s(x) - 1$. A symmetric condition holds if y is the right child of its parent. (See Fig. 9.)

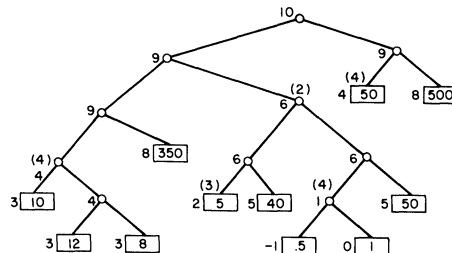


FIG. 9. A locally biased binary tree.

Biased binary trees are a binarized version of biased 2, 4 trees; if we take a biased binary tree and condense into a single node all adjacent nodes of the same rank, we obtain a biased 2, 4 tree. (See Fig. 10.) Biased binary trees generalize symmetric binary *B*-trees [5], which have been described as red-black trees [12]. In the case of equal weights, we obtain the red-black representation of a biased binary tree by calling an edge *red* if parent and child have the same rank and *black* if their ranks differ by one; in this case all nodes are major. If we want to be colorful we can in the general case call an edge *blue* if it joins a minor child with its parent; then we can call biased binary trees *red, black and blue trees*.

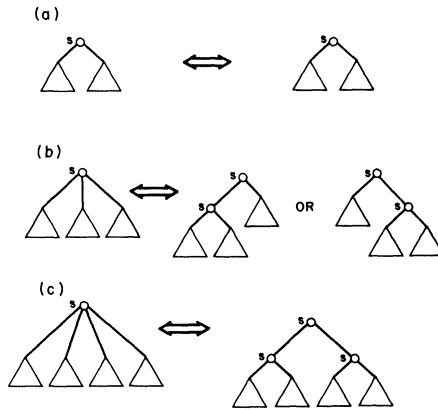


FIG. 10. Correspondence between biased 2, 4 trees and biased binary trees. (a) 2-node. Rank of node is *s*. Roots of subtrees denoted by triangles have ranks less than *s*. (b) 3-node. There are two possible binarized forms. (c) 4-node.

All the theorems presented in § 3 for biased 2, *b* trees hold for biased binary trees, since we can regard biased binary trees as just a representation of biased 2, 4 trees. In particular biased binary trees have ideal access time for all items. In the remainder of this section we shall give algorithms for joining and splitting biased binary trees. The correspondence with biased 2, 4 trees is somewhat loose because there are two representations of a 3-node.

We begin by presenting an algorithm for join. If *x* is a node, we denote the left child of *x* by *l(x)* and the right child of *x* by *r(x)*; if *x* is a leaf, *l(x) = r(x) = null*. By *promoting* a node we mean increasing its rank by one. The algorithm uses two functions, *tilt left (x)* and *tilt right (x)*, whose behavior is as follows:

tilt left (x): If both children of internal node *x* have the same rank as *x*, promote *x* and return *x*. Otherwise if the right but not the left child of *x* has the same rank as *x*, perform a single left rotation at node *x* (see Fig. 11) and return the new parent of *x* (the old right child of *x*). In all other cases merely return *x*.

tilt right (x): Symmetric to *tilt left (x)*.

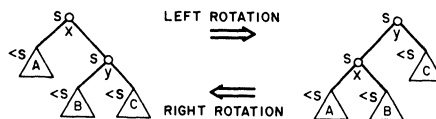


FIG. 11. A single rotation at node *x*.

Remark. If x' is the node returned by *tilt left* (x), then the leaf descendants of x' in the new tree are exactly the leaf descendants of x in the old tree. Also, x' and its right child have different ranks. Achieving the latter condition is the purpose of the *tilt left* operation.

The join algorithm consists of a function *local join* (x, y) that returns the root of the tree formed by joining the trees with roots x and y . The function *local join* (x, y) is defined by the following cases (see Fig. 12):

Case 1. $s(x) = s(y)$, or $s(x) > s(y)$ and x is a leaf, or $s(x) < s(y)$ and y is a leaf. Create and return a new node with left child x , right child y and rank $\max \{s(x), s(y)\} + 1$.

Case 2. $s(x) > s(y)$ and x is not a leaf. Replace x by *tilt left* (x). Let z be the right child of x . Define the new right child of x to be *local join* (z, y) and return x .

Case 3. $s(x) < s(y)$ and y is not a leaf. Symmetric to Case 2.

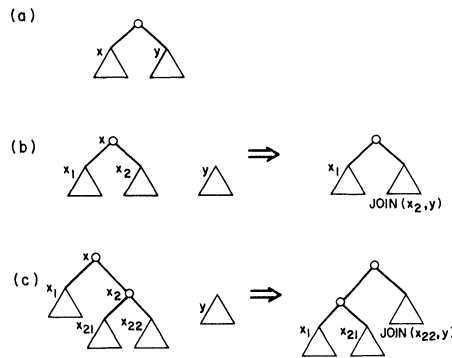


FIG. 12. Join algorithm for locally biased binary trees. (a) Case 1. Terminate. (b) Case 2 with x not a right-leaning 3-node. Promote x if x a 4-node. Replace right child of x by the join of this child and y . (c) Case 2 with x a right-leaning 3-node. Rotate left at x . Replace right child of root by the join of this child and y .

We can verify the correctness of this algorithm as follows. The function call *local join* (x, y) returns a tree whose root has rank $\max \{s(x), s(y)\}$ or $\max \{s(x), s(y)\} + 1$. Furthermore if Case 2 occurs and a promotion takes place (which happens if $s(x) > s(y)$, x is not a leaf, and $s(x) = s(l(x)) = s(r(x))$), then in the next call, which is *local join* ($r(x), y$), Case 2 also occurs but neither a promotion nor a rotation takes place, by properties (ii) and (iii). Thus *local join* ($r(x), y$) returns a node of rank $s(r(x))$. Similarly if Case 3 occurs and a promotion takes place, the next call *local join* ($x, l(y)$) returns a node of rank $s(l(y))$. It follows that if the original call *local join* (x, y) returns a node, say z , of rank $\max \{s(x), s(y)\} + 1$, both children of z have rank less than $s(z)$. An inductive case analysis using this fact shows that *local join* is correct.

To establish a time bound for local join we can use the same credit invariant for biased binary trees that we used for biased 2, b trees; a count of credits as in § 3 proves Theorem 2 for biased binary trees. Theorems 3 and 4 for biased binary trees follow immediately. Although we have defined local join recursively, it is easy to give an iterative, purely top-down version. We leave this as an exercise.

We can use the same split algorithm on biased binary trees that we used on biased 2, b trees and the number of cases is much reduced. To split a tree at a leaf x , we initialize v (the current node) to be x , and q and r (the roots of the left and right trees, respectively) to be null. Then we repeat the following general step until v is the root of the tree (see Fig. 13), where $p(v)$ denotes the parent of node v :

General step. Case 1. v is the right child of $p(v)$. Let y be the left child of $p(v)$. If $q = \text{null}$, replace q by y ; if $q \neq \text{null}$, replace q by *local join* (y, q). Replace v by $p(v)$. Destroy $r(v)$ if it is not the original leaf x .

Case 2. v is the left child of $p(v)$. Symmetric to Case 1.

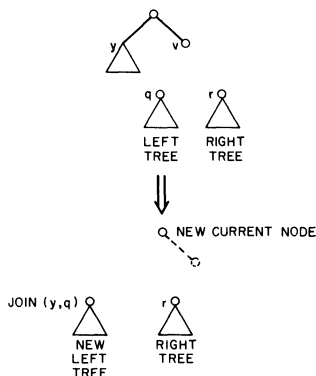


FIG. 13. One step of split algorithm.

To split a tree at an internal node x , we proceed as above except that we initialize q to be the left child of x and r to be the right child of x ; in this case neither q nor r is ever **null**. The correctness of three-way split and two-way split is immediate. Theorems 5–7 hold for biased binary trees, as we can easily establish using virtually the same proofs as in § 3. Thus the amortized time bounds derived for biased 2, b trees hold for biased trees. Note that when a biased binary tree whose root has rank k is split, the resulting tree(s) all have rank at most $k + 1$.

As in § 3, we can improve the worst-case-per-operation behavior of biased binary trees by strengthening the bias property (iv). A *globally biased binary tree* is a full binary search tree having properties (i), (ii), (iii) and the following:

(iv') *global bias*. If y is a minor node with parent x , then any neighboring leaf of y has rank at least $s(x) - 1$.

We can modify the join algorithm so that it produces a globally biased tree if the two input trees are globally biased, although the number of cases increases. As in § 3, the idea is to continue the join until finding a leaf, instead of terminating when encountering two nodes of equal rank. The resulting algorithm *global join* (x, y) consists of the following cases (see Fig. 14):

Case 1. $s(x) \geq s(y)$ and x is a leaf, or $s(x) \leq s(y)$ and y is a leaf. Create and return a new node with left child x , right child y and rank $\max\{s(x), s(y)\} + 1$.

Case 2. $s(x) \geq s(y)$ and x is not a leaf. Replace x by *tilt left* (x). Let z be the right child of x . Define the new right child of x to be *global join* (z, y) and return x .

Case 3. $s(x) < s(y)$ and y is not a leaf. Symmetric to Case 2.

Case 4. $s(x) = s(y)$ and neither x nor y is a leaf. If $s(r(x)) < s(x)$, let $u = x$; otherwise let $u = r(x)$. (In either case $s(r(u)) < s(x)$ and $s(u) = s(x)$.) If $s(l(y)) < s(y)$, let $v = y$; otherwise let $v = l(y)$. Perform *global join* ($r(u), l(v)$); let z be the root of the resulting tree.

Case 4a. $s(z) = s(x)$. Replace $r(u)$ by $l(z)$, $l(v)$ by $r(z)$, $l(z)$ by x , $r(z)$ by y and $s(z)$ by $\max\{s(x), s(y)\} + 1$. Return z .

Case 4b. ($s(z) < s(x)$).

Case 4b(i). $u = r(x)$. Replace $r(x)$ by $l(u)$, $l(v)$ by z , $l(u)$ by x , $r(u)$ by y and $s(u)$ by $\max\{s(x), s(y)\} + 1$. Return u .

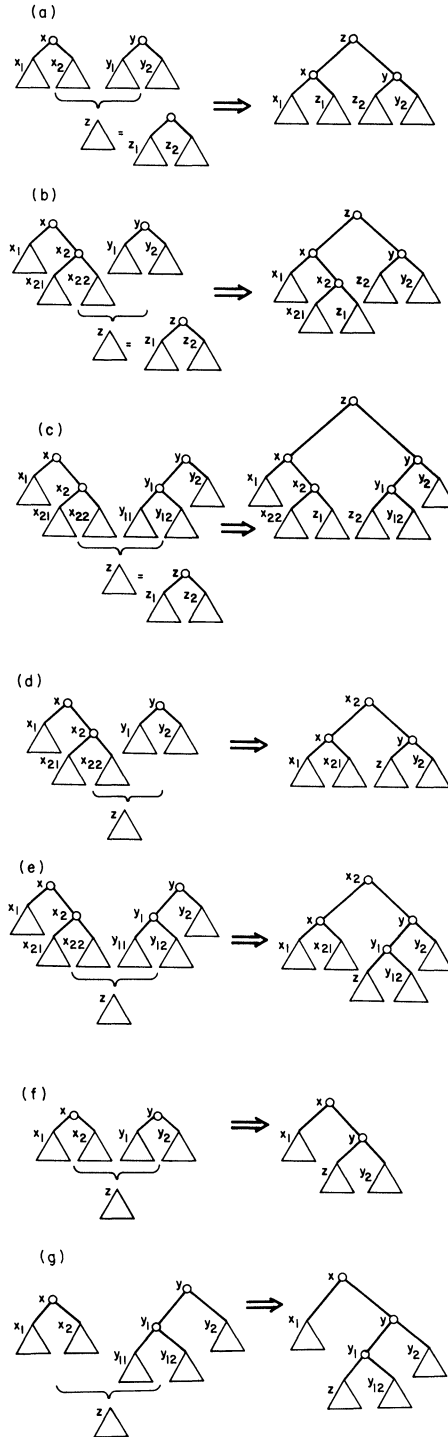


FIG. 14. Case 4 of global join algorithm (equal ranks, non terminating). (a) Case 4a ($s(z) = s(x)$) with $u = x, v = y$. (b) Case 4a with $u = r(x), v = y$. Case 4a with $u = x, v = l(y)$ is symmetric. (c) Case 4a with $u = r(x), v = l(y)$. (d) Case 4b(i) with $v = y$. Case 4b(ii) with $u = x$ is symmetric. (e) Case 4b(i) with $v = l(y)$. Case 4b(ii) with $u = r(x)$ is symmetric. (f) Case 4b(iii) with $v = y$ and Case 4b(v). Case 4b(iv) with $u = x$ is symmetric. (g) Case 4b(iii) with $v = l(y)$. Case 4b(iv) with $u = r(x)$ is symmetric.

Case 4b(ii). $v = l(y)$. Replace $l(y)$ by $r(v)$, $r(u)$ by z , $l(v)$ by x , $r(v)$ by y and $s(v)$ by $\max \{s(x), s(y)\} + 1$. Return v .

Case 4b(iii). $u = x$ and $s(x) = s(l(x))$. Replace $l(v)$ by z , $r(x)$ by y and $s(x)$ by $s(x) + 1$. Return x .

Case 4b(iv). $v = y$ and $s(y) = s(r(y))$. Replace $r(u)$ by z , $l(y)$ by x and $s(y)$ by $s(y) + 1$. Return y .

Case 4b(v). $u = x$, $s(x) > s(l(x))$, $v = y$ and $s(y) > s(r(y))$. Replace $l(v)$ by z and $r(x)$ by y . Return x .

Remark. Cases 4b(i) and (ii) are nondisjoint, as are Cases 4b(iii) and (iv). If two cases are possible, the choice can be made arbitrarily.

A straightforward but tedious case analysis verifies the correctness of this method. With this implementation of global join and with split implemented using local join, Theorems 8–11 hold for globally biased binary trees. As with local join, global join can be implemented as an iterative, purely top-down method if desired.

Our last result in this section is an algorithm which constructs a globally biased binary tree of n items in $O(n)$ time. We use the same approach as in § 3; namely, we begin with an empty tree and successively join each item into the tree, proceeding left-to-right. To join each new leaf into the tree, we use a bottom-up method. To simplify the joins, we maintain the invariant that nodes down the right path of the tree strictly decrease in rank. To give access to the tree, we maintain a pointer to its rightmost leaf. To join a single leaf y with the tree, we start at the rightmost leaf x and walk up the tree, replacing x by $p(x)$, until x is the tree root or $s(p(x)) > s(y)$. We thus create a new node with left child x , right child y and rank $\max \{s(x), s(y)\} + 1$. If x was the old tree root we are finished. If not, we make the new node the right child of the old parent of x . Then we perform a tilt left on this old parent and walk up toward the root, performing a tilt left on each node, until reaching the root or performing a tilt left that does not cause a promotion. (See Fig. 15.)

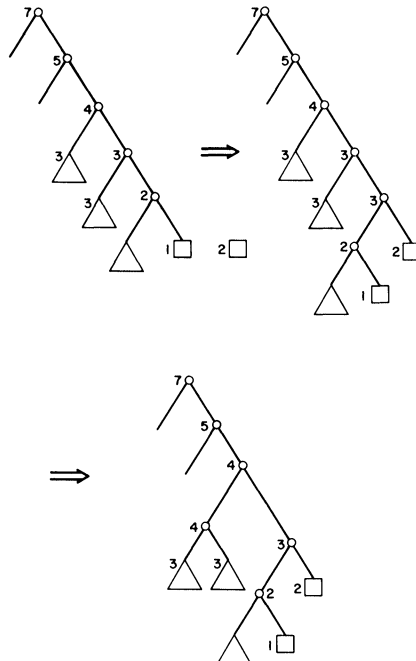


FIG. 15. Bottom-up join of a single item with a globally biased tree.

The correctness of this method is obvious; the same argument used in § 3 shows that forming an n -leaf tree by $n-1$ successive bottom-up joins requires $O(n)$ time; that is, Theorem 12 holds for biased binary trees.

There are many alternatives to the specific update algorithms we have presented for biased binary trees. By varying the order in which the basic operations of promotion and rotation are performed, one may obtain a range of different implementations. Guibas and Sedgewick [12] have explored such variants for the equal-weight case; we leave it as an exercise generalizing their algorithms to the biased case.

5. Summary, applications and related work. We have presented two classes of biased search trees. Each has a locally biased and a globally biased version. All our search trees have ideal worst-case access times for all items. Locally biased search trees have fast amortized update times, as given in Theorems 2 and 4–7. Globally biased search trees have fast worst-case update times, as given in Theorems 8–11.

Biased search trees have a number of applications, three of which we list below. This list is meant to be illustrative, not exhaustive.

1. *Dictionaries with access weights.* The most obvious application of a biased search tree is to store a table, such as a name table in a compiler, a natural-language dictionary, or a telephone directory. If we have a priori estimates of the access frequencies, we can use these as weights. Alternatively, we can keep a frequency count for each item and use this as its weight, increasing the count by one each time we access the item. With this method the time to rebalance after increasing a count is proportional to the access time, and the time for an insertion is also proportional to the access time, since an item has an initial count of one.

We can also include weights for unsuccessful searches in this scheme: we assign to the leaf containing item i a weight equal to our estimate of the frequency of successful searches for i plus the frequency of unsuccessful searches for items between item $i-1$ and item i ; any such unsuccessful search will terminate at the leaf containing item i (or at an ancestor of that leaf if double keys are used; see Appendix B). When a new item is inserted between items $i-1$ and i , we must somehow apportion the weight for unsuccessful searches between $i-1$ and i to the two new intervals created by the insertion.

By perturbing the weights, we can guarantee an access time of $O(\min\{\log n, \log(W/w_i)\})$ for every item i , thus obtaining the behavior of balanced and biased search trees simultaneously. To do this, we assign to every item i a weight of $1/n + w_i/W$. It is not necessary to update the weights of all the items every time n changes; it suffices to update all the weights whenever n changes by a factor of two. When amortized over a sequence of insertions and the deletions, the time for updating weights is $O(1)$ per insertion or deletion.

2. *Tries and multidimensional search trees.* Suppose the items to be stored are k -dimensional vectors (or equivalently lists of length k) ordered lexicographically, and that comparing the corresponding components of two items takes $O(1)$ time. We can use biased search trees to store collections of such vectors so that access, insertion, deletion, join and split take $O(\log n + k)$ time, either in the amortized sense if we use locally biased trees or in the worst-case sense if we use globally biased trees. See Mehlhorn [26] and Güting and Kriegel [13]. The idea extends to allow the vectors to have weights measuring access frequencies [14] and to allow partial-match queries [23].

3. *Dynamic trees.* A number of network optimization algorithms require a data structure to represent a collection of rooted trees on which we can perform the following two update operations:

- link* (v, w): If v is the root of one tree and w is a node in another tree, combine the trees containing v and w by adding an edge joining v and w .
- cut* (v, w): If there is an edge joining v and w , delete it, thereby breaking the tree containing v and w into two trees, one containing v and one containing w .

Using biased search trees we have been able to develop a data structure for such *dynamic trees* in which link or cut operations, as well as other operations of interest, take $O(\log n)$ time per operation [28]. We divide each dynamic tree into a collection of disjoint paths and represent each path by a biased search tree. This data structure leads to improved running times for several network optimization algorithms. For example, we are able to find a maximum network flow in an n -vertex, m -edge graph in $O(nm \log n)$ time.

Recently Feigenbaum and Tarjan [9] have developed two additional types of biased search trees. These are a biased form of B -trees and a biased form of weight-balanced trees. The biased B -trees have $O(\log_b (W/w_i))$ worst-case access time, where b is the maximum number of children per node, and have correspondingly efficient update times. They exist in both locally biased and globally biased forms. Kriegel and Vaishnavi [22] have proposed a data structure with similar access times but less favorable update times.

In another related development Sleator and Tarjan [30], [31] have devised “self-adjusting” binary search trees with amortized access and update times similar to those of biased search trees. The advantage of self-adjusting trees is their simplicity, since there is no balance condition to maintain. The disadvantages of self-adjusting trees are that they must be adjusted frequently (even during accesses), and the time bound for access is amortized rather than worst-case.

Appendix A. Tree terminology. A *rooted tree* is either empty or consists of a single node r , called the *root*, and a set of zero or more rooted trees T_1, \dots, T_k that are node-disjoint and do not contain r . The roots r_1, \dots, r_k of T_1, \dots, T_k are the *children* of r ; r is the *parent* of r_1, \dots, r_k . A node without children is a *leaf*; a node with at least one child is an *internal node*. Two nodes with the same parent are *siblings*. The *degree* of a node is the number of its children.

A *path* of length $l-1$ in a tree is a sequence of nodes v_1, v_2, \dots, v_l such that v_{i+1} is a child of v_i for $1 \leq i < l$. The path goes from v_1 *down* to v_l and from v_l *up* to v_1 . A node v is an *ancestor* of a node w and w is a descendant of v if there is a path from v down to w . (Every node is an ancestor and a descendant of itself.) If w is a leaf, it is a *leaf descendant* of v . Two nodes are *unrelated* if neither is an ancestor of the other.

Let v be any node in a tree T . There is a unique path from the root of T down to v ; the length of this path is the *depth* of v . The *height* of v is the length of the longest path from v down to a leaf. The *subtree rooted at v* is the tree whose root is v containing all the descendants of v . The *nearest common ancestor* of two nodes v and w is the node of maximum depth that is an ancestor of both v and w .

An *ordered tree* is a rooted tree such that the children of every node v are totally ordered. A child x of v is to the *left* of another child y , and y is to the *right* of x , if x occurs first in the ordering of the children of v . If no sibling occurs between x and y in the ordering, x is the *left sibling* of y , y is the *right sibling* of x , and x and y are *neighboring siblings*. The first child of a node is its *left* (or *leftmost*) *child* and the last child is its *right* (or *rightmost*) *child*.

The ordering of children imposes an order on any two unrelated nodes v and w ; v is to the *left* of w , and w is to the *right* of v , if there are siblings v' and w' such that v' is to the left of w' , v' is an ancestor of v , and w' is an ancestor of w . This relation totally orders the leaf descendants of any node x ; the *left* (or *leftmost*) *leaf descendant* of x is the first leaf in the ordering and the *right* (or *rightmost*) *leaf descendant* of x is the last. The *left path* from x is the path from x down to its leftmost leaf descendant; the *right path* from x is the path from x down to its rightmost leaf descendant. The *left neighboring leaf* of x is the rightmost leaf (if any) to the left of the leftmost leaf descendant of x ; the *right neighboring leaf* of x is the leftmost leaf (if any) to the right of the rightmost leaf descendant of x .

Appendix B. Keys in a search tree. Let S be a totally ordered set. A *search tree* for S is an ordered tree containing the items of S in its leaves, one item per leaf, in left-to-right order. In order to use the search tree to access S , we must store auxiliary items, called *keys*, in the internal nodes. We shall consider two possibilities. The first is the *single key* representation: if x is an internal node with k children, x contains $k - 1$ keys, called *left keys*, one for each child y of x except the rightmost. The key for y is the largest item in the subtree rooted at y . The second is the *double key* representation: in addition to left keys, every internal node x contains a *right key* for each of its children y except the leftmost. The key for y is the smallest item in the subtree rooted at y . Every item in the tree except the largest occurs exactly once as a left key; every key except the smallest occurs exactly once as a right key. (See Fig. 16.)

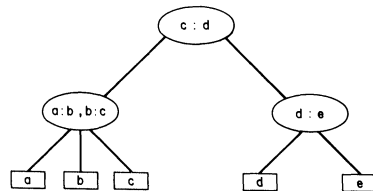


FIG. 16. Keys in a search tree. Items are letters in alphabetical order. Left keys appear before colons, right keys after. A pair of keys $k_1 : k_2$ consisting of a left key k_1 and a right key k_2 corresponds to an open interval of items missing from the tree.

Left keys (or right keys) suffice for searching from the root of a given item i . We initialize the current node x to be the root and repeat the following step until x is a leaf. Then either x contains i or i is not in the tree.

Search step. Select the smallest left key in x no less than i . Replace x by the child y of x corresponding to this key.

Using both left and right keys expedites unsuccessful searches. If i is an item, let i^- be the last item before i and i^+ the first item after i . We define the *handle* of i to be the leaf containing i if i is in the search tree and the nearest common ancestor of the leaves containing i^- and i^+ if not; in this case the handle contains i^- as a left key and i^+ as a right key. A search for i can stop at the handle of i : we terminate the search when the current node x is a leaf or i lies strictly between a left key in x and the next larger right key. To deal with the case of an item smaller than the smallest item in the tree or larger than the largest, we maintain a header for the tree containing its smallest and largest items; then unsuccessful searches for items outside the range of the tree take $O(1)$ time.

Updating a search tree generally requires a sequence of local rebalancing steps, each of which changes the structure of the tree. For the binary search trees, considered

in § 4, we need two symmetric rebalancing steps: a left single rotation and a right single rotation. (See Fig. 17.) A single rotation takes $O(1)$ time.

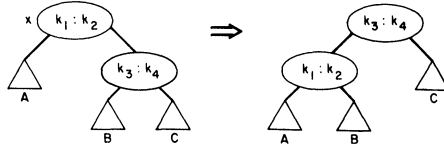


FIG. 17. Updating of keys during a left single rotation at node x . (Right single rotation is symmetric.)

For the search trees whose nodes can have more than two children, considered in § 3, we also need two rebalancing operations: a *split*, which splits an internal node into two neighboring siblings, and its inverse, a *fuse*, which combines two neighboring siblings into one. (See Fig. 18.) Either a split or a fuse requires $O(1)$ time, assuming a fixed upper bound on the maximum number of children of a node.

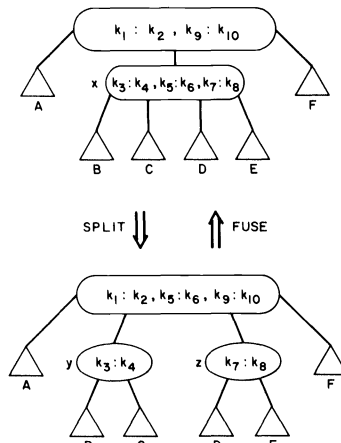


FIG. 18. Splitting a node x . A left key and a right key move up. Inverse operation is fusing nodes y and z .

REFERENCES

- [1] N. ABRAMSON, *Information Theory and Coding*, McGraw-Hill, New York, 1963.
- [2] G. M. ADELSON-VELSKII AND Y. M. LANDIS, *An algorithm for the organization of information*, Soviet Math. Dokl., 3 (1962), pp. 1259-1262.
- [3] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
- [4] J. L. BAER, *Weight-balanced trees*, Proc. AFIPS Nat. Comp. Conf., 44 (1975), pp. 467-472.
- [5] R. BAYER, *Symmetric binary B-trees: data structure and maintenance algorithms*, Acta Inform., 1 (1972), pp. 290-306.
- [6] R. BAYER AND E. M. MCCREIGHT, *Organization and maintenance of large ordered indexes*, Acta Inform., 1 (1972), pp. 173-189.
- [7] S. W. BENT, D. D. SLEATOR AND R. E. TARJAN, *Biased 2-3 trees*, Proc. Twenty-First Annual IEEE Symposium on Foundations of Computer Science, 1980, pp. 248-254.
- [8] S. W. BENT, *Dynamic weighted data structures*, Ph.D. thesis, Computer Science Dept., Stanford Univ., Stanford, CA, 1982.
- [9] J. FEIGENBAUM AND R. E. TARJAN, *Two new kinds of biased search trees*, Bell System Tech. J., 62 (1983), pp. 3139-3158.
- [10] M. L. FREDMAN, *Two applications of a probabilistic search technique: sorting $X + Y$ and building balanced search trees*, Proc. Seventh ACM Symposium on Theory of Computing, 1975, pp. 240-244.
- [11] A. M. GARSIA AND M. L. WACHS, *A new algorithm for minimal binary search trees*, this Journal, 6 (1977), pp. 622-642.

- [12] L. G. GUIBAS AND R. SEDGEWICK, *A dichromatic framework for balanced trees*, Proc. Nineteenth Annual IEEE Symposium on Foundations of Computer Science, 1978, pp. 8–21.
- [13] H. GÜTING AND H. P. KRIEGEL, *Multidimensional B-tree: an efficient dynamic file structure for exact match queries*, Informatik-Fachberichte 33, Springer, Berlin 1980, pp. 375–388.
- [14] ———, *Dynamic k-dimensional multiway search under time-varying access frequencies*, Lecture Notes in Computer Science 104, Springer, Berlin, 1981, pp. 135–145.
- [15] T. C. HU, *Combinatorial Algorithms*, Addison-Wesley, Reading, MA, 1982.
- [16] T. C. HU, D. J. KLEITMAN AND J. K. TAMAKI, *Binary search trees optimum under various criteria*, SIAM J. Appl. Math., 37 (1979), pp. 246–256.
- [17] T. C. HU AND A. C. TUCKER, *Optimal computer-search trees and variable-length alphabetic codes*, SIAM J. Appl. Math., 21 (1971), pp. 514–532.
- [18] D. A. HUFFMAN, *A method for the construction of minimum redundancy codes*, Proc. IRE, 40 (1952), pp. 1098–1101.
- [19] D. E. KNUTH, *Optimum binary search trees*, Acta Inform., 1 (1971), pp. 14–25.
- [20] ———, *The Art of Computer Programming, Vol. 3, Sorting and Searching*, Addison-Wesley, Reading, MA, 1975.
- [21] J. F. KORSCH, *Greedy binary search trees are nearly optimal*, Inform. Proc. Letters, 13 (1981), pp. 16–19.
- [22] H. P. KRIEGEL AND V. K. VAISHNAVI, *Weighted multidimensional B-trees used as nearly optimal dynamic dictionaries*, Mathematical Foundations of Computer Science, Czechoslovakia 1981.
- [23] ———, *A nearly optimal dynamic tree structure for partial-match queries with time-varying frequencies*, Proc. CISS, 1981.
- [24] K. MEHLHORN, *Nearly optimal binary search trees*, Acta Inform., 5 (1975), pp. 287–295.
- [25] ———, *Arbitrary weight changes in dynamic trees*, Bericht 78/04, Angewandte Mathematik und Informatik, Universität des Saarlandes, 1978.
- [26] ———, *Dynamic binary search*, this Journal, 8 (1979), pp. 175–198.
- [27] J. NIEVERGELT AND E. M. REINGOLD, *Binary search trees of bounded balance*, this Journal, 2 (1973), pp. 33–43.
- [28] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 26 (1983), pp. 362–391; see also Proc. Thirteenth Annual ACM Symposium on Theory of Computing, 1981, pp. 114–122.
- [29] ———, *Self-adjusting binary trees*, Proc. Fifteenth Annual ACM Symposium on Theory of Computing, 1983, pp. 235–245.
- [30] ———, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach., to appear.
- [31] R. E. TARJAN, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.
- [32] K. UNTERAUER, *Dynamic weighted binary search trees*, Acta Inform., 11 (1979), pp. 341–362.
- [33] F. F. YAO, *Efficient dynamic programming using quadrangle inequalities*, Proc. Twelfth Annual ACM Symposium on Theory of Computing, 1980, pp. 429–435.

PRIORITY NETWORKS OF COMMUNICATING FINITE STATE MACHINES*

MOHAMED G. GOUDA† AND LOUIS E. ROSIER†

Abstract. Consider a network of two communicating finite state machines which exchange messages over two one-directional, unbounded channels, and assume that each machine receives the messages from its input channel based on some fixed (partial) priority relation. We address the problem of whether the communication of such a network is deadlock-free and bounded. We show that the problem is undecidable if the two machines exchange two types of messages. The problem is also undecidable if the two machines exchange three types of messages, and one of the channels is known to be bounded. However, if the two machines exchange two (or less) types of messages, and one channel is known to be bounded, then the problem becomes decidable. The problem is also decidable if one machine sends one type of message and the second machine sends two (or less) types of messages; the problem becomes undecidable if the second machine sends three types of messages. The problem is also decidable if the message priority relation is empty. We also address the problem of whether there is a message priority relation such that the priority network behaves like a FIFO network. We show that the problem is undecidable in general, and present some special cases for which the problem becomes decidable.

Key words. bounded communication, communicating finite state machines, communication progress, communication protocols, deadlock detection, message passing, priority channels, priority systems, unboundedness detection

1. Introduction. Networks of communicating finite state machines have proven extremely useful in the modeling [4], analysis [1], [2], [23], and synthesis [14], [25] of communication protocols and distributed systems. However, most previous work (cf. [1]–[4], [14], [18], [23]–[25]) has focused on FIFO networks, i.e. networks where each machine receives the messages from its input channel based on the well-known First-In-First-Out discipline. In this paper, we consider instead priority networks where messages are received based on a fixed, partial-ordered priority relation. There are two practical reasons to consider this class of networks:

(i) In a number of existing communication protocols and distributed systems, messages are actually received based on a fixed priority relation rather than a FIFO discipline. (For example, INTERRUPT messages have a higher priority over sequenced messages in the packet layer of X.25 [21].) It is more appropriate to model and analyze such systems using priority networks than FIFO networks.

(ii) In many cases, it is possible to select some fixed priority relation such that the resulting priority network behaves like a FIFO network. (For example in § 7, we show that the Call Establishment and Clear procedures of the Binary Synchronous Protocol [12], which are usually modeled by a FIFO network, can be modeled by a priority network.)

In this paper, we consider a network of two communicating finite state machines that exchange messages over two unbounded, one-directional channels. Each machine has a finite number of states (called nodes) and state transitions (called edges). Each state transition of a machine is accompanied by either sending one message to the output channel of the machine or receiving one message from the input channel of the machine. (Formal definitions are presented later.)

An example of a priority network is shown in Fig. 1. It consists of two machines; machine M is called the requestor and machine N is called the responder. The requestor

* Received by the editors August 23, 1983, and in revised form January 3, 1984. This research was supported in part by the University Research Institute, The University of Texas at Austin and the IBM Corporation.

† Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712.

continuously sends a request message and receives a reply message. There are two types of requests, “regular” and “urgent” (denoted g_1 and g_2 respectively in Fig. 1), and two types of replies, “regular” and “urgent” (denoted g'_1 and g'_2 respectively in Fig. 1). After the requestor sends a g_1 message, it waits to receive a g'_1 message; however it can also send a g_2 message in which case it must first receive the corresponding g'_2 message before receiving the g'_1 message. This implies that g_2 and g'_2 have higher priorities than g_1 and g'_1 respectively. Figure 1c shows the “state reachability graph” of the network. Each node in this graph corresponds to one reachable state of the network, and is labeled by a four-tuple: The first (second) component refers to a node in machine M (N), and the third (fourth) component refers to the contents of the input channel of machine M (N), where E denotes the empty channel. Notice that the only next state after $[3, 1, E, g_1g_2]$ is $[3, 3, E, g_1]$, and not $[3, 2, E, g_2]$; this is because g_2 has a higher priority than g_1 .

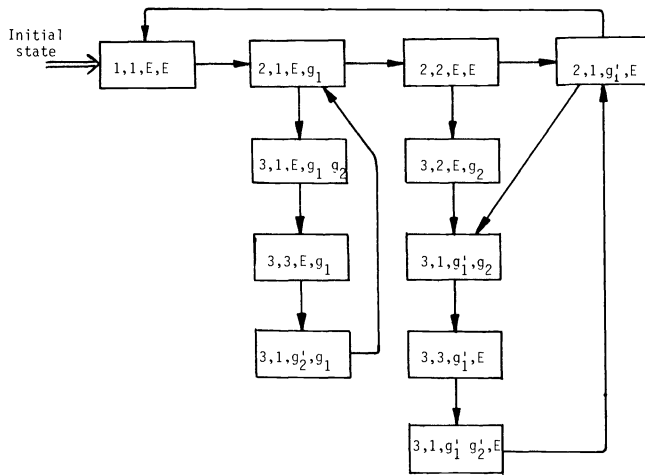
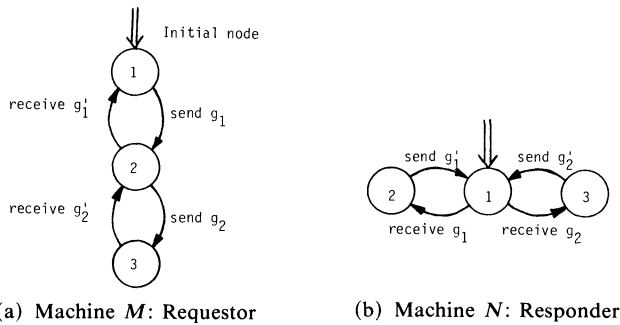


FIG. 1. A priority network example.

This model is equivalent, in computational power, to certain classes of extended Petri nets, in particular those with coloured or priority tokens [6], [16]. However, the model presented here is more concise (since the channels and their contents are not modeled explicitly), and so is more convenient to use in modeling communication protocols and distributed systems.

Our results focus on the problem of whether the communication of a priority network is deadlock-free (i.e. the network can never reach a state after which no further progress is possible), and/or bounded (i.e. the number of reachable states is finite). We provide decidability/undecidability results for these problems with respect to restricted classes of priority networks. We consider restrictions on the number of allowable message types and the size of the priority relation. We also examine the case where one of the channels is known to be bounded. The results presented here define sharp boundaries between the decidable and undecidable cases. They also depart considerably from similar results in the literature concerning FIFO networks [2], [18].

The paper is organized as follows. In § 2, the model of priority networks is presented formally. In § 3, we show that the problem of detecting deadlocks and unboundedness is undecidable even if the machines exchange only two types of messages. We also consider the case where one of the two channels is known to be bounded, and show that three types of messages can make the problem undecidable in this case. (This problem is decidable in the case of FIFO networks [2].) Then in § 4, we show that the same problem becomes decidable if only two types of messages are allowed. In § 5, we consider the case of one of the two machines sending one type of message. We show that the problem is undecidable if the other machine sends three or more types of messages, and is decidable if the other machine sends two or less types of messages. (Both problems are decidable in the case of FIFO networks [18].) However, the latter result can be generalized to the case of three or more messages, if only two message types are mentioned in the priority relation. In § 6, we examine the simplification (or reduction) of the message priority relation. In particular, we argue that if the message priority is reduced, and if the network after the reduction is deadlock-free and bounded, then the network before the reduction is also deadlock-free and bounded. Moreover, if the priority is reduced to the limit (i.e. all messages are received on a random basis), then the problems of detecting deadlocks and/or unboundedness are reduced to the reachability and unboundedness problems of vector addition systems [10], [11], [13], [19], and so are decidable. In § 7, we discuss how to select the message priority relation such that the priority network behaves like a FIFO network.

2. Priority networks. A *message system* is an ordered pair $(G, <)$, where G is a finite, nonempty set of *messages*, and $<$ is a partial order over G called the *message priority relation*. If two distinct messages g_1 and g_2 in G are such that (g_1, g_2) is in $<$, denoted by $g_1 < g_2$, then g_2 is said to have a *higher priority* than g_1 . The number $|G|$ of the messages in set G of a message system is called the *size* of the message system.

A *communicating machine* M over a message system $(G, <)$ is a finite directed labeled graph with two types of edges namely *sending* and *receiving edges*. A sending (receiving) edge is labeled $\text{send}(g)$ ($\text{receive}(g)$) for some message g in G . One of the nodes in M is identified as the *initial node*, and each node in M is reachable by a directed path from the initial node. For convenience, we assume that each node in M has at least one outgoing edge; outgoing edges of the same node have distinct labels. If the outgoing edges of a node are all sending (all receiving), then the node is called a *sending (receiving) node*; otherwise it is called a *mixed node*.

Let M and N be two communicating machines over the same message system $(G, <)$; the pair (M, N) is called a *priority network* of M and N .

A *state* of network (M, N) is a four-tuple $[v, w, x, y]$, where v is a node in M , w is a node in N , and x and y are two multisets of messages in G . Informally, a state $[v, w, x, y]$ of network (M, N) means that the execution of the two machines M and

N has reached nodes v and w (respectively), while the input channels of M and N contain the multisets x and y (respectively) of messages.

The *initial state* of a priority network (M, N) is $[v_0, w_0, E, E]$, where v_0 is the initial node of M , w_0 is the initial node of N , and E is the empty multiset.

Let $s = [v, w, x, y]$ be a state of a priority network (M, N) , and let e be an outgoing edge of node v or w . A state s' of (M, N) is said to *follow s over e* iff one of the following four conditions are satisfied:

(i) e is a sending edge, labeled send (g), from v to v' in M , and $s' = [v', w, x, y']$ where y' is obtained by adding exactly one g to y .

(ii) e is a sending edge, labeled send (g), from w to w' in N , and $s' = [v, w', x', y]$ where x' is obtained by adding exactly one g to x .

(iii) e is a receiving edge, labeled receive (g), from v to v' in M , and x contains at least one g , and $s' = [v', w, x', y]$ where x' is obtained by removing exactly one g from x , and if v has an outgoing edge labeled receive (g'), where $g < g'$, then x contains no g' .

(iv) e is a receiving edge, labeled receive (g), from w to w' in N , and y contains at least one g , and $s' = [v, w', x, y']$ where y' is obtained by removing exactly one g from y , and if w has an outgoing edge labeled receive (g'), where $g < g'$, then y contains no g' .

The last parts of conditions (iii) and (iv) mean that messages are received in accordance with their priorities; the highest priority available message is received first. (Unrelated messages can be received in any order.)

Let s and s' be two states of a priority network (M, N) ; state s' is said to *follow s* iff there exists an edge e in M or N such that s' follows s over e .

Let s and s' be two states of network (M, N) ; state s' is *reachable from s* iff either $s = s'$ or there exist states s_1, s_2, \dots, s_r such that $s = s_1, s' = s_r$, and for $i = 1, \dots, r-1, s_{i+1}$ follows s_i .

A state of network (M, N) is *reachable* iff it is reachable from the initial state of (M, N) .

A reachable state $s = [v, w, x, y]$ of a priority network (M, N) is called a *deadlock state* iff the following three conditions are satisfied:

(i) Both v and w are receiving nodes.

(ii) Either $x = E$ (the empty multiset) or for any message g in x , there is no outgoing edge, from node v , labeled receive (g).

(iii) Either $y = E$ or for any message g in y , there is no outgoing edge, from node w , labeled receive (g).

If no reachable state of network (M, N) is a deadlock state, then the communication of (M, N) is said to be *deadlock-free*.

Let (M, N) be a priority network over $(G, <)$. The input channel of machine $M(N)$ is said to be *bounded* by some positive integer K iff for any reachable state $[v, w, x, y]$ of (M, N) , $|x|(|y|) \leq K$, where $|x|$ is the number of messages in the multiset x . The communication of a network is *bounded* by K iff each of its two channels is bounded by K . If there is no such K , then the communication of (M, N) is *unbounded*.

3. Undecidable results. In the next two sections, we consider the problem of detecting deadlocks and unboundedness for three classes of priority networks:

1. The message system is of size less than or equal two.
2. The message system is of size less than or equal two, and one of the two channels is known to be bounded.

3. The message system is of size greater than or equal three, and one of the two channels is known to be bounded.

In this section, we show that the problem is undecidable for classes 1 and 3, and in § 4, we show that the problem is decidable for class 2. These results are interesting since they depart from the corresponding results for similar classes of FIFO networks. More specifically, the problem is decidable for classes 2 and 3 of FIFO networks, but remains undecidable for class 1 of FIFO networks [2].

THEOREM 1. *It is undecidable whether the communication of a class 1 priority network is both deadlock-free and bounded.*

Proof. We show that any 2-counter machine T [15] (to be defined later), with no input, can be simulated by a priority network (M, N) over a message system $(G, <)$ such that the following three conditions are satisfied:

- (i) $G = \{g_0, g_1\}$, and $< = \{g_0 < g_1\}$.
- (ii) The communication of (M, N) is deadlock-free.
- (iii) The communication of (M, N) is bounded by K iff the values of the two counters of T never exceed K .

Assume that there is an algorithm A to decide whether the communication of any such network is deadlock-free and bounded; then this algorithm can decide whether any 2-counter machine T halts as follows. First, construct from T a priority network which satisfies the above three conditions. Second, apply algorithm A to this network. If the answer is “yes”, then (from conditions (ii) and (iii) above) T has a finite number of reachable configurations and its halting can be decided by exploring all the reachable configurations. If the answer is “no”, then (from (ii) and (iii) above) T does not halt. Since it is undecidable whether any 2-counter machine halts [15], algorithm A cannot exist. It remains now to describe how to simulate any 2-counter machine by a priority network which satisfies the above three conditions; but first we define briefly 2-counter machines.

A 2-counter machine [7], [15] is an offline deterministic Turing machine whose two storage tapes are semi-infinite, and whose tape alphabet contains only two symbols Z and B . The first (leftmost) cells in both tapes are marked with Z symbols; all other tape cells are marked with B (for blank) symbols. Initially, each tape head scans the first cell of its tape. A nonnegative integer “ i ” can be stored at a tape by moving the tape head i cells to the right. Each move of the machine increments (decrements), by one each of the two stored numbers by moving the respective tape head one cell to the right (left). Each move of the machine depends on whether each of the two stored integers is currently greater than or equal zero. This is checked by examining the symbol in the currently scanned cell in each tape: A Z symbol indicates that the integer is zero, whereas a B symbol indicates that it is greater than zero.

Let T be a 2-counter machine; T can be simulated by a priority network (M, N) over $(G, <)$, where $G = \{g_0, g_1\}$ and $< = \{g_0 < g_1\}$. Machine M simulates the finite control of T and N acts as an “echoer” that transmits the contents of its input channel to its output channel. At certain instances, the number of g_1 messages in the network will equal $2^i \cdot 3^j$, where i and j are the two integers currently stored in the counters of T ; a similar encoding technique is used in [15]. The simulation proceeds in phases. First M will process the number represented by the g_1 messages in its input channel. After which, in the next phase, N will send them all back so that M can continue with its next phase. The g_0 messages are used to insure the proper synchronization between the phases of M and N , and to allow each machine to deduce when there are no longer g_1 messages in its input channel (this determines when a phase is over). The details of the simulation can be found in [5]. \square

THEOREM 2. *It is undecidable whether the communication of a class 3 priority network is both deadlock-free and bounded. The result holds even if the message system is of size three and the message priority relation has two elements.*

Proof. As in the proof of Theorem 1, we show that any 2-counter machine T can be simulated by a priority network (M, N) over $(G, <)$ such that the following three conditions are satisfied:

(i) $G = \{g_0, g_1, g_2\}$, $< = \{g_0 < g_1, g_0 < g_2\}$, and the input channel of one machine, say M , is known to be bounded.

(ii) The communication of (M, N) is deadlock-free.

(iii) The communication of (M, N) is bounded by $2K + 6$ iff the values of the two counters of T never exceed K .

Machine M simulates the finite control of T while N acts as a "source" for the new messages to be added to the input channel of M . The number of g_1 (g_2) messages in the input channel of M corresponds to the integer stored in the first (second) counter of T . The g_0 messages are used for synchronization between M and N . Each time M wants to update the values of its counters it sends a g_0 message to N . (This is the only type of message M is allowed to send.) On receiving g_0 , N sends six messages; two of each type. M is now free to update the value of its counters and proceed with the simulation. Further details can be found in [5]. \square

The proofs of Theorems 1 and 2 show that the property of freedom of deadlocks and boundedness is undecidable. The same proofs also show that boundedness (by itself) is undecidable. To show that freedom of deadlocks (by itself) is undecidable, the proofs of Theorems 1 and 2 need to be modified slightly. The simulation of the 2-counter machine T proceeds as discussed before until T halts, in which case M enters a special node that has a self-loop labeled $\text{receive}(g)$ for each message g in G . In other words, T halts iff (M, N) can reach a deadlock. This proves that freedom of deadlocks (by itself) is undecidable.

4. A decidable case: $|G|=2$ and one channel is bounded. In this section, we show that detecting deadlocks and/or unboundedness for class 2 priority networks is decidable. For the sake of discussion in this section, let (M, N) be a class 2 priority network over $(G, <)$ where $G = \{g_1, g_2\}$ and $< = \{g_1 < g_2\}$, and assume that the input channel of one machine, say M , is bounded by the positive integer K .

For i greater than or equal 0, define $A(i)$ to be the set of all states $[v, w, x, y]$ of (M, N) , where multiset y contains at most i occurrences of message g_1 , or i occurrences of message g_2 . In what follows, we will be interested in the three sets $A(0)$, $A(L)$, and $A(2L)$, where $L = (n + K + 2) * \max(m, n)$, m = the number of nodes in machine M , and n = the number of nodes in machine N .

The possible contents of multiset y in each state $[v, w, x, y]$ of (M, N) can be represented by "points" in the space illustrated in Fig. 2. The points of the two g_1 - g_2 axes (or Barrier 0) correspond to the states in $A(0)$. The points between Barrier 1 and Barrier 0 correspond to the states in $A(L)$. The points between Barrier 2 and Barrier 0 correspond to the states in $A(2L)$.

LEMMA 1. *If network (M, N) reaches a state s not in $A(L)$, then starting from s , M can stay "dormant" and N can progress (sending at most n messages) until (M, N) reaches a state in $A(0)$.*

Proof. Assume that (M, N) reaches a state s not in $A(L)$. In this state, the input channel of N has at least $(n + K + 2)n$ messages of each type (g_1 and g_2). Then starting from s , M can stay dormant and N can progress until all occurrences of g_1 or all occurrences of g_2 are completely "depleted" from the input channels of N , i.e. the

network (M, N) reaches a state s' in $A(0)$. Notice that s' can be reached from s after at least $(n + K + 2)n$ steps (executed by N) since it takes at least that many steps to deplete all the messages of one type. It remains now to show that N can send at most n messages during the period from s to s' . This is shown by contradiction.

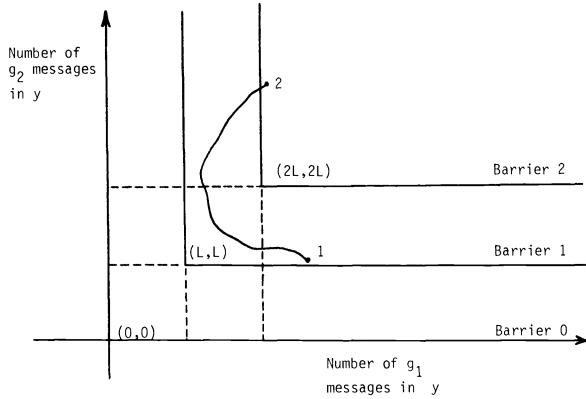


FIG. 2. Possible contents of multiset y in each state $[v, w, x, y]$ of (M, N) .

Assume that during the period from s to s' , N sends $n + 1$ messages, i.e. N traverses $n + 1$ sending edges e_1, \dots, e_{n+1} . Two of these edges say e_a and e_b , must be outgoing edges of the same node d in N ; node d must be in a directed cycle C (of length less than or equal n) that contains the sending edge e_a in N . Also, from the starting state $s = [v, w, x, y]$, there must be a directed path P , from node w to node d , of length less than or equal n . Since N can execute at least $(n + K + 2)n$ steps starting from state $s = [v, w, x, y]$, and since it can execute that many steps along any directed path starting from node w , then assume that N executes along path P then along cycle C for $K + 1$ times. (This is possible since the length of this compound path is less than or equal $(K + 2)n$ which is less than $(n + K + 2)n$.) However, along this compound path the sending edge e_a is traversed $K + 1$ times causing N to send $K + 1$ messages while M is dormant. This contradicts the assumption that the input channel of M is bounded by K . \square

LEMMA 2. If network (M, N) reaches a state s not in $A(L)$ and later reaches a state s' such that none of the reached states from s to s' is in $A(0)$, then N can send at most n messages during the period from s to s' .

Proof (by contradiction). Assume that N sends $n + 1$ messages during the period from s to s' , i.e. N traverses $n + 1$ sending edges e_1, \dots, e_{n+1} during this period. In what follows, we are only concerned with the computation of machine N from s to s' . Now at state s the input channel of N has at least $(n + K + 2)n$ messages of each type. Since e_1 is reachable in the ensuing computation (of N) and no intermediate move of this computation depends on the input channel being devoid of either message type, there must exist a different computation, beginning at the same point which traverses e_1 after at most n steps. (This computation is derived from the original computation by removing all the loops in the computation preceding the traversal of e_1 .) In a similar manner, the resulting computation can be modified further so that N can traverse e_2 after at most n steps from traversing e_1 , and can traverse e_3 after at most n steps from traversing e_2 , and so on. In other words, N can traverse these $n + 1$ sending edges after at most $(n + 1)n$ steps from s . Since $(n + 1)n$ is less than $(n + K + 2)n$,

then N can execute that many steps while M remains dormant. If N executes these $(n+1)n$ steps while M remains dormant, the network will not reach a state in $A(0)$ even though N has sent more than n messages; this contradicts Lemma 1. \square

LEMMA 3. *If network (M, N) reaches a state not in $A(2L)$, then the communication of (M, N) is unbounded.*

Proof. Assume that network (M, N) reaches a state not in $A(2L)$, i.e. a state $[v, w, x, y]$ where the contents of y are beyond Barrier 2. Let point 2, in Fig. 2, correspond to the state of the network where Barrier 2 is first crossed. Also, let point 1, in Fig. 2, correspond to the state of the network when it last crosses Barrier 1 up to the time of point 2. From Lemma 2, at the network progresses from point 1 to point 2, N can send at most n messages. Hence during this period, M can receive at most $n+K$ messages and can send at least $(n+K+2)m$ messages (to increase the contents of its input channel from L to $2L$ so that the network can reach point 2). Therefore, as the network progresses from point 1 to point 2, M must traverse more than m successive sending edges, i.e. it must traverse a cycle whose edges are all sending edges. Thus the communication of (M, N) is unbounded. \square

THEOREM 3. *It is decidable whether the communication of any class 2 priority network is both deadlock-free and bounded.*

Proof. Let (M, N) be any class 2 priority network as defined at the beginning of this section. We show that this network can be simulated by a nondeterministic 1-counter machine T (to be defined), with no input, such that the communication of (M, N) is bounded iff there exists a constant C where the counter value of T never exceeds C . (A nondeterministic 1-counter machine is similar to the deterministic 2-counter machine mentioned in the previous section except that the machine's moves are allowed to be nondeterministic, and the machine has a single counter or tape instead of two [7], [15].) Deciding whether there exists C such that the counter value of a 1-counter machine never exceeds C in every possible computation is Nondeterministic Logspace Complete (in the size of the machine) [18]. (Since these devices are essentially nondeterministic pushdown automata many decision problems are decidable [7]. Other related problems involving 1-counter automata (e.g. equivalence) have been studied. (See e.g. [8], [9], [22].)) Therefore boundedness of (M, N) can be decided, and so both boundedness and freedom of deadlocks can be decided. It remains now to show how to define T from (M, N) .

The finite control of T is capable of remembering (at any instant):

- (i) the current nodes of M and N ,
- (ii) the current contents of the (bounded by K) input channel of M (these can be represented by two integers between 0 and K , one integer for each message type), and
- (iii) the current contents of the (possibly unbounded) input channel of N up to $2L$ messages of each type (these can be represented by two integers between 0 and $2L$, one integer for each message type).

T simulates the network (M, N) by choosing nondeterministically to simulate a move of M or N . Whenever the input channel of N has more than $2L$ messages of each type (indicating that the communication of (M, N) is unbounded by Lemma 3) T enters a state where it continuously increments its counter. Clearly the communication of (M, N) is bounded iff there exists C such that the counter value of T never exceeds C in every possible computation. \square

From the proof of Theorem 3, boundedness (by itself) is decidable for class 2 priority networks. The following theorem shows that freedom of deadlocks (by itself) is also decidable for the same class.

THEOREM 4. *It is decidable whether the communication of any class 2 priority network is deadlock-free.*

Proof. Let (M, N) be any class 2 priority network as defined at the beginning of this section. We show that this network can be simulated by a nondeterministic 1-counter machine T such that the communication of (M, N) is deadlock-free iff T never halts. Deciding whether any nondeterministic 1-counter machine halts is Nondeterministic Logspace Complete (in the size of the machine) [18], and so deadlocks can be detected for class 2 priority networks.

The finite control of T is capable of remembering (at any instant):

- (i) the current nodes of M and N ,
- (ii) the current contents of the (bounded by K) input channel of M ,
- (iii) the current contents of the (possibly unbounded) input channel of N up to L messages of each type, and
- (iv) a string of length at most n over the messages g_1 and g_2 .

T simulates the network (M, N) by choosing nondeterministically to simulate a move of M or N :

(i) *Simulating a move of M .* If the move can be simulated without causing more than L messages of each type to appear in the input channel of N , then the move is simulated directly. Otherwise, T must first simulate enough moves of N to bring down the number of one message type in this channel to $L-1$. However, some of these moves of N may send messages to M . T does not simulate these moves in the usual way; instead each sent message is concatenated to the right-hand side of the string in the finite control of T . By Lemma 2, N cannot send more than n messages until its input channel is depleted of one type of message. At such a time the simulation can proceed directly again. After T has executed the above actions, the simulation of the original move of M can now take place.

(ii) *Simulating a move of N .* If the string in the finite control is empty, the move is simulated directly. Otherwise T moves the leftmost message of the stored string to the bounded channel. (That this may imply executing several moves of N is not important since only the message sent can affect the execution of M .)

Clearly T can reach a halting state iff (M, N) can reach a deadlock state. \square

In the proofs of Theorems 3 and 4, we have assumed that the bound of the bounded channel in any class 2 priority network is known. Suppose, however, that this is not the case. In this case, the bound can be obtained by an unbounded search (by Theorem 1 this is the best we can hope for) using the simulation procedure in the proof of Theorem 4. In this procedure, if the bounded channel is not bounded by some assumed value K , then the simulating 1-counter machine can reach a state where the bounded channel has $K+1$ messages.

5. The case of one machine sending one type of message. Consider a priority network (M, N) over $(G, <)$, where one machine, say N , sends one type of message, i.e. there is a message g in G such that each sending edge in N is labeled $\text{send}(g)$. The other machine M is assumed to send any number of message types from G . Let s_M denote the number of distinct message types sent by M . The decidability of the problem of whether the communication of any such network is both deadlock-free and bounded depends on the value of s_M :

(i) If $s_M = 1$, the the problem can be reduced [3], [24] to the problem of “whether the reachability set of any vector addition system is finite?” which is decidable [11], [13], [19]. (See the next section.)

(ii) If $s_M = 2$, then the problem is decidable as discussed in this section.

(iii) If s_M is greater than or equal 3, then the problem becomes undecidable. This can be shown using an identical proof to that of Theorem 2.

(These results for priority networks are different from the corresponding results for FIFO networks. The problem for FIFO network is always decidable, and in fact Nondeterministic Logspace Complete, regardless of the value of s_M [18].)

THEOREM 5. *Let (M, N) be a priority network over $(G, <)$, and assume that M sends two types of messages g_1 and g_2 , where $g_1 < g_2$, and that N sends one type of message. The communication of (M, N) is unbounded iff one of the following two conditions is satisfied:*

A. *There are two reachable states $s = [v, w, x, y]$ and $s' = [v, w, x', y']$ such that the following three conditions hold:*

- (i) s' is reachable from s .
- (ii) If state s' is reached from s via a state $s'' = [v'', w'', x'', y'']$, then $|y_2''| > 0$, where $|y_2''|$ is the number of g_2 messages in y'' .
- (iii) Either $(|x| \leq |x'| \text{ and } |y_1| \leq |y_1'| \text{ and } |y_2| < |y_2'|)$,
or $(|x| \leq |x'| \text{ and } |y_1| < |y_1'| \text{ and } |y_2| \leq |y_2'|)$,
or $(|x| < |x'| \text{ and } |y_1| \leq |y_1'| \text{ and } |y_2| \leq |y_2'|)$,

where

$|x|$ is the number of messages in x ,

$|y_i|$ ($i = 1, 2$) is the number of g_i messages in y , and

$|y_i'|$ ($i = 1, 2$) is the number of g_i messages in y' .

B. *There are two reachable states $s = [v, w, x, y]$ and $s' = [v, w, x', y']$ such that the following three conditions hold:*

- (i) s' is reachable from s .
- (ii) $|y_2| = |y_2'| = 0$.
- (iii) Either $(|x| \leq |x'| \text{ and } |y_1| < |y_1'|)$, or $(|x| < |x'| \text{ and } |y_1| \leq |y_1'|)$.

Proof. If part. We show that condition A is sufficient for the communication to be unbounded. (Proving that condition B is also sufficient for the communication to be unbounded is similar.) Assume that there are two reachable states $s = [v, w, x, y]$ and $s' = [v, w, x', y']$ of (M, N) such that the three conditions in A hold. From i, state s' is reachable from s over a sequence of directed edges that form a directed cycle C_M (which starts and ends with node v) in M , and a directed cycle C_N (which starts and ends with node w) in N . From (ii) and (iii), M and N can traverse the same two cycles any number of times. From (iii), each time the two cycles are traversed, the number of messages in one channel increases while the number of messages in the other channel remains the same or increases. Therefore, the communication is unbounded.

Only if part. We show that if the communication is unbounded and condition A is not satisfied, then condition B must be satisfied. Assume that the input channel of one machine, say M , is unbounded. Now consider a breadth first expansion of the reachable states of (M, N) , where successors of a state are generated iff they have not been generated earlier in the expansion (i.e. each state generated is distinct). Since the set of reachable states is infinite and in particular M 's input channel is unbounded, there is an infinite sequence of reachable distinct states s_0, s_1, \dots of (M, N) such that the following three conditions hold:

- (i) s_0 is the initial state of (M, N) .
- (ii) For $i = 0, 1, \dots$, s_{i+1} follows s_i .
- (iii) For any K , there is a state $s'' = [v'', w'', x'', y'']$ in the sequence such that $|x''| > K$. Because this sequence is infinite, there must be a node pair (v, w) , where node v is in M , node w is in N , and the pair (v, w) is repeated infinitely often in the states of the

infinite sequence. Since this sequence does not satisfy condition A, it must have an infinite number of states $s'' = [v'', w'', x'', y'']$ where the number of g_2 messages in y'' equals zero. Because there are infinitely many such distinct states, condition B must be satisfied. \square

Based on the above theorem, the following algorithm can decide the boundedness of any priority network (M, N) over $(G, <)$, where N sends one type of message and M sends two types of messages g_1 and g_2 , $g_1 < g_2$:

(i) Let T be a directed rooted tree whose nodes are labeled with reachable states of (M, N) and whose directed edges correspond to the "follow" relation. Initially T has exactly one node labeled with the initial state of (M, N) .

(ii) **while** T has a leaf node n' labeled with a state s' that is followed by some state
do

if node n' has an ancestor node n labeled with state s in T such that s and s' satisfy condition A or B (in Theorem 5)

then stop: The communication of (M, N) is unbounded

else find all the states s_1, \dots, s_r which follow s' ;

add nodes n_1, \dots, n_r to T ;

label each node n_i with state s_i ;

add a directed edge from node n' to each n_i in T ;

(iii) **stop:** The communication of (M, N) is bounded.

Two comments concerning the above algorithm are in order:

(i) The above algorithm is guaranteed to terminate. This is because (from the proof of Theorem 5) every infinite path whose nodes are labeled with distinct states in tree T must reach, after a finite number of nodes, two nodes whose state labels satisfy condition A or B.

(ii) The above algorithm can decide boundedness; hence it can be used to decide both boundedness and freedom of deadlocks. This completes the proof for the following theorem.

THEOREM 6. *It is decidable whether the communication of a priority network, where one machine sends one type of message and the other machine sends two types of messages, is both deadlock-free and bounded.*

Note that the preceding theorem does not address the problem of detecting deadlocks in an unbounded network. Theorems 5 and 6 can be generalized in a straightforward fashion to the class of priority networks where one machine sends one type of message, the other machine sends an arbitrary number of message types, and the message priority relation is a singleton set. They can also be generalized to class 3 priority networks where the priority relation is a singleton set. Note that the class 3 priority network constructed in the proof of Theorem 2 is such that one machine sends one type of message and the message priority relation has two elements. Hence, this is the best that can be done.

6. Priority reduction and the decidability of the random reception discipline. Let $(G, <_1)$ and $(G, <_2)$ be two message systems with the same set of messages. $(G, <_2)$ is called a *priority reduction* of $(G, <_1)$ iff $<_2$ is a subset of $<_1$, i.e. for any two messages g_1 and g_2 in G , if $g_1 <_2 g_2$, then $g_1 <_1 g_2$.

The next theorem, whose proof is straightforward, states that if the priority of a message system for some network is reduced, and if the resulting communication (after the priority reduction) is shown to be deadlock-free and bounded, then the original communication (before the reduction) is also deadlock-free and bounded.

THEOREM 7. *Let R_1 denote a priority network (M, N) over $(G, <_1)$, and R_2 denote a priority network (M, N) over $(G, <_2)$, and assume that $(G, <_2)$ is a priority reduction of $(G, <_1)$. If the communication of R_2 is deadlock-free and bounded, then the communication of R_1 is deadlock-free and bounded. \square*

Theorem 7 is useful iff priority reduction can lead to simpler proofs for freedom of deadlocks and boundedness. From the discussion at the end of § 5, if the priority relation is reduced to a single element, then the problem becomes decidable for priority networks where one machine sends only one type of message and for class 3 priority networks. Also, the next theorem states that if the priorities are reduced to the limit (i.e. all sent messages are of equal priorities, and so are received on a random basis), then the problem of whether the communication is both deadlock-free and bounded becomes decidable.

THEOREM 8 (the random reception theorem). *It is decidable whether the communication of any priority network, with empty message priority relation, is deadlock-free and/or bounded.*

Sketch of the proof. Any priority network (M, N) over $(G, <)$, where $<$ is empty can be simulated [16] by a vector addition system U such that the communication of (M, N) is bounded iff the reachability set of U is finite. Finiteness of the reachability sets of vector addition systems is decidable [10], and so is boundedness for priority networks with empty message priority relations. Therefore, the property of both freedom of deadlocks and boundedness is also decidable.

Also, any priority network with empty message priority relation can be simulated by a vector addition system such that the network can reach a deadlock state iff the reachability set of the vector addition system contains a predefined finite set of vectors [16]. The reachability problem of vector addition systems is decidable [11], [13], [19], and so is freedom of deadlocks (by itself) for priority networks with empty message priority relations. \square

It is straightforward to show that Theorem 8 can be generalized to the case of a priority network with r communicating machines (r greater than or equal 2) provided that the message priority relation is empty.

7. Achieving the FIFO discipline using priorities. From Theorem 1 (or 2), priority networks can simulate any 2-counter machine; therefore they can simulate any FIFO network. In this section, we discuss the following special type of simulation. Given two communicating machines M and N whose message labels are taken from a finite set G of messages, is there a message priority relation $<$ such that the priority network (M, N) over $(G, <)$ “behaves like a FIFO network”? But before we define how a priority network behaves like a FIFO network, we first need to add more structure to the concept of a state of a priority network.

As defined in § 2, a state of a priority network is a four-tuple $[v, w, x, y]$, where both x and y are multisets of messages. We adopt the following convention:

- (i) Both x and y are represented as strings of messages.
- (ii) When a machine M (N) sends a message g , then g is concatenated to the right-hand side of y (x) yielding $y \cdot g$ ($x \cdot g$), where “ \cdot ” is the string concatenation operator.
- (iii) When a machine M (N) receives a message g , then the leftmost occurrence of g in x (y) is removed.

From (i) and (ii), if a message g is to the left of a message g' in x or y , then g must have been sent “before” g' . This implies that the leftmost message in x (y) is the current “oldest” message in x (y). From (iii), whenever a machine M or N receives

a message g , it must receive the oldest available copy of this message. Notice that this convention does not violate the state reachability of a priority network; it merely indicates for any reachable state $[v, w, x, y]$, the order in which the messages in x and y have been sent.

A priority network (M, N) over $(G, <)$ is said to *behave like a FIFO network* iff the following four conditions are satisfied for any reachable state $[v, w, x, y]$ of the network:

(i) If $x = g \cdot x'$ and v has no outgoing edge labeled receive (g), then v has no outgoing edge labeled receive (g') for any other message g' in x' .

(ii) If $y = g \cdot y'$ and w has no outgoing edge labeled receive (g), then w has no outgoing edge labeled receive (g') for any other message g' in y' .

(iii) If $x = g \cdot x'$ and v has an outgoing edge labeled receive (g), then $g' < g$ for any other message g' in x' where v has an outgoing edge labeled receive (g').

(iv) If $y = g \cdot y'$ and w has an outgoing edge labeled receive (g), then $g' < g$ for any other message g' in y' where w has an outgoing edge labeled receive (g').

The question “Given M, N , and G , is there a $<$ such that (M, N) over $(G, <)$ behaves like a FIFO network?” may have a positive or negative answer depending on the given M, N , and G . For example, the answer for the two machines in Fig. 3 is “no”, and the answer for the two machines in Fig. 4 is “yes”. (If the priority network in Fig. 4 behaves like a FIFO network, then it models the call establishment and clear procedures for the Binary Synchronous Protocol [12], where

machine M models the primary station,
 machine N models the secondary station,
 message g_1 is the “initial inquiry” message SYN SYN ENQ,
 message g_2 is the “try again” message SYN SYN WACK,
 message g_3 is the “negative ACK” message SYN SYN NACK,
 message g_4 is the “positive ACK” message SYN SYN ACK, and
 message g_5 is the “clear” message SYN SYN EOT.)

Unfortunately, the above question is undecidable in general. A proof of this can be outlined as follows. Simulate any 2-counter machine T using a priority network (M, N) over $(G, <)$ that behaves like a FIFO network until T reaches a halting state in which case M and N start to execute the two machines in Fig. 3 (i.e. those whose

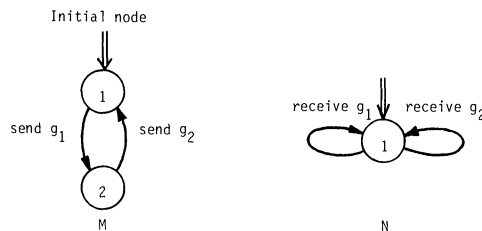


FIG. 3. Two communicating machines whose priority network cannot behave like a FIFO network.

priority network does not, for any $<$, behave like a FIFO network). Thus T halts iff there is no $<$ such that (M, N) over $(G, <)$ behaves like a FIFO network. In [5], we describe a priority network that simulates T while behaving like a FIFO network; this completes the proof of the following theorem.

THEOREM 9. *It is undecidable whether, for any two communicating machines M and N whose message labels are taken from a set G , there exists a message priority relation $<$ such that (M, N) over $(G, <)$ behaves like a FIFO network. \square*

There are special cases for which the above problem becomes decidable. For instance, if the communication between M and N , assuming a FIFO discipline, is bounded (i.e. the number of distinct reachable states is finite), then the problem can be decided by straightforward state exploration. For example, by examining all the reachable states of network (M, N) , in Fig. 4, assuming a FIFO discipline, one can deduce that the priority relation $\leq = \{g_1 < g_2, g_1 < g_3, g_1 < g_4\}$ can make the priority network (M, N) over $(G, <)$ behave like a FIFO network.

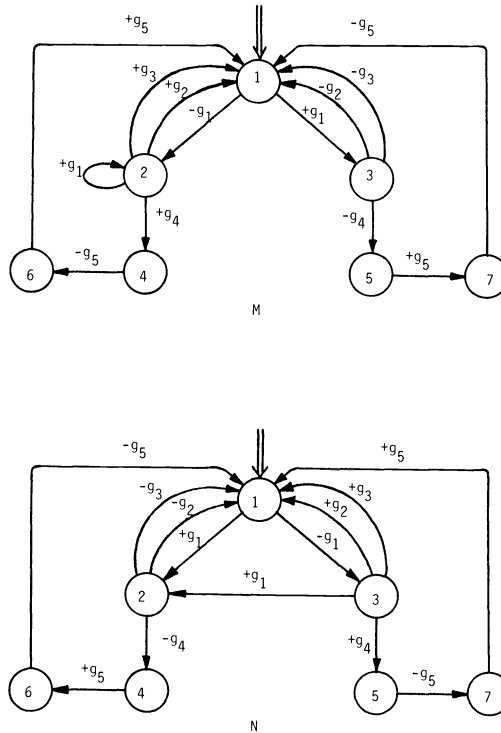


FIG. 4. Two communicating machines whose priority network with message priority $\leq = \{g_1 < g_2, g_1 < g_3, g_1 < g_4\}$ behaves like a FIFO network (Notation: $-g$ means send g , $+g$ means receive g).

This decidability procedure operates on the reachable state space of the network; hence it yields exponential complexity. In some other cases the decidability algorithm needs only to operate on the directed graphs of the two machines yielding polynomial complexity. One such a case is where the two communicating machines are "compatible" as defined next.

Two communicating machines M and N are called *compatible* iff the directed graphs of M and N are isomorphic as follows:

- (i) For every sending (receiving) node in one machine, there is a receiving (sending) node in the other machine.
- (ii) Neither machine has any mixed nodes.
- (iii) For every sending (receiving) edge labeled send (g) (receive (g)) in one machine, there is a receiving (sending) edge labeled receive (g) (send (g)) in the other machine.

If a priority network (M, N) of two compatible machines behaves like a FIFO network, then its communication is guaranteed to be deadlock-free [4]. Moreover, if each directed cycle in M or N has at least one sending and one receiving edge, then

the communication is also bounded [4]. The next theorem states that it is decidable whether a priority network of compatible machines behaves like a FIFO network. The decidability algorithm (in the theorem's proof) operates on the directed graphs of the two machines yielding polynomial complexity.

THEOREM 10. *It is decidable whether for any two compatible machines M and N whose message labels are taken from a set G , there exists a message priority relation $<$ such that (M, N) over $(G, <)$ behaves like a FIFO network.*

Proof. We first present a decidability algorithm for the problem, then prove its correctness. The algorithm consists of three steps:

- (i) Initially, $<$ is the empty set.
- (ii) **for** each receiving node u in M or N , and
 for each two distinct messages g and g' in G
 do
 if there are two outgoing edges labeled receive (g) and receive (g') from u ,
 and if there is a directed path of all receiving edges from the edge labeled
 receive (g) to an edge labeled receive (g')
 then add $g' < g$ to the set $<$
- (iii) **if** the resulting $<$ is a partial order
 then stop: (M, N) over $(G, <)$ behaves like a FIFO network
 else stop: no $<'$ can make (M, N) over $(G, <')$ behave like a FIFO network

If part. Assume that the resulting $<$ of step (ii) is a partial order; we show that (M, N) over $(G, <)$ behaves like a FIFO network. Let $s = [v, w, x, y]$ be the first reachable state, from the initial state, that does not satisfy the definition of "behave like a FIFO network". Without loss of generality, assume that the problem is with the v and x components of s . Therefore, state s must satisfy at least one of the following two conditions:

- (i) $x = g \cdot x'$ where x' contains a message g' distinct from g , and v has an outgoing edge labeled receive (g'), and has no outgoing edge labeled receive (g).
- (ii) $x = g \cdot x'$ where x' contains a message g' distinct from g , and v has two outgoing edges labeled receive (g) and receive (g'), and $g' < g$ is not in $<$.

Since s is the first reachable state that violates the FIFO behavior and since the two machines M and N are compatible, M and N must have reached s via two directed paths p and q such that $p = \langle a, b, \dots, v \rangle$ and $q = \langle a', b', \dots, v', \dots, w \rangle$, where the nodes a', b', \dots , and v' in N correspond to the nodes a, b, \dots and v in M (respectively). Moreover, the directed path $\langle v', \dots, w \rangle$ must consist entirely from the sending edges which have sent the message sequence $x = g \cdot x'$. Thus v' must have an outgoing edge labeled receive (g). Because of the compatibility of M and N , v must have an outgoing edge labeled receive (g). Therefore, condition (i) cannot be satisfied. Also message g' must have been sent along the path $\langle v', \dots, w \rangle$ in N ; it must be expected along the corresponding path of all receiving edges in M . Hence $g' < g$ must have been added to $<$ in step (ii) of the decidability algorithm, and condition (ii) cannot be satisfied.

Only if part. Assume that the resulting $<$ is not a partial order. Assume also that there is a partial order $<'$ such that (M, N) over $(G, <')$ behaves like a FIFO network. Since $<'$ is a partial order while $<$ is not, there must be an element $g' < g$ in $<$ but not in $<'$. In other words, there must be a node v in M or N with two outgoing edges labeled receive (g) and receive (g'), and there must be a directed path of all receiving edges from the edge labeled receive (g) to an edge labeled receive (g'). Without loss of generality, assume that this node v is in M . Because (M, N) over $(G, <')$ behaves

like a FIFO network, and because M and N are compatible, it is possible that the network reaches a state $s = [v, v', E, E]$, where v' is the sending node (in N) that corresponds to the receiving node v in M , and E denotes the empty string. From state s , N can send a sequence x of messages starting with g and ending with g' ($x = g \cdot \dots \cdot g'$) guiding the network into a state $s' = [v, w, x, E]$. This reachable state s' violates condition (ii) which is required for the network to behave like a FIFO network. Therefore, there is no partial order $<'$ such that (M, N) over $(G, <')$ behaves like a FIFO network. \square

REFERENCES

- [1] G. BOCHMANN, *Finite state description of communication protocols*, Computer Networks, 2 (1978), pp. 361-371.
- [2] D. BRAND AND P. ZAFIROPULO, *On communicating finite-state machines*, J. Assoc. Comput. Mach., 30 (1983), pp. 323-342.
- [3] P. CUNHA AND T. MAIBAUM, *A synchronization calculus for message-oriented programming*, Proc. 2nd International Conference on Distributed Computing Systems, April 1981, pp. 433-445.
- [4] M. GOUDA, E. MANNING AND Y. YU, *On the progress of communication between two finite state machines*, Dept. Computer Sciences, Tech. Rep. No. 200, Univ. Texas, May 1982, revised August 1983.
- [5] M. GOUDA AND L. ROSIER, *Priority networks of communicating finite state machines*, Dept. Computer Sciences, Tech. Rep. No. 83-10, Univ. Texas, Austin, August 1983.
- [6] M. HACK, *Decidability questions for Petri nets*, Ph.D. dissertation, Dept. Electrical Engineering, Massachusetts Institute of Technology, Cambridge, 1975.
- [7] J. HOPCROFT AND J. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.
- [8] O. IBARRA, *Reversal-bounded multicounter machines and their decision problems*, J. Assoc. Comput. Mach., 25 (1978), pp. 116-133.
- [9] O. IBARRA AND L. ROSIER, *On restricted one-counter machines*, Math. Systems Theory, 14 (1981), pp. 241-245.
- [10] R. KARP AND R. MILLER, *Parallel program schemata*, J. Comput. System Sci., 3 (1969), pp. 147-195.
- [11] S. KOSARAJU, *Decidability of reachability in vector addition systems*, Proc. 14th Annual ACM Symposium on the Theory of Computing, 1982, pp. 267-281.
- [12] S. LAM, *Data link control procedures*, in Computer Communications, Vol. I, Principles, W. Chou, ed., Prentice-Hall, Englewood Cliffs, NJ, 1983, pp. 81-113.
- [13] E. MAYR, *An algorithm for the general Petri net reachability problem*, Proc. 13th Annual ACM Symposium on the Theory of Computing, 1981, pp. 238-246.
- [14] P. MERLIN AND G. BOCHMANN, *On the construction of communication protocols and module specification*, Pub. 352, Dept. d'Informatique de Recherche Opérationnelle, Université de Montréal, Montréal, Quebec, January 1980.
- [15] M. MINSKY, *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, NJ, 1967.
- [16] J. PETERSON, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [17] C. RACKOFF, *The covering and boundedness problems for vector addition systems*, Theoret. Comput. Sci., 6 (1978), pp. 223-231.
- [18] L. ROSIER AND M. GOUDA, *Deciding progress for a class of communicating finite state machines*, Proc. Eighteenth Annual Conference on Information Sciences and Systems, Princeton Univ., Princeton, NJ, 1984.
- [19] G. SACERDOTE AND R. TENNEY, *The decidability of the reachability problem for vector addition systems*, Proc. of the 9th Annual ACM Symposium on Theory of Computing, 1977, pp. 61-76.
- [20] W. SAVITCH, *Relationships between nondeterministic and deterministic tape complexities*, J. Comput. System Sci., 4 (1970), pp. 177-192.
- [21] A. TANNENBAUM, *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [22] L. VALIANT AND M. PATERSON, *Deterministic one-counter automata*, J. Comput. Sys. Sci., 10 (1975), pp. 340-350.
- [23] Y. YU AND M. GOUDA, *Deadlock detection for a class of communicating finite state machines*, IEEE Trans. Comm., COM-30, 12 (1982), pp. 2514-2518.
- [24] ———, *Unboundedness detection for a class of communicating finite state machines*, Inform. Proc. Lett., 17 (1983), pp. 235-240.
- [25] P. ZAFIROPULO et al., *Towards analyzing and synthesizing protocols*, IEEE Trans. Comm., COM-28, 4 (1980), pp. 651-661.

ON RESTRICTING THE SIZE OF ORACLES COMPARED WITH RESTRICTING ACCESS TO ORACLES*

TIMOTHY J. LONG†

Abstract. A restricted relativization of NP, denoted $NP_B(\cdot)$, was introduced in [SIAM J. Comput., 13 (1984), pp. 461–487] in the study of positive relativizations on the $P = ? NP$ question. The $NP_B(\cdot)$ restriction allows nondeterministic polynomial-time oracle machines to query only polynomially many strings in its computation trees. In this paper we compare the language classes $NP_B(A)$, relative to arbitrary sets A , with the language classes $NP(S)$, relative to sparse sets S , showing that it is not always possible to obtain a class specified by $NP_B(A)$ as an $NP(S)$ class and vice versa. As a corollary to these results, we prove that there is a sparse set S such that for all tally sets T , $S \not\equiv_T^{SN} T$. This implies that the relationship established in [5] that for every sparse set S there is a tally set T such that $S \equiv_T^{NP} T$ cannot be improved to any of the strong nondeterministic polynomial-time degrees. Finally, we strengthen a result appearing in [4] by showing that nondeterministic oracle programs for sets $B \in \Sigma_k^P - \Delta_k^P$, for $k \geq 2$, must search through exponentially many strings (infinitely often) that are actually in the oracle set when using oracle sets from Σ_{k-1}^P . As a consequence, sparse sets at some Σ^P level of the polynomial-time hierarchy cannot be used as oracles for sets properly at the next Σ^P level.

Key words. oracle machines, relativizations, restricted relativizations, sparse sets, tally sets, polynomial-time hierarchy

1. Introduction. In [4], Book, Long and Selman introduced a restricted form, denoted $NP_B(\cdot)$, of the standard relativization of NP. For every set A , $NP_B(A)$ consists of exactly those languages $L \in NP(A)$ for which there is a nondeterministic polynomial time oracle machine M that witnesses $L \in NP(A)$ and a polynomial q such that, for all strings x , M queries the oracle for set A about $q(|x|)$ or fewer distinct strings on input x . The importance of $NP_B(\cdot)$ is due to the fact that it yields a “positive relativization” of the $P = ? NP$ question; that is, $NP_B(\cdot)$ has the property that $P = NP$ if and only if $P(A) = NP_B(A)$ for every set A [4].

Berman and Hartmanis [3] introduced the idea of a polynomially sparse set. (We will use the term sparse set for polynomially sparse set.) Set S is sparse if there is a polynomial p such that the number of strings in S of length at most n is bounded by $p(n)$, for all n . If a nondeterministic polynomial time oracle machine M is using a sparse oracle set S , then the portion of S that M can query, on any input, contains at most polynomially (in the length of the input) many strings. On the other hand, if M' is a nondeterministic polynomial time oracle machine witnessing that a language L is in $NP_B(A)$ for some oracle set A , then the number of strings that M' queries on any input relative to A is polynomially bounded in the length of the input. These two observations suggest that there may be some formal connections between $NP(S)$, for sparse sets S , and $NP_B(A)$, for arbitrary sets A . Further, recent results in Long and Selman [10] and in Balcázar, Book and Schöning [2] not only indicate the importance of sparse oracles, but they also suggest connections between $NP(S)$ and $NP_B(A)$. For example, it is shown in [10] that if there is a sparse set S such that the polynomial-time hierarchy relative to S extends to some level properly past $\Delta_2^{P,S}$, then the unrelativized hierarchy properly extends to the same level. On the other hand, it is shown in [2] that if there is a sparse set S such that the polynomial-time hierarchy relative to S collapses

* Received by the editors October 27, 1983, and in revised form March 19, 1984.

† Department of Computer Science, New Mexico State University, Las Cruces, New Mexico 88003.
Present address, Department of Computer and Information Science, Ohio State University, Columbus, Ohio 43210.

to some level, then the unrelativized hierarchy collapses as well. Thus, questions such as “How many proper levels does the polynomial-time hierarchy contain?” can be relativized by restricting the space of oracle sets to include sparse sets only. The use of sparse oracles is, in this sense, analogous with the use of $NP_B(\cdot)$ in relativizing questions such as $P = ? NP$ and $NP = ? co-NP$.

In this paper we present negative results about the relationships between these two notions by proving that there is a recursive sparse set S such that $NP(S) \neq NP_B(A)$ for all sets A , and by proving that there is a recursive set A such that $NP_B(A) \neq NP(S)$ for all sparse sets S . These results say that it is not always possible to obtain a class of languages that is specified by $NP(S)$ (for sparse sets S) as an $NP_B(A)$ class (for arbitrary sets A) and vice versa.

We also obtain a negative result about the relationship between sparse sets and tally sets (i.e., subsets of $\{1\}^*$) in terms of polynomial-time degrees. Hartmanis [5] has shown that for every sparse set S there is a tally set T such that $S \equiv_T^{NP} T$. ($S \equiv_T^{NP} T$ denotes that $S \leq_T^{NP} T$ and $T \leq_T^{NP} S$, where \leq_T^{NP} denotes nondeterministic polynomial-time Turing reducibility (Ladner, Lynch and Selman [8]).) We show that there is a recursive sparse set S such that for all tally sets T , $S \not\equiv_T^{SN} T$. ($S \not\equiv_T^{SN} T$ denotes that it is not the case that $S \leq_T^{SN} T$ and $T \leq_T^{SN} S$, where \leq_T^{SN} denotes strong nondeterministic polynomial-time Turing reducibility (Long [9]).) Thus, the relationship established by Hartmanis is close to the best possible for arbitrary sparse sets and polynomial degrees.

Finally, we strengthen a theorem appearing in [4] stating that for each $k \geq 2$, if $C \in \Sigma_k^P - \Delta_k^P$, then $C \notin NP_B(A)$ for all $A \in \Sigma_{k-1}^P$. This implies that all nondeterministic polynomial-time programs for C must query exponentially many strings infinitely often when using oracle sets from Σ_{k-1}^P . We prove that for each $k \geq 2$, if $C \in \Sigma_k^P - \Delta_k^P$, then all nondeterministic polynomial-time programs for C must infinitely often query exponentially many strings *that are actually in the oracle set* when using oracle sets from Σ_{k-1}^P . As a consequence, sparse sets at level Σ_k^P , $k > 0$, of the polynomial-time hierarchy cannot be used as oracles for sets properly at level Σ_{k+1}^P . Of course, it is not known if there are sparse sets in the polynomial-time hierarchy that are not also in P .

2. Preliminaries. All sets are assumed to be over the fixed alphabet $\Sigma = \{0, 1\}$, with λ denoting the string of the length 0. Whenever we define a set over a larger alphabet, say $\Gamma = \{0, 1, \#\}$, for example, it is assumed that strings $y \in \Gamma^*$ are coded over Σ in polynomial time. If $A \subseteq \Sigma^*$, then the cardinality of A is denoted by $\|A\|$. If $A, B \subseteq \Sigma^*$, then $A \oplus B = \{x0 \mid x \in A\} \cup \{y1 \mid y \in B\}$. $<$ denotes the standard lexicographic order on Σ^* . We will be coding 2-ary, 3-ary and 4-ary predicates as subsets of Σ^* . These coding functions will always be denoted $\langle \cdot \cdot \cdot \rangle$, and it is assumed that $\langle \cdot \cdot \cdot \rangle$ and all of its inverses are computable in polynomial time.

For any finite set $D \subseteq \Sigma^*$, we let $c(D)$ denote an encoding of D over Σ ; i.e., $c(D) \in \Sigma^*$. It is assumed that for strings $y \in \Sigma^*$ and finite sets $D \subseteq \Sigma^*$, computing $c(D \cup \{y\})$ from $c(D)$ and y and deciding if $y \in D$ from $c(D)$ and y can both be done in time polynomial in $|c(D)| + |y|$. For any $D \subseteq \Sigma^*$ and $n \in \mathbb{N}$ (\mathbb{N} denotes the natural numbers) let $D^{\leq n} = \{x \mid x \in D \text{ and } |x| \leq n\}$. Then, $c(D^{\leq n})$ denotes an encoding of an initial segment of D . A set $S \subseteq \Sigma^*$ is *sparse* if there is a polynomial p such that $\|S^{\leq n}\| \leq p(n)$ for all $n \in \mathbb{N}$. A *tally* set T is any subset of $\{1\}^*$. If $x \in \Sigma^*$, then we can view x as denoting a natural number, denoted n_x , in binary notation. For each $x \in \Sigma^*$ and $A \subseteq \Sigma^*$, let $\text{tally}(x) = 1^{n_x}$ and let $\text{tally}(A) = \{\text{tally}(x) \mid x \in A\}$.

We assume that oracle machines are equipped with the three special states, QUERY, YES and NO, which have their usual meaning. For any oracle machine M and set $A \subseteq \Sigma^*$, the notation M^A denotes that M is using A as its oracle set,

and $L(M, A)$ denotes the language accepted by M relative to A . For each $i \in N$, let p_i denote the polynomial $p_i(n) = n^i + i$ for all $n \in N$. $M_0^{(\cdot)}, M_1^{(\cdot)}, M_2^{(\cdot)}, \dots$ ($NM_0^{(\cdot)}, NM_1^{(\cdot)}, NM_2^{(\cdot)}, \dots$) is an effective enumeration of the deterministic (respectively, nondeterministic) oracle machines that run in polynomial time with polynomial p_i bounding the running time of $M_i^{(\cdot)}$ and $NM_i^{(\cdot)}$. Letting D range over finite subsets of Σ^* , define the set $K = \{\langle NM_i, x, c(D), 0^k \rangle \mid NM_i^D(x) \text{ accepts in } k \text{ or fewer steps}\}$. It is well known that K is an NP-complete set.

For every set A , $NP(A) = \bigcup_{i \in N} L(NM_i, A)$ and $P(A) = \bigcup_{i \in N} L(M_i, A)$. For each class of sets \mathcal{C} , $NP(\mathcal{C}) = \bigcup_{A \in \mathcal{C}} NP(A)$ and $P(\mathcal{C}) = \bigcup_{A \in \mathcal{C}} P(A)$. Then, the polynomial-time hierarchy (PH) is $\{\Sigma_i^P, \Pi_i^P, \Delta_i^P \mid i \in N\}$, where $\Sigma_0^P = \Pi_0^P = \Delta_0^P = P$ and for each $i \geq 0$, $\Sigma_{i+1}^P = NP(\Sigma_i^P)$, $\Pi_{i+1}^P = \text{co-}\Sigma_{i+1}^P = \{\bar{A} \mid A \in \Sigma_{i+1}^P\}$, and $\Delta_{i+1}^P = P(\Sigma_i^P)$. PH was introduced by Meyer and Stockmeyer [13] and further developed in Wrathall [16] and Stockmeyer [15]. For every set $A \subseteq \Sigma^*$, the polynomial-time hierarchy relative to A (PH^A) is $\{\Sigma_i^{P,A}, \Pi_i^{P,A}, \Delta_i^{P,A} \mid i \in N\}$, where $\Sigma_0^{P,A} = \Pi_0^{P,A} = \Delta_0^{P,A} = P(A)$ and for each $i \geq 0$, $\Sigma_{i+1}^{P,A} = NP(\Sigma_i^P)$, $\Pi_{i+1}^{P,A} = \text{co-}\Sigma_{i+1}^P$, and $\Delta_{i+1}^{P,A} = P(\Sigma_i^{P,A})$.

We will use several types of polynomial-time reducibilities:

\leq_m^P denotes polynomial-time many-one reducibility (Karp [7]);

\leq_{tr}^P denotes polynomial-time truth-table reducibility (see Ladner, Lynch and Selman [8]);

\leq_T^{SN} denotes strong nondeterministic polynomial-time Turing reducibility (see Long [9] and Selman [14]);

\leq_c^{NP} denotes nondeterministic polynomial-time conjunctive truth-table reducibility (see [8]);

\leq_T^{NP} denotes nondeterministic polynomial-time Turing reducibility (see [8]).

The definition of \leq_m^P is well known. For \leq_T^{NP} , recall that $A \leq_T^{NP} B$ if and only if $A \in NP(B)$. The exact definitions of \leq_{tr}^P and \leq_c^{NP} are highly technical and not necessary for our purposes. $A \leq_T^{SN} B$ if there is a nondeterministic polynomial-time oracle machine M that recognizes A when using B as its oracle set. As a consequence, $A \leq_T^{SN} B$ if and only if $A \leq_T^{NP} B$ and $\bar{A} \leq_T^{NP} B$. Also, if \leq_α is any reducibility, then $A \equiv_\alpha B$ denotes $A \leq_\alpha B$ and $B \leq_\alpha A$.

For any nondeterministic oracle machine M and sets $A, B \subseteq \Sigma^*$, the notation $A \leq^n B$ via M denotes that $\forall x \in \Sigma^* (|x| \leq n \Rightarrow (M^B(x) \text{ accepts} \Leftrightarrow x \in A^{\leq^n}))$. If $|x| > n$, we do not care what the result of $M^B(x)$ is.

DEFINITION 2.1. For any oracle machine M , any set A and any string x , let $Q(M, A, x)$ be the set of strings y such that in some computation of M in input x relative to A , the oracle is queried about string y .

DEFINITION 2.2. For any oracle machine M , any set A , any input string x , and any integer $k > 0$, let $Q(M, A, x, k)$ be the subset of $Q(M, A, x)$ such that $y \in Q(M, A, x, k)$ if and only if there is a computation of M relative to A on input x that queries the oracle at least k times, and at the k th time that M enters the QUERY state in this computation, y is the string on the query tape.

In the computations of a machine M relative to an oracle set A on input x , $Q(M, A, x, 1)$ is the set of strings queried the first time M reaches a query configuration, $Q(M, A, x, 2)$ is the set of strings queried the second time M reaches a query configuration, etc.

DEFINITION 2.3. For any set A , $NP_B(A)$ is the class of languages L such that $L \in NP(A)$ is witnessed by a machine M such that for some polynomial q and all x , $\|Q(M, A, x)\| \leq q(|x|)$.

$NP_B(\cdot)$ we defined in [4] in the study of positive relativizations of the $P = ? NP$ and $NP = ? \text{co-NP}$ questions. The major results found there are that $P = NP$ if and only

if $P(A) = NP_B(A)$ for all sets A , and $NP = \text{co-NP}$ if and only if $NP_B(A) = \text{co-NP}_B(A)$ for all sets A . The next proposition is from [4, Cor. 5.6] and will be important for some of our proofs.

PROPOSITION 2.4. *For every set A , $NP_B(A) \subseteq P(K \oplus A)$.*

In addition to oracle machines and corresponding language classes, we will also consider oracle transducers and corresponding function classes. A deterministic (non-deterministic) oracle transducer is a deterministic (nondeterministic) oracle machine with distinguished accepting states and a distinguished output tape. An oracle transducer T computes a value y on an input string x using oracle set A if there is an accepting computation of T on x relative to A such that y is the final content of T 's output tape. In general, a deterministic (nondeterministic) oracle transducer computes a partial, single-valued (multivalued) function.

DEFINITION 2.5. For every set A :

(a) $PF(A)$ is the set of all partial, single-valued functions computed by deterministic polynomial-time oracle transducers relative to A .

(b) $NPMV(A)$ is the set of all partial, multivalued functions computed by non-deterministic polynomial time oracle transducers relative to A .

The next proposition is a straightforward relativization of a result in [4, Prop. 3.4].

PROPOSITION 2.6. *For any multivalued function f of one argument, define the single-valued function g as follows:*

$$g(x, 0^k) = \begin{cases} c(\{y \mid y \text{ is a value of } f(x)\}) & \text{if } \|\{y \mid y \text{ is a value of } f(x)\}\| \leq k, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Suppose $f \in NPMV(A)$ via oracle transducer T for some set A . Then there are sets $OKCON_T^A$ and ACC_T such that

- (a) $OKCON_T^A \in NP(A)$,
- (b) $ACC_T \in P$,
- (c) *domain of $g \in P(OKCON_T^A \oplus ACC_T)$, and*
- (d) $g \in PF(OKCON_T^A \oplus ACC_T)$.

Proof. Let $f \in NPMV(A)$ via oracle transducer T with running time bounded by polynomial q . Define the set ACC_T to be the set of all accepting configurations of T . Define the set $OKCON_T^A$ as follows: $\langle x, c(D), I \rangle$ is an element of $OKCON_T^A$, where x is an input word of T , I is a configuration of T , and D is a finite set, if and only if there is a computation of T relative to A , starting from configuration I , that outputs a string y , in $q(|x|)$ or fewer steps, which is not in D . It is easy to verify that $ACC_T \in P$ and that $OKCON_T^A \in NP(A)$.

Assume that T has nondeterministic fan-out two, so that every configuration I of T has at most two successors, left (I) and right (I). The following procedure shows that $g \in PF(OKCON_T^A \oplus ACC_T)$.

```

begin
  input  $x$  and  $0^k$ ;
   $D := \emptyset$ ;
   $j := 0$ ;
   $I_0 :=$  initial configuration of  $T$  on  $x$ ;
  while  $\langle x, c(D), I_0 \rangle \in OKCON_T^A$  and  $j \leq k$  do
    begin (*  $j = \|D\|$  *)
       $I := I_0$ ;
      while  $I \notin ACC_T$  do
        if  $\langle x, c(D), \text{left}(I) \rangle \in OKCON_T^A$ 

```

```

    then  $I := \text{left}(I)$ 
    else  $I := \text{right}(I)$ ;
     $D := D \cup [\text{string on output tape of configuration } I]$ ;
     $j := j + 1$ 
end;
if  $j \leq k$ 
    then halt in an accepting state with  $c(D)$  on the output tape
    else halt in a nonaccepting state
end.

```

The outer while-loop iterates for at most $k + 1$ times. Each execution of the inner while-loop iterates at most $q(|x|)$ times. Thus, the entire procedure runs in polynomial time relative to $\text{OKCON}_T^A \oplus \text{ACC}_T$. Using the loop invariant $j = \|D\|$, it follows that the procedure accepts x and outputs $c(\{y \mid y \text{ is a value of } f(x)\})$ if and only if $\|\{y \mid y \text{ is a value of } f(x)\}\| \leq k$. \square

Finally, for sparse sets S , we will be interested in the function enum_S such that $\text{enum}_S(0^n) = c(S^{\leq n})$ for all $n \in \mathbb{N}$. In general, $\text{enum}_S \notin \text{PF}(S)$. By enriching S , it is possible to obtain an oracle set from which enum_S can be computed in polynomial time.

DEFINITION 2.7. For any set S , define the set $\text{prefix}(S) = \{\langle y, 0^n \rangle \mid \exists z (yz \in S \text{ and } |yz| \leq n)\}$.

$\text{prefix}(S)$ consists of pairs $\langle y, 0^n \rangle$ such that y is a prefix of some string in S of length at most n . It is important to note that $\text{prefix}(S)$ is a sparse set whenever S is. The next proposition uses techniques that appeared in Mahaney [11].

PROPOSITION 2.8. *If S is a sparse set, then $\text{enum}_S \in \text{PF}(\text{prefix}(S) \oplus S)$.*

Proof. The following procedure shows that $\text{enum}_S \in \text{PF}(\text{prefix}(S) \oplus S)$. The idea of the procedure is to use the set $\text{prefix}(S)$ to “guide” the search for elements of S .

```

begin
  input  $0^n$ ;
   $D := \emptyset$ ;
  tail_of_queue :=  $\lambda$ ;
  while queue is not empty do
    begin
       $y := \text{head\_of\_queue}$ ;
      if  $y \in S$  then  $D := D \cup [y]$ ;
      if  $\langle y0, 0^n \rangle \in \text{prefix}(S)$  then tail_of_queue :=  $y0$ ;
      if  $\langle y1, 0^n \rangle \in \text{prefix}(S)$  then tail_of_queue :=  $y1$ 
    end;
  output  $C(D)$ 
end.

```

First consider the running time of the procedure. The fact that S is a sparse set implies that there is a polynomial, say p , such that $\|S^{\leq n}\| \leq p(n)$ for all n . For each n , there are at most $n \cdot p(n)$ pairs of the form $\langle y, 0^n \rangle$ such that $\langle y, 0^n \rangle \in \text{prefix}(S)$. The procedure considers each such pair exactly once. Thus, the number of elements placed on the queue is at most $n \cdot p(n)$ so that the procedure runs in polynomial time relative to $\text{prefix}(S) \oplus S$.

Correctness of the procedure follows from the observation just made; namely, that each pair $\langle y, 0^n \rangle \in \text{prefix}(S)$ is placed on the queue. If $y \in S^{\leq n}$, then $\langle y, 0^n \rangle \in \text{prefix}(S)$. Thus, all $y \in S^{\leq n}$ are added to D . \square

3. Replacing small oracles with restricted access. In this section we consider two questions.

1. Suppose that $C \leq_T^{\text{NP}} S$ for an arbitrary set C and an arbitrary sparse set S . Is it the case that $C \in \text{NP}_B(A)$ for some set A ? In this case we are comparing the reducibility \leq_T^{NP} with sparse oracles to the reducibility $\text{NP}_B(\cdot)$ with arbitrary oracles, starting with a given \leq_T^{NP} -reduction to a sparse set.

2. For each sparse set S , is it the case that there is a set A such that $\text{NP}(S) = \text{NP}_B(A)$? In this case we are comparing the language classes $\text{NP}(\cdot)$ relative to sparse sets with the language classes $\text{NP}_B(\cdot)$ relative to arbitrary sets, starting with a given class $\text{NP}(\cdot)$ relative to a sparse set.

The answer to Question 1 is obviously yes since $C \in \text{NP}_B(C)$ for all sets C . Thus, we consider the further possibility that there is always a sparse set A such that $C \in \text{NP}_B(A)$. Theorem 3.2 shows that this is the case, while Proposition 3.1 warns that A cannot always be the original sparse set S such that $B \leq_T^{\text{NP}} S$.

In [4, Thm. 6.4], a recursive set F is constructed so that $\text{NP}_B(F) \subsetneq \text{NP}(F)$. As constructed, F is actually a sparse set. Thus, letting $S = F$ and letting $C \in \text{NP}(F) - \text{NP}_B(F)$, we immediately have the following proposition.

PROPOSITION 3.1. *There exist a recursive set C and a recursive sparse set S such that $C \leq_T^{\text{NP}} S$ and $C \notin \text{NP}_B(S)$.*

THEOREM 3.2. *For every set C and sparse set S , if $C \leq_T^{\text{NP}} S$, then there is a sparse set S' such that $C \in \text{NP}_B(S')$.*

Proof. Suppose that $C \leq_T^{\text{NP}} S$ via NM_i for some set C and sparse set S . On input x , NM_i can query strings of length at most $p_i(|x|)$. Thus, $x \in L(NM_i, S)$ if and only if $\langle NM_i, x, \text{enum}_S(0^{p_i(|x|)}), 0^{p_i(|x|)} \rangle \in K$. But we know that $\text{enum}_S \in \text{PF}(\text{prefix}(S) \oplus S)$ from Proposition 2.8 and also that $\text{prefix}(S) \oplus S$ is a sparse set. Letting $S' = \text{prefix}(S) \oplus S$ and letting p be a polynomial that bounds the running time of deterministic oracle transducer computing enum_S relative to S' , the following program shows that $C \in \text{NP}_B(S')$.

```

begin
  input  $x$ ;
  using oracle set  $S'$ , deterministically compute  $\text{enum}_S(0^{p_i(|x|)})$ 
  making only  $p(p_i(|x|))$  queries;
  if  $\langle NM_i, x, \text{enum}_S(0^{p_i(|x|)}), 0^{p_i(|x|)} \rangle \in K$ 
  then accept input  $x$ 
end.

```

□

Theorem 3.2 shows that whenever a set C is \leq_T^{NP} -reducible to a sparse set S , then there is another sparse set S' such that the \leq_T^{NP} -reduction of C to S can be replaced by a $\text{NP}_B(\cdot)$ -reduction of C to S' .

We now turn to Question 2 and consider the class $\text{NP}(S)$ for an arbitrary sparse set S . It follows immediately from Theorem 3.2 that there is a set A , in fact a sparse set A , such that $\text{NP}(S) \subseteq \text{NP}_B(A)$. (Just let $A = \text{prefix}(S) \oplus S$.) The question under consideration here is whether there is a set A such that $\text{NP}(S) = \text{NP}_B(A)$. Our next result states that there is no such set A for arbitrary sparse sets S .

Before proceeding to Theorem 3.3, we consider why $\text{NP}(S) \subsetneq \text{NP}_B(\text{prefix}(S) \oplus S)$ for some sparse set S . In general, $\text{prefix}(S) \in \text{NP}(S) = \Sigma_1^{\text{P}, S}$, but $\text{prefix}(S) \notin \text{P}(S)$. In particular, for the S to be constructed in Theorem 3.3, $\text{prefix}(S) \notin \text{P}(S)$. This implies that $\text{NP}_B(\text{prefix}(S) \oplus S)$ will include sets in $\Sigma_2^{\text{P}, S}$ that are not in $\Sigma_1^{\text{P}, S}$ when $\text{NP}(S) \subsetneq \text{NP}_B(\text{prefix}(S) \oplus S)$. (Since $\text{NP}_B(\text{prefix}(S) \oplus S) \subseteq \text{P}(K \oplus (\text{prefix}(S) \oplus S))$,

$NP_B(\text{prefix}(S) \oplus S)$ actually contains sets in $\Delta_2^{P,S} - \Sigma_1^{P,S}$ when $NP(S) \subsetneq NP_B(\text{prefix}(S) \oplus S)$.

THEOREM 3.3. *There exists a sparse set S such that for all sets A , $NP(S) \neq NP_B(A)$.*

Proof. Assume, for now, the existence of a sparse set S such that $NP(K \oplus S) \cap \text{co-}NP(K \oplus S) \subsetneq NP(S)$. The construction of S will be discussed later. Let A be an arbitrary set and consider $NP_B(A)$.

Case 1. $A \notin NP(S)$. In this case, $NP_B(A) \neq NP(S)$, since $A \in NP_B(A) - NP(S)$.

Case 2. $A \in NP(S) - \text{co-}NP(S)$. In this case, $NP_B(A) \neq NP(S)$, since $\bar{A} \in NP_B(A) - NP(S)$.

Case 3. $A \in NP(S) \cap \text{co-}NP(S)$.

Let $L \in NP(S) - NP(K \oplus S) \cap \text{co-}NP(K \oplus S)$. Such an L exists by our assumption about S . Now suppose that $L \in NP_B(A)$. By Proposition 2.4, $L \in P(K \oplus A)$. But $P(K \oplus A) \subseteq NP(K \oplus S) \cap \text{co-}NP(K \oplus S)$, when $A \in NP(S) \cap \text{co-}NP(S)$, implying the contradiction that $L \in NP(K \oplus S) \cap \text{co-}NP(K \oplus S)$. Therefore, $L \in NP(S) - NP_B(A)$ and $NP(S) \neq NP_B(A)$.

Since Cases 1, 2 and 3 cover all possibilities, $NP(S) \neq NP_B(A)$, and since A was arbitrary, $NP(S) \neq NP_B(A)$ for all sets A .

Construction of a sparse set S such that $NP(K \oplus S) \cap \text{co-}NP(K \oplus S) \subsetneq NP(S)$ is done by a straightforward modification of the techniques of Baker, Gill and Solovay [1]. Specifically, this can be done by constructing a sparse set S such that the set $L(S) = \{x \mid \exists y(|y| = |x| \text{ and } y \in S)\}$ is in $NP(S)$ and not in $NP(K \oplus S) \cap \text{co-}NP(K \oplus S)$. \square

Theorem 3.3 also yields information about the relationship between sparse and tally sets in terms of polynomial degrees. We use two lemmas, the first of which is from Selman [14, Thm. 13].

LEMMA 3.4. *For all sets A and B , $NP(A) = NP(B)$ if and only if $A \equiv_T^{SN} B$.*

Proof. Let A and B be arbitrary sets and assume that $NP(A) = NP(B)$. Since A and \bar{A} are both in $NP(A)$ and $NP(A) = NP(B)$, $A \leq_T^{NP} B$ and $\bar{A} \leq_T^{NP} B$. This implies that $A \leq_T^{SN} B$. Similarly, $B \leq_T^{SN} A$.

Now assume that $A \equiv_T^{SN} B$ and let $L \in NP(A)$. It is easy to verify that $L \leq_T^{NP} A$ and $A \leq_T^{SN} B$ imply that $L \leq_T^{NP} B$. Therefore, $L \in NP(B)$ and $NP(A) \subseteq NP(B)$. Similarly, $NP(B) \subseteq NP(A)$ and hence, $NP(A) = NP(B)$. \square

LEMMA 3.5. *For all tally sets T , $NP(T) = NP_B(T)$.*

Proof. $NP_B(A) \subseteq NP(A)$ for all sets A , so in particular, $NP_B(T) \subseteq NP(T)$ for all tally sets T . Now let $L \in NP(T)$ for some tally set T . Observing that $\text{enum}_T \in PF(T)$, the construction used in the proof of Theorem 3.2 can easily be modified to show that $L \in NP_B(T)$. Thus, $NP(T) \subseteq NP_B(T)$ and $NP(T) = NP_B(T)$. \square

THEOREM 3.6. *There exists a recursive sparse set S such that for all tally sets T , $S \not\equiv_T^{SN} T$.*

Proof. Applying Theorem 3.3, let S be a recursive sparse set such that $NP(S) \neq NP_B(A)$ for all sets A . If there is a tally set T such that $S \equiv_T^{SN} T$, then $NP(S) = NP(T)$ by Lemma 3.4. But by Lemma 3.5, $NP(T) = NP_B(T)$, implying that $NP(S) = NP_B(T)$. This is a contradiction to the choice of S , so $S \not\equiv_T^{SN} T$ for all tally sets T . \square

It is interesting to compare the negative result of Theorem 3.6 to the positive relation between sparse and tally sets established by Hartmanis [5]. Reviewing his construction, let S be a sparse set and define the set S' such that $\#n\#i\#j\#t\#d$ is an element of S' if and only if $\exists x, x_1, x_2, \dots, x_i, y_1, y_2, \dots, y_j$ such that

- (a) $x, x_1, x_2, \dots, x_i, y_1, y_2, \dots, y_j$ are all in S ,
- (b) $x_1 < x_2 < \dots < x_i < x \leq y_1 < y_2 < \dots < y_j$,
- (c) $|x| = n$,

- (d) $|y_j| = n$, and
 (e) the t th digit of x is d .

(n, i, j and t are integers represented in binary.) It is easy to see that $\text{tally}(S') \leq_c^{\text{NP}} S$ via a nondeterministic polynomial time transducer that, on input $\text{tally}(\#n\#i\#j\#t\#d)$, nondeterministically guesses $x, x_1, x_2, \dots, x_i, y_1, y_2, \dots, y_j$, deterministically verifies conditions (b)–(e), and then outputs a conjunctive truth-table asking whether each of $x, x_1, x_2, \dots, x_i, y_1, y_2, \dots, y_j$ are in the oracle set. On the other hand, $S \leq_t^{\text{P}} \text{tally}(S')$. To see this, let p be a polynomial such that $\|S^{\leq n}\| \leq p(n)$, let $x = d_1, d_2, \dots, d_{|x|}$, where $d_k \in \{0, 1\}$ for $k = 1, 2, \dots, |x|$, for each i and j , let $F(x, i, j)$ be the formula $\bigwedge_{1 \leq k \leq |x|} \text{tally}(\#|x|\#i\#j\#k\#d_k) \in \text{tally}(S')$, and for each k let $C(x, k)$ be the formula $\text{tally}(\#|x|\#k-1\#1\#1\#0) \in \text{tally}(S') \vee \text{tally}(\#|x|\#k-1\#1\#1\#1) \in \text{tally}(S') \wedge ((\text{tally}(\#|x|\#k\#1\#1\#0) \notin \text{tally}(S')) \wedge \text{tally}(\#|x|\#k\#1\#1\#1) \notin \text{tally}(S'))$. Note that $C(x, k)$ is true if and only if $k = \|S^{\leq |x|}\|$. Then, $x \in S$ if and only if the formula

$$\bigwedge_{k \leq p(|x|)} \left[C(x, k) \wedge \bigvee_{i < k} F(x, i, k-i) \right] \text{ is true.}$$

It follows that $S \equiv_T^{\text{NP}} \text{tally}(S')$. Theorem 3.6 shows that this cannot be strengthened so that S and some tally set T are in the same strong nondeterministic polynomial-time degree.

4. Replacing restricted access with small oracles. In this section we consider two questions analogous to those considered in § 3.

1. Suppose that $C \in \text{NP}_B(A)$ for arbitrary sets A and C . Is it the case that there is a sparse set S such that $C \leq_T^{\text{NP}} S$?

2. For each set A , is there a sparse set S such that $\text{NP}_B(A) = \text{NP}(S)$?

For every set C , $C \in \text{NP}_B(C)$. Thus, Question 1 is actually asking if every set C is \leq_T^{NP} -reducible to a sparse set. In Theorem 4.4, we prove the existence of a recursive set C such that $C \not\leq_T^{\text{NP}} S$ for all sparse sets S . This shows that the answer to both Questions 1 and 2 is no.

The proof of Theorem 4.4 follows the proof of a similar result by Kannan [6]. To state his result, we need the following definitions.

DEFINITION 4.1. A function f is *super-polynomial* if for each integer $k \geq 1$, $\lim_{n \rightarrow \infty} n^k / f(n) = 0$.

DEFINITION 4.2. A function $f(n) \geq n$ is *time-constructible* if there is an $O(f(n))$ -time bounded Turing machine that, on each input of length n , computes $f(n)$ in binary.

PROPOSITION 4.3. [6, Lemma 3]. *If $f(n)$ is a super-polynomial time-constructible function, then $\text{SPACE}(f(n))$ contains a language C such that $C \not\leq_T^{\text{P}} S$, for all sparse sets S .*

(In [6], this lemma is actually stated in terms of small (polynomial) circuits. Here we use the equivalent notion of \leq_T^{P} -reductions to sparse sets [3].) Our proof of the next theorem is a straightforward modification of Kannan's proof to obtain a recursive set C such that $C \not\leq_T^{\text{NP}} S$, for all sparse sets S , as opposed to $C \not\leq_T^{\text{P}} S$, for all sparse sets S .

THEOREM 4.4. *There exists a recursive set C such that $C \not\leq_T^{\text{NP}} S$ for all sparse sets S .*

Proof. Before proceeding with the details of the construction of C , we establish some extra notation and assumptions. For the encoding of finite sets, we assume that there is a polynomial p such that for all finite sets $D \subseteq \Sigma^*$, $|c(D)| \leq p(\|D\| + \max\{|y| : y \in D\})$. For each polynomial p_j , $j \in \mathbb{N}$, set S is p_j -sparse if for all $n \in \mathbb{N}$, $\|S^{\leq n}\| \leq p_j(n)$. For all $i, j \in \mathbb{N}$, define requirement $R_{i,j}$ to be that $C \neq L(NM_i, S)$

for all p_j -sparse sets S . By satisfying $R_{i,j}$ for all $i, j \in N$, it will follow that $C \not\leq_T^{\text{NP}} S$ for all sparse sets S .

The set C is constructed in stages with stage n of the construction determining, for each $x \in \Sigma^*$ such that $|x| = n$, whether $x \in C$ or $x \in \bar{C}$. In addition, letting $n = \langle i, j, k \rangle$, where i, j and k are natural numbers represented in binary, stage n will also try to satisfy requirement $R_{i,j}$.

Consider, for the moment, requirement $R_{i,j}$ with respect to input strings of length n . Because NM_i runs in time p_i , the longest string that NM_i can query, on any input of length n , has length at most $p_i(n)$. Thus, if NM_i is using a p_j -sparse set S as its oracle set, then the portion of S that NM_i can access, on inputs of length n , has size at most $p_j(p_i(n))$; that is, $\|S^{\leq p_i(n)}\| \leq p_j(p_i(n))$. By our assumption about the encoding of finite sets, $|c(S^{\leq p_i(n)})| \leq p(p_j(p_i(n)) + p_i(n))$. It now follows that requirement $R_{i,j}$ could be satisfied using inputs of length n if, for each string y such that $|y| \leq p(p_j(p_i(n)) + p_i(n))$ and such that $y = c(D)$ for some finite set D , there is a string x such that $|x| = n$ and such that it is not the case that $x \in C \Leftrightarrow NM_i^D(x)$ accepts. When some stage n successfully satisfies requirement $R_{i,j}$, this will in fact be the case.

Finally, let $\{0, 1\}^n = \{x_1, x_2, \dots, x_{2^n}\}$. We now proceed with the construction of C .

Stage $n = \langle i, j, k \rangle$

begin

if $p(p_j(p_i(n)) + p_i(n)) \geq 2^n$

then put all strings of length n into C

else begin

$\mathcal{D}_0 := \{c(D) \mid |c(D)| \leq p(p_j(p_i(n)) + p_i(n)) \text{ where}$

$D \text{ ranges over finite subsets of } \Sigma^*\}$;

for $l := 1$ to 2^n **do**

begin

(1) **YES** := $\{c(D) \mid c(D) \in \mathcal{D}_{l-1} \text{ and } NM_i^D(x_l) \text{ accepts}\}$;

(2) **NO** := $\{c(D) \mid c(D) \in \mathcal{D}_{l-1} \text{ and } NM_i^D(x_l) \text{ rejects}\}$;

if $\|\text{YES}\| \geq \|\text{NO}\|$

then begin

(3) **put** x_l into \bar{C} ;

(4) $\mathcal{D}_l := \text{NO}$

end

else begin

put x_l into C ;

$\mathcal{D}_l := \text{YES}$

end

end (* for-loop *)

end (* else *)

end.

To see that every requirement $R_{i,j}$ is satisfied, fix i and j , let k be a natural number such that for $n = \langle i, j, k \rangle$, $p(p_j(p_i(n)) + p_i(n)) < 2^n$, and consider stage n of the construction of C .

PROPERTY I. For each l , $0 < l \leq 2^n$, $\|\mathcal{D}_l\| \leq \|\mathcal{D}_{l-1}\|/2$.

Proof. This is obvious since the sets YES and NO partition \mathcal{D}_{l-1} into two disjoint subsets and since \mathcal{D}_l is then defined to be the smaller of YES or NO. \square

It now follows that $\mathcal{D}_{2^n} = \emptyset$ because $\|\mathcal{D}_0\| \leq 2^{p(p_j(p_i(n)) + p_i(n))}$ and because $\lceil \log_2(\|\mathcal{D}_0\|) \rceil + 1 \leq 2^n$ when $p(p_j(p_i(n)) + p_i(n)) < 2^n$.

PROPERTY II. *If $c(D) \in \mathcal{D}_{l-1}$ and $c(D) \notin \mathcal{D}_l$ for some $l, 0 < l \leq 2^n$, then it is not the case that $x_l \in C \Leftrightarrow NM_i^D(x_l)$ accepts.*

Proof. Suppose that $c(D) \in \mathcal{D}_{l-1}$ and that $c(D) \notin \mathcal{D}_l$ for some l such that $0 < l \leq 2^n$. $c(D) \in \mathcal{D}_{l-1}$ implies that $c(D) \in \text{YES}$ at (1) or that $c(D) \in \text{NO}$ at (2). If $c(D) \in \text{YES}$, then $c(D) \notin \mathcal{D}_l$ implies that \mathcal{D}_l was set to NO at (4) and that x_l was assigned to \bar{C} at (3). Thus, $c(D) \in \text{YES}$ implies that $NM_i^D(x_l)$ accepts and that $x_l \in \bar{C}$. Similarly, if $c(D) \in \text{NO}$, then $NM_i^D(x_l)$ rejects and $x_l \in C$. \square

Each string y such that $y = c(D)$ for some finite set D and such that $|y| \leq p(p_i(p_i(n)) + p_i(n))$ is in the set \mathcal{D}_0 and not in \mathcal{D}_{2^n} since $\mathcal{D}_{2^n} = \emptyset$. Thus, for each such y there is an $l, 0 < l \leq 2^n$, such that $y \in \mathcal{D}_{l-1}$ and $y \notin \mathcal{D}_l$. Applying Property II, for each such y , there is a string x such that $|x| = n$ and it is not the case that $x \in C \Leftrightarrow NM_i^D(x)$ accepts. Thus, requirement $R_{i,j}$ has been satisfied using strings of length n and Theorem 4.4 is proved. \square

COROLLARY 4.5. *There exists a recursive set C such that $NP_B(C) \neq NP(S)$ for all sparse sets S .*

Proof. Use C from Theorem 4.4. Then $C \in NP_B(C)$, but $C \notin NP(S)$ for all sparse sets S . \square

Theorem 4.4 and Corollary 4.5 show that it is not always possible to replace a NP_B -reduction to an arbitrary set with a \leq_T^{NP} -reduction to a sparse set, and that it is not always possible to obtain a class of languages specified by $NP_B(A)$, using arbitrary sets A , as an $NP(S)$ class using a sparse set S .

Finally, we conclude this section with some remarks concerning the proof of Theorem 4.4. First, with careful programming, the construction of C can be done in exponential space. Second, the proof technique generalizes to any class of reduction procedures that

- (i) always halt,
- (ii) are effectively enumerable, and
- (iii) for which there is a polynomial bounding the length of strings queried (as a function of the length of the input).

Thus, there is a recursive set C that is not Turing reducible in polynomial space, for example, to any sparse set S . This last remark was brought to the author's attention by Ker-I Ko.

5. Use of oracles in the polynomial-time hierarchy. In this section we strengthen two results appearing in Book, Long and Selman [4]. These results are restated here in the next two propositions.

PROPOSITION 5.1. [4]. *For each $k \geq 1$, $NP_B(\Sigma_k^P) = \Delta_{k+1}^P$.*

PROPOSITION 5.2. [4]. *For each $k \geq 2$, if $C \in \Sigma_k^P - \Delta_k^P$ then $C \notin NP_B(\Sigma_{k-1}^P)$.*

Proposition 5.2 says that if $C \in \Sigma_k^P - \Delta_k^P$ for some $k \geq 2$, then for every nondeterministic polynomial-time bounded oracle machine M and for every set $A \in \Sigma_{k-1}^P$ such that $C = L(M, A)$, $\|Q(M, A, x)\| > p(|x|)$ for infinitely many x for all polynomials p . Thus, nondeterministic oracle machines for C must be asking exponentially many questions infinitely often when using oracle sets from Σ_{k-1}^P .

Propositions 5.1 and 5.2 will be strengthened by relaxing the NP_B constraint as in the following definition.

DEFINITION 5.3. *For every set A , $NP_{PB}(A)$ is the class of languages $L \in NP(A)$ for which there is a polynomial p and a nondeterministic polynomial-time oracle machine M witnessing $L \in NP(A)$ such that $\|Q(M, A, x) \cap A\| \leq p(|x|)$ for all strings x .*

Notice that $NP_{PB}(\cdot)$ only restricts the number of strings queried that are actually in the oracle set, while $NP_B(\cdot)$ restricts the total number of strings queried. Also, note

that for sparse sets S , $NP(S) = NP_{PB}(S)$ and that for all sets A , $NP_B(A) \subseteq NP_{PB}(A) \subseteq NP(A)$.

THEOREM 5.3. *For each $k \geq 1$, $NP_{PB}(\Sigma_k^P) = \Delta_{k+1}^P$.*

Proof. It is easy to see that $\Delta_{k+1}^P \subseteq NP_{PB}(\Sigma_k^P)$ since $\Delta_{k+1}^P = P(\Sigma_k^P) \subseteq NP_{PB}(\Sigma_k^P)$. To argue that $NP_{PB}(\Sigma_k^P) \subseteq \Delta_{k+1}^P$, let $C \in NP_{PB}(\Sigma_k^P)$. Then there is a polynomial p , a nondeterministic polynomial-time oracle machine NM_i , and a set $A \in \Sigma_k^P$ such that $C = L(NM_i, A)$ and $\|Q(NM_i, A, x) \cap A\| \leq p(|x|)$ for all strings x . We, will prove that $C \in \Delta_{k+1}^P$ by developing a procedure that, on input x , computes the finite set $D = Q(NM_i, A, x) \cap A$ deterministically in polynomial time using an oracle set from Σ_k^P . Once D has been obtained, then $x \in C$ if and only if $\langle NM_i, x, c(D), 0^{p_i(|x|)} \rangle \in K$. Since $k \geq 1$ implies that $K \in \Sigma_k^P$, testing membership in K can also be done deterministically in polynomial time using an oracle set from Σ_k^P . The simulation just outlined shows that $C \in \Delta_{k+1}^P$. This proof technique is used extensively in [4].

Computation of the table D will be accomplished by an iterative process that uses a multivalued function denoted $NEXT_{NM_i}^A$. $NEXT_{NM_i}^A$ is defined as follows: For each input string x of NM_i , finite set F and natural number n , string y is a value of $NEXT_{NM_i}^A(x, c(F), 0^n)$ if

(1) $y \in A$, and

(2) there is a computation of NM_i on input x such that y is the string on NM_i 's oracle tape the n th time that the computation enters the QUERY state and if w is any string queried in this computation prior to the n th query, then the answer to the query about w is YES if and only if $w \in F$.

$NEXT_{NM_i}^A \in NPMV(A)$. In fact, $NEXT_{NM_i}^A \in NPMV(E)$ via some nondeterministic polynomial-time oracle transducer T where E is any set in Σ_{k-1}^P such that $A \leq_T^{NP} E$. The set E can be used to verify (1) nondeterministically in polynomial time. Condition (2) can be checked nondeterministically in polynomial time without using an oracle.

Let g be the function of two arguments in the class $PF(OKCON_T^E \oplus ACC_T)$ obtained from $NEXT_{NM_i}^A$ by application of Proposition 2.6. On input x , the following oracle procedure, when using oracle set $OKCON_T^E \oplus ACC_T$, computes the set $D = Q(NM_i, A, x) \cap A$ deterministically in polynomial-time and then determines if $x \in C$ by using K as an oracle set.

begin

input x ;

$D := \emptyset$;

for $k := 1$ **to** $p_i(|x|)$ **do**

(3) **if** $g(\langle x, c(D), 0^k \rangle, 0^{p(|x|)})$ is defined

(4) **then** $D := g(\langle x, c(D), 0^k \rangle, 0^{p(|x|)}) \cup D$;

if $\langle NM_i, x, c(D), 0^{p_i(|x|)} \rangle \in K$

then accept

else reject

end.

First consider the running time of this procedure. The for-loop iterates for $p_i(|x|)$ times. Each iteration completes execution in polynomial-time relative to $OKCON_T^E \oplus ACC_T$ since $g \in PF(OKCON_T^E \oplus ACC_T)$ and the domain of $g \in P(OKCON_T^E \oplus ACC_T)$. Therefore, the entire procedure runs in deterministic polynomial-time relative to $(OKCON_T^E \oplus ACC_T) \oplus K$. Furthermore, $(OKCON_T^E \oplus ACC_T) \oplus K \in \Sigma_k^P$. To see this, note that $ACC_T \in P$ implies that $ACC_T \in \Sigma_k^P$ for all $k \geq 0$, that $K \in NP$ implies that $K \in \Sigma_k^P$ for all $k \geq 1$, and that $OKCON_T^E \in \Sigma_k^P$ since $E \in \Sigma_{k-1}^P$ and $OKCON_T^E \in NP(E)$ by Proposition 2.6.

Now consider the correctness of the procedure. We claim that

$$(5) \quad D = \bigcup_{j \leq k} Q(NM_i, A, x, j) \cap A \text{ at the end of } k \text{ iterations of the for-loop.}$$

This can be proved by induction on k . When $k = 0$, $D = \emptyset = \bigcup_{j \leq 0} Q(NM_i, A, x, j) \cap A$ so that the base case holds. For the inductive step, assume that (5) is true after $k - 1$ iterations for the for-loop for some $k > 0$ and consider the k th iteration of the for-loop. By the induction assumption,

$$(6) \quad D \text{ contains exactly those strings in } A \text{ that are queried in computations of } NM_i \text{ on } x \text{ relative to } A \text{ during the first } k - 1 \text{ times that these computations enter the QUERY state.}$$

If $g(\langle x, c(D), 0^k \rangle, 0^{p(|x|)})$ is defined at line (3), then its value is the set of strings in A queried the k th time that computations of NM_i enter the QUERY state, subject to the constraint that earlier queries are answered YES if and only if the string being queried is in D . It follows from observation (6) that $D = Q(NM_i, A, x, k) \cap A$ at line (3) and that $D = \bigcup_{j \leq k} Q(NM_i, A, x, j) \cap A$ at line (4). (Also, recall that $\|Q(M, A, X) \cap A\| \leq p(|x|)$.) If $g(\langle x, c(D), 0^k \rangle, 0^{p(|x|)})$ is not defined at line (3), then no computations subject to the same constraint query an element of A the k th time that they enter the QUERY state. Using observation (6) again, $Q(NM_i, A, x, k) \cap A = \emptyset$ so that $D = \bigcup_{j \leq k} Q(NM_i, A, x, j) \cap A$ at line (4).

Thus, (5) holds for all k implying that $D = Q(NM_i, A, x) \cap A$ when execution of the for-loop terminates. The correctness of the procedure is thus established and the proof completed. \square

Theorem 5.3 gives an interesting characterization of Δ_k^P , for $k \geq 2$, in terms of nondeterministic polynomial-time oracle machines. Two corollaries also follow easily from Theorem 5.3.

COROLLARY 5.4. *For each $k \geq 2$, if $C \in \Sigma_k^P - \Delta_k^P$ then $C \notin NP_{PB}(\Sigma_{k-1}^P)$.*

Proposition 5.2 implies that if the polynomial-time hierarchy extends to some level $k \geq 2$ with $\Delta_k^P \subsetneq \Sigma_k^P$, then the hierarchy stands at this level because the power of nondeterminism is necessary for searching through oracle sets when accepting languages in $\Sigma_i^P - \Delta_i^P$ for $2 \leq i \leq k$. Corollary 5.4 extends this by saying that nondeterminism is necessary for searching for *positive information* in oracle sets when accepting languages in $\Sigma_i^P - \Delta_i^P$ for $2 \leq i \leq k$.

COROLLARY 5.5. *For each $k \geq 2$, if $C \in \Sigma_k^P - \Delta_k^P$ then $C \notin NP(S)$ for all sparse sets $S \in \Sigma_{k-1}^P$.*

It is interesting to consider the contrapositive of Corollary 5.5; that is, if $C \in NP(S)$ for some sparse set $S \in \Sigma_{k-1}^P$, then $C \in \Delta_k^P$ for $k \geq 2$.

By noting that $S \in \Sigma_{k-1}^P$ if and only if prefix $(S) \in \Sigma_{k-1}^P$ for $k \geq 2$, a much more direct proof of this corollary suggests itself. Let $C = L(NM_j, S)$. Then $x \in C$ if and only if $\langle NM_j, x, \text{enum}_S(0^{p_j(|x|)}), 0^{p_j(|x|)} \rangle \in K$, and $\text{enum}_S(0^{p_j(|x|)})$ can be computed deterministically in polynomial time relative to prefix $(S) \oplus S$. This idea appears in Mahaney [11]. In addition, if C has some type of polynomial self-reducibility property, then much stronger results may be possible from the hypothesis that $C \in NP(S)$; namely, that C belongs to one of the lower levels of the polynomial-time hierarchy. Results of this type appear in Mahaney and Simon [12].

Acknowledgments. I would like to thank Ker-I Ko and the anonymous referee, both of whom suggested the proof of Theorem 4.4.

REFERENCES

- [1] T. BAKER, J. GILL AND R. SOLOVAY, *Relativizations of the P = NP? question*, this Journal, 4 (1975), pp. 431-442.
- [2] J. BALCÁZAR, R. BOOK AND U. SCHÖNING, *The polynomial-time hierarchy, sparse oracles, and lowness*, manuscript, 1983.
- [3] L. BERMAN AND J. HARTMANIS, *On isomorphisms and density of NP and other complete sets*, this journal, 6 (1977), pp. 305-322.
- [4] R. BOOK, T. LONG AND A. SELMAN, *Quantitative relativizations of complexity classes*, this Journal, 13 (1984), pp. 461-487.
- [5] J. HARTMANIS, *On sparse sets in NP-P*, Inform. Proc. Letters, 16 (1983), pp. 55-60.
- [6] R. KANNAN, *Circuit-size lower bounds and non-reducibility to sparse sets*, Inform. Control., 55 (1982), pp. 40-56.
- [7] R. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, Miller and Thatcher, eds., Plenum, New York, 1972, pp. 85-103.
- [8] R. LADNER, N. LYNCH AND A. SELMAN, *A comparison of polynomial time reducibilities*, Theoret. Comput. Sci., 1 (1975), pp. 103-123.
- [9] T. LONG, *Strong nondeterministic polynomial-time reducibilities*, Theoret. Comput. Sci., 21 (1982), pp. 1-25.
- [10] T. LONG AND A. SELMAN, *Relativizing complexity classes with sparse oracles*, manuscript, 1983.
- [11] S. MAHANEY, *Sparse complete sets for NP: solution to a conjecture of Berman and Hartmanis*, J. Comput. Syst. Sci., 25 (1982), pp. 130-143.
- [12] S. MAHANEY AND J. SIMON, *On sparse sets in PTAPE*, manuscript.
- [13] A. MEYER AND L. STOCKMEYER, *the equivalence of regular expressions with squaring requires exponential space*, in Proc., 13th IEEE SWAT (1972), pp. 125-129.
- [14] A. SELMAN, *Polynomial time enumeration reducibility*, this Journal, 7 (1978), pp. 440-457.
- [15] L. STOCKMEYER, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1-22.
- [16] C. WRATHALL, *Complete sets and the polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 23-33.

ON THE EQUIVALENCE AND CONTAINMENT PROBLEMS FOR UNAMBIGUOUS REGULAR EXPRESSIONS, REGULAR GRAMMARS AND FINITE AUTOMATA*

R. E. STEARNS† AND H. B. HUNT III†

Abstract. The known proofs that the equivalence and containment problems for regular expressions, regular grammars and nondeterministic finite automata are PSPACE-complete [SM] depend upon consideration of highly unambiguous expressions, grammars and automata. Here, we prove that such dependence is inherent.

Deterministic polynomial-time algorithms are presented for the equivalence and containment problems for unambiguous regular expressions, unambiguous regular grammars and unambiguous finite automata. The algorithms are then extended to ambiguity bounded by a fixed k . Our algorithms depend upon several elementary observations on the solutions of systems of homogeneous linear difference equations with constant coefficients and their relationship with the number of derivations of strings of a given length n by a regular grammar.

Key words. unambiguous regular expressions, regular grammars, finite automata, finite state transducers, equivalence and containment problems, homogeneous linear difference equations

1. Introduction. The equivalence and containment problems for regular expressions, regular grammars and nondeterministic finite automata have been extensively studied in the technical literature. Both problems are known to be PSPACE-complete [SM], and thus are probably computationally intractable. Several possible ways to circumvent this intractability have been proposed in the literature. One way is to restrict attention to grammars, expressions and automata that only denote proper subfamilies of the regular sets [B], [Mc], [McP], [Z]. However in [H], we showed that the complexity of the equivalence and containment problems does *not* in general depend upon the structure of the languages denoted. A second way, proposed in [HRS], [H], is to place restrictions on the structure of the grammars, expressions and automata considered, rather than on the languages they denote.

In this paper we consider the structural restriction that the grammars, expressions and automata be *unambiguous*. (Informally, a language descriptor is unambiguous if each of the elements of the language it denotes can only be obtained in one way.) This restriction is very natural since strings in a language often have only one meaning, and one expects that the derivation of a string is attached to its meaning.

We present deterministic polynomial-time algorithms for the equivalence and containment problems for unambiguous regular expressions, regular grammars and nondeterministic finite automata. These algorithms are then generalized to deterministic polynomial-time algorithms for the equivalence and containment problems for regular expressions, regular grammars and nondeterministic finite automata of any *fixed* degree of ambiguity. Evidence is presented that our results are close to the best possible. For example, we show that the equivalence and containment problems for regular expressions, regular grammars and nondeterministic finite automata of bounded degree of ambiguity are CoNP-hard.

Section 2 presents the basic definitions, § 3 the relevant ideas from the theory of difference equations, and § 4 contains the basic results including the polynomial-time equivalence and containment algorithms.

* Received by the editors December 6, 1983 and in final form April 10, 1984.

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

2. Notation and definitions. We assume that the reader is familiar with the standard notation and terminology for regular expressions, regular grammars, deterministic and nondeterministic finite automata, the complexity classes P, NP and PSPACE, and the concepts of polynomial reducibility, NP-hard problems and PSPACE-hard problems. Otherwise see [AHU], [AU]. We denote the sets of natural numbers and real numbers by N and R , respectively.

DEFINITION 2.1. CoNP is the set of all languages over $\{0, 1\}$ that are complements of languages recognizable by nondeterministic polynomially time-bounded Turing machines. A language is CoNP-hard if every language in CoNP is polynomially reducible to it.

It is easily seen that a language L over an alphabet Σ is CoNP-hard if and only if its complement is NP-hard.

Since most of our results are presented in terms of finite automata, we give the definitions of the relevant concepts for them.

DEFINITION 2.2. A nondeterministic finite automaton $M = (S, I, \delta, s_1, F)$, where

- (1) S is a finite nonempty set of *states*;
- (2) I is a finite nonempty set of *input letters*;
- (3) δ is a function from $S \times (I \cup \{\lambda\})$ into the power set of S ;
- (4) $s_1 \in S$ is the *start state*; and
- (5) $F \subset S$ is the set of *accepting states*.

If there is $s \in S$ for which $\delta(s, \lambda)$ is a nonempty subset of S , then the automaton M is said to *have λ -transitions*. If the machine has no λ -transitions, then δ may be regarded as a function from $S \times I$ into the power set of S . If furthermore each $\delta(s, a)$ is a one element set, δ is regarded as a function from $S \times I$ into S and the automaton M is said to be a *deterministic finite automaton*.

The function δ is extended to the domain $S \times I^*$ in the standard manner. The *size* of M , denoted by $|M|$, is defined to equal $|S| \cdot |I|$.

Our definition of the size of M is a bit misleading since it represents the number of values of $\delta(s, a)$ that must be specified but not the space required for the actual specification. Our results hold as long as the size of M is bounded by some fixed polynomial function of $|S|$ and $|I|$.

DEFINITION 2.3. Let $M = (S, I, \delta, s_1, F)$ be a nondeterministic finite automaton. The *language accepted* by M , denoted by $L(M)$, is the set

$$\{w \in I^* \mid \delta(s_1, w) \cap F \neq \emptyset\}.$$

A string w is said to be *accepted* by the automaton M if and only if $w \in L(M)$.

DEFINITION 2.4. Let $M = (S, I, \delta, s_1, F)$ be a nondeterministic finite automaton. By a *state transition sequence* for M , we mean a finite nonempty sequence $\sigma = ((q_1, a_1), \dots, (q_n, a_n), q_{n+1})$, where

- (1) $(q_i, a_i) \in S \times (I \cup \{\lambda\})$ for $1 \leq i \leq n$;
- (2) $q_{n+1} \in S$; and
- (3) $q_{i+1} \in \delta(q_i, a_i)$ for $1 \leq i \leq n$.

The sequence σ is said to be a *state transition sequence* for w , where $w = a_1 \cdots a_n$, that *takes M from state q_1 to state q_{n+1}* . If $q_1 = s_1$ and $q_{n+1} \in F$, then the sequence σ is said to be an *accepting state transition sequence* for w . The *length* of the sequence σ , denoted by $|\sigma|$, equals n . (Thus if the automaton M does *not* have λ -transitions, then $|\sigma| = |w|$.)

Let $k \geq 1$. If, for all $w \in L(M)$, there exist at most k accepting state transition sequences for w , then M is said to be *ambiguous of degree $\leq k$* . If M is ambiguous of degree ≤ 1 , then M is said to be *unambiguous*. If there exists $k \in N$ for which M is ambiguous of degree $\leq k$, then M is said to have bounded degree of ambiguity.

DEFINITION 2.5. Let $M(= S, I, \delta, s_1, F)$ be a nondeterministic finite automaton. The function TRAN-SEQ_M is the function from $S \times N$ to N defined by:

$\text{TRAN-SEQ}_M(s, k)$ for $s \in S$ and $k \in N$ equals the number of state transition sequences of length k which take state s into an accepting state.

The function ACC-SEQ_M is the function from N to N defined by:

$\text{ACC-SEQ}_M(k)$ for $k \in N$ equals the number of accepting state transition sequences of length k .

DEFINITION 2.6. The *equivalence* and *containment problems* for a class C of regular expressions, regular grammars or nondeterministic finite automata are the problems of determining, given $M, N \in C$, if $L(M) = L(N)$ or $L(M) \subset L(N)$, respectively.

We also need the following elementary definition from the difference calculus.

DEFINITION 2.7. Let A be a function from N to R . We say that A *satisfies a homogeneous linear difference equation with constant coefficients of degree n* if and only if there exist constants $c_i \in R$, for $1 \leq i \leq n$, with $c_n \neq 0$ such that

$$\sum_{i=0}^n c_i \cdot A(k+i) = 0 \quad \text{for all } k \geq 0.$$

Henceforth, we abbreviate ‘‘homogeneous linear difference equations with constant coefficients’’ by ‘‘difference equation’’.

Analogues of the concepts of unambiguity, ambiguity of degree $\leq k$, and bounded degree of ambiguity can be defined for regular expressions and for the regular grammars. There exist well-known deterministic polynomial-time algorithms for converting a regular expression or a regular grammar into a nondeterministic finite automaton that accepts the same language [AU], [AHU]. These algorithms preserve the properties of unambiguity, ambiguity of degree $\leq k$ and bounded degree of ambiguity. Hence, we omit further discussion of these facts and present our results in terms of nondeterministic finite automata.

3. Lemmas on difference equations. We present three elementary lemmas on difference equations. In §§ 4 and 5 we use these lemmas to prove the correctness of our deterministic polynomial-time algorithms for equivalence and containment problems.

LEMMA 3.1. *Let S be a nonempty finite set. For all $s \in S$, let A_s be a function from N to R such that*

$$(3.1) \quad A_s(k+1) = \sum_{t \in S} d_{s,t}^1 \cdot A_t(k) \quad \text{for all } k \geq 0,$$

where $d_{s,t}^1$ is a real constant for all $s, t \in S$. Then for all $s \in S$, the function A_s satisfies a difference equation of degree $\leq |S|$.

Proof. We show how to derive the difference equation for one particular A_s . The first step is to obtain $|S|$ difference equations of the form

$$A_s(k+j) = \sum_{t \in S} d_{s,t}^j \cdot A_t(k) \quad \text{for all } k \geq 0$$

one equation for each value of j between 1 and $|S|$.

The equation for $j = 1$ is simply the corresponding equation from set (3.1) (i.e., the equation whose left-hand side is the particular A_s under consideration).

The equations for subsequent j are obtained inductively. Given the equation for $A_s(k+j)$, replace k by $k+1$ to obtain

$$A_s(k+j+1) = \sum_{t \in S} d_{s,t}^j \cdot A_t(k+1).$$

Then, replace each $A_t(k+1)$ using (3.1). The resulting equation is

$$A_s(k+j+1) = \sum_{t \in S} d_{s,t}^{j+1} \cdot A_t(k)$$

where

$$d_{s,t}^{j+1} = \sum_{u \in S} c_{s,u} \cdot d_{u,t}^j.$$

We thus have the desired equation for $j+1$.

The derived $|S|$ equations relate the quantities $A_s(k+1), \dots$, and $A_s(k+|S|)$ with the $|S|$ quantities $A_t(k)$ for $t \in S$. Standard elimination techniques for systems of linear equations can be used to eliminate the quantities $A_t(k)$ for $t \in S - \{s\}$. This is because there are $|S|$ equations and only $|S|-1$ quantities to be eliminated. The resulting difference equation is the equation whose existence is asserted in the statement of the lemma. \square

The construction in the proof of Lemma 3.1 is a standard construction (see any text on difference equations). The resulting equation can be used to obtain a "closed form" formula for $A_s(k)$. Since this formula has constants which are roots of a polynomial equation, the formula can not be considered "closed" from a computational point of view. However, we use the existence of the difference equations only for proofs and not for computation.

LEMMA 3.2. *Let A and B be functions from N to R such that A and B satisfy difference equations of degrees a and b , respectively. Then the function D from N to R defined by, for all $k \in N$, $D(k) = A(k) - B(k)$, satisfies a difference equation of degree $\leq a + b$. Hence, if for $0 \leq k \leq a + b - 1$, $A(k) = B(k)$, then $A(k) = B(k)$ for all $k \in N$.*

Proof. We first show that the function D satisfies a difference equation of degree $\leq a + b$.

By definition for all $k \geq 0$, the function D satisfies the equation

$$D(k+j) = A(k+j) - B(k+j) \quad \text{for } 0 \leq j \leq a + b.$$

Moreover the difference equations satisfied by A and by B allow;

- (1) the replacement of any $A(k+j)$ by a linear combination of $A(k+i)$ for $0 \leq i \leq j$ provided $j \geq a$; and
- (2) the replacement of any $B(k+j)$ by a linear combination of $B(k+i)$ for $0 \leq i \leq j$ provided $j \geq b$. Hence, by repeated applications of [1] and of [2] for all $k \geq 0$,

$$D(k+j) = \sum_{i=0}^{a-1} a_i^j \cdot A(k+i) + \sum_{i=0}^{b-1} b_i^j \cdot B(k+i)$$

for all $j \geq 0$.

As in the proof of Lemma 3.1, elimination of the $a+b$ quantities $A(k+i)$ for $0 \leq i \leq a-1$ and $B(k+i)$ for $0 \leq i \leq b-1$ from the $a+b+1$ equations for $D(k), \dots$, and $D(k+a+b)$ yields a difference equation of degree $\leq a+b$ relating $D(k), \dots$, and $D(k+a+b)$. This equation is the equation whose existence is affirmed in the statement of the lemma.

Now consider the last statement in the lemma. If $A(k) = B(k)$ for $0 \leq k \leq a+b-1$, then $D(k) = 0$ over this range. But since $D(k)$ satisfies an equation of degree $\leq a+b$, all subsequent values of $D(k)$ must be zero and $A(k) - B(k)$ is zero for all k . \square

LEMMA 3.3. *Let $k \geq 1$. Let A_1, \dots , and A_k be functions from N to R such that each function A_i satisfies a difference equation of degree a_i . Let c_1, \dots , and c_k be elements of R . Then the function A from N to R defined by $A(n) = \sum_{i=1}^k c_i \cdot A_i(n)$ for all $n \in N$, satisfies a difference equation of degree $\leq \sum_{i=1}^k a_i$.*

Proof. The proof is a straightforward extension of the techniques used in the proof of Lemma 3.2. \square

4. Unambiguous regular descriptions. We use the observations about systems of difference equations in §3 to derive deterministic polynomially time-bounded algorithms for the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. For automaton M , let $\text{ACC}_M(k)$ represent the number of strings of length k accepted by M . The correctness of our algorithm is based upon the following fact:

Let M be an unambiguous finite automaton with start state s_0 and with no λ -transitions. Then for all $k \geq 0$,

$$\text{ACC}_M(k) = \text{ACC-SEQ}_M(k).$$

Our algorithms are derived and their correctness is proven by a sequence of lemmas. The techniques apply directly to automata without λ -transitions. However, we start with more general automata because they are the output of the standard procedures which produce automata from expressions and grammars. The first lemma in the sequence allows us to consider just the case without λ -transitions.

LEMMA 4.1. *There exists a deterministic polynomially time-bounded algorithm that takes an unambiguous finite automaton $M = (S, I, \delta, s_0, F)$ as input and outputs an equivalent unambiguous finite automaton M' with no λ -transitions and with at most $|S|$ states.*

Proof. The existence of such a deterministic polynomially time-bounded algorithm is implied by the following:

(1) Let \rightarrow_M be the binary relation on S defined by, $s \rightarrow_M t$ for $s, t \in S$ if and only if $t \in \delta(s, \lambda)$. Let \rightarrow_M^* be the transitive reflexive closure of \rightarrow_M . Relations \rightarrow_M and \rightarrow_M^* are computable deterministically from M in polynomial time.

(2) Let M' be the nondeterministic finite automaton (S, I, δ', s_0, F') , where δ' is defined by, for all $s \in S$ and $a \in I$, $\delta'(s, a) = \{t'' \in S \mid \exists t' \in S \text{ for which } s \rightarrow_M^* t' \text{ and } t'' \in \delta(t', a)\}$, and $F' = \{s \in S \mid \exists t \in F \text{ for which } s \rightarrow_M^* t\}$. Machine M' is unambiguous, has no λ -transitions, and is equivalent to M . \square

LEMMA 4.2. *Let $M = (S, I, \delta, s_0, F)$ be a nondeterministic finite automaton with no λ -transitions. Let $s \in S$. Let $k \in N$. Then*

$$\text{TRAN-SEQ}_M(s, k+1) = \sum_{t \in S} a_{s,t} \cdot \text{TRAN-SEQ}_M(t, k), \quad \text{where } a_{s,t} = |\{a \in I \mid t \in \delta(s, a)\}|.$$

Proof. The proof is immediate from the definition of an accepting state transition sequence, since by assumption the automaton M has no λ -transitions. \square

LEMMA 4.3. *Let $n \in N$. For any nondeterministic n state finite automaton M with no λ -transitions, the function ACC-SEQ_M satisfies a difference equation of degree $\leq n$.*

Proof. The equations in the statement of Lemma 4.2 are of the form of Lemma 3.1 if we let the set S be the set of M 's states. For all $t \in S$, let $A_t(k) = \text{TRAN-SEQ}_M(t, k)$. Letting s_0 be the start state of M , $A_{s_0}(k) = \text{ACC-SEQ}_M(k)$. \square

Lemma 4.3 is illustrated by Example 4.10. The reader may want to study this example before continuing.

LEMMA 4.4. *There exists a deterministic polynomially time-bounded algorithm that, given as input a nondeterministic finite automaton $M = (S, I, \delta, s_0, F)$ with no λ -transitions and a string 1^n , gives as output the first n values of ACC-SEQ_M .*

Proof. The obvious way to compute the first n values of ACC-SEQ_M is:

- (1) to first compute the values of $\text{TRAN-SEQ}_M(s, 0)$ for all $s \in S$; and then,
- (2) to perform an iteration n times in which the values of $\text{TRAN-SEQ}_M(s, i+1)$

are computed from the values of $\text{TRAN-SEQ}_M(s, i)$ for $s, i \in S$ by means of Lemma 4.2. For each $s \in S$,

$$\text{TRAN-SEQ}_M(s, 0) = \begin{cases} 1 & \text{if } s \in F, \\ 0 & \text{otherwise.} \end{cases}$$

The coefficients $a_{s,t}$ from Lemma 4.2 have size at most $|I|$ and the sum has $|S|$ terms. Since the $\text{TRAN-SEQ}_M(s, 0)$ are at most 1, it follows by induction that the $\text{TRAN-SEQ}_M(s, k)$ are at most $(|I| \cdot |S|)^k$. Thus all values computed during the algorithm are positive integers whose length in binary is at most $n \cdot \log_2(|I| \cdot |S|)$. Thus, all multiplications and additions can be accomplished in time polynomial in $|M|$ and n . Since the number of operations executed during the computation is also bounded by a polynomial in $|M|$ and n , the computation can be accomplished deterministically in time polynomial in $|M|$ and n . \square

LEMMA 4.5. *There exists a deterministic polynomially time-bounded algorithm that, given as input an n_1 state unambiguous finite automaton M_1 and an n_2 state unambiguous finite automaton M_2 both with no λ -transitions, gives as output an $n_1 \cdot n_2$ state unambiguous finite automaton M with no λ -transitions such that*

$$L(M) = L(M_1) \cap L(M_2).$$

Proof. Letting $M_1 = (S_1, I_1, \delta_1, s_1, F_1)$ and $M_2 = (S_2, I_2, \delta_2, s_2, F_2)$, construct

$$M = (S, I, \delta, s, F) \quad \text{where } S = S_1 \times S_2, \quad I = I_1 \cup I_2,$$

$\delta((t_1, t_2), a) = (\delta_1(t_1, a), \delta_2(t_2, a))$ for $a \in I_1 \cap I_2$, $s = (s_1, s_2)$, and $F = F_1 \times F_2$.

The accepting state transition sequences for this machine are precisely those that project into accepting state transition sequences for M_1 and M_2 , so the number of ways M accepts a string w is the product of the ways M_1 accepts w and M_2 accepts w . Because M_1 and M_2 are unambiguous, they each accept a string in either zero or one ways and so M accepts in one way if both M_1 and M_2 accept and M does not accept otherwise. Thus M is unambiguous and $L(M) = L(M_1) \cap L(M_2)$. \square

THEOREM 4.6. *There exists a deterministic polynomially time-bounded algorithm that, given as input an n_1 state unambiguous finite automaton M_1 and an n_2 state unambiguous finite automaton M_2 such that $L(M_1) \subset L(M_2)$, decides if the set containment is proper.*

Proof. By Lemma 4.1 the automata M_1 and M_2 can be converted deterministically in polynomial-time into equivalent unambiguous finite automata M'_1 and M'_2 with no λ -transitions such that M'_1 is also an n_1 state automaton and M'_2 is also an n_2 state automaton.

Since $L(M_1) \subset L(M_2)$ by assumption, the set containment is proper if and only if $\text{ACC-SEQ}_{M'_1}(k) \neq \text{ACC-SEQ}_{M'_2}(k)$ for some k . By Lemma 4.3 the functions $\text{ACC-SEQ}_{M'_1}$ and $\text{ACC-SEQ}_{M'_2}$ satisfy difference equations of degrees n_1 and n_2 , respectively. Thus by Lemma 3.2 the functions $\text{ACC-SEQ}_{M'_1}$ and $\text{ACC-SEQ}_{M'_2}$ are equal if and only if, for all natural $k < n_1 + n_2$, $\text{ACC-SEQ}_{M'_1}(k) = \text{ACC-SEQ}_{M'_2}(k)$. By Lemma 4.4 this can be checked deterministically in polynomial-time. \square

COROLLARY 4.7. *The equivalence and containment problems for unambiguous finite automata are decidable deterministically in polynomial-time.*

Proof. The corollary follows immediately from Lemma 4.5 and Theorem 4.6, since for finite automata M_1 and M_2 , $L(M_1) \subset L(M_2)$ if and only if the language $L(M_1) \cap L(M_2)$ is not properly contained in the language $L(M_1)$. \square

This corollary combines with a result of Gurari and Ibarra [GI] to give the following result: There is a deterministic polynomially time-bounded algorithm which decides if two single-valued unambiguous finite state transducers are equivalent.

Finally, let M_1 and M_2 be unambiguous finite automata such that $L(M_2) - L(M_1) \neq \emptyset$. It is natural to ask the question:

Can a string $w \in L(M_2) - L(M_1)$ be found deterministically in time polynomial in $|M_1| + |M_2|$?

The answer to this question is "yes" as shown by the next two theorems.

THEOREM 4.8. *There exists a deterministic polynomial-time algorithm that, given as input an n_1 state unambiguous finite automaton M_1 and an n_2 state unambiguous finite automaton M_2 such that $L(M_1)$ is properly contained in $L(M_2)$, gives as output a string $w \in L(M_2) - L(M_1)$ of minimal length such that $|w| < n_1 + n_2$.*

Proof. As in the proof of Theorem 4.6, we can assume that the automata M_1 and M_2 have no λ -transitions. As shown in the proof of Theorem 4.6, the language $L(M_1)$ is properly contained in the language $L(M_2)$ if and only if there exists a string $w \in L(M_2) - L(M_1)$ such that $|w| < n_1 + n_2$. Moreover such a string w of length $j < n_1 + n_2$ exists if and only if $\text{ACC-SEQ}_{M_1}(j) \neq \text{ACC-SEC}_{M_2}(j)$.

Let $M_1 = (S_1, I, \delta_1, s_1, F_1)$ and $M_2 = (S_2, I, \delta_2, s_2, F_2)$ be unambiguous finite automata with no λ -transitions such that $L(M_2)$ properly contains $L(M_1)$. Let $I = \{a_1, \dots, a_m\}$ be of cardinality m . The following algorithm, given inputs M_1 and M_2 , outputs a string w of minimal length such that $w \in L(M_2) - L(M_1)$ and $|w| < |S_1| + |S_2|$:

- (1) Compute $j_0 = \min \{j \mid \text{ACC-SEQ}_{M_1}(j) \neq \text{ACC-SEC}_{M_2}(j)\}$.
- (2) $x \leftarrow \lambda$.
- (3) $A_x \leftarrow \{s_1\}$ and $B_x \leftarrow \{s_2\}$.
- (4) If $|x| = j_0$, then halt with output "x".
- (5) For $1 \leq i \leq m$,
 $A_{xa_i} \leftarrow \{s' \in S_1 \mid \text{there exists } s \in A_x \text{ for which } s' \in \delta_1(s, a_i)\}$
 $B_{xa_i} \leftarrow \{t' \in S_2 \mid \text{there exists } t \in B_x \text{ for which } t' \in \delta_2(t, a_i)\}$.
- (6) For $1 \leq i \leq M$,

$$d_i = \sum_{t' \in B_{xa_i}} \text{ACC-SEQ}_{M_2}(t, j_0 - |x| - 1) - \sum_{s' \in A_{xa_i}} \text{ACC-SEQ}_{M_1}(s', j_0 - |x| - 1).$$

- (7) Letting i_0 be the least i such that $d_i \neq 0$, $x \leftarrow xa_{i_0}$.
- (8) Go to 4.

The algorithm is deterministic and polynomially time-bounded since:

- (a) $j_0 < |S_1| + |S_2|$; and j_0 can be computed from M_1 and M_2 deterministically in polynomial-time by Lemma 4.4 and Theorem 4.6.
- (b) The loop consisting of statements (4), (5), (6), (7) and (8) is executed j_0 times.
- (c) For all $s \in S$, $t \in S_2$ and $i, k \leq j_0$, the integers $\text{ACC-SEC}_{M_1}(s, i)$ and $\text{ACC-SEQ}_{M_2}(t, k)$ can be computed from M_1 and M_2 deterministically in polynomial-time by arguments analogous to the proofs of Lemmas 4.3 and 4.4. The correctness of the algorithm follows from the induction hypothesis:

for $0 \leq j \leq j_0$, after j times around the loop consisting of statements (4), (5), (6), (7) and (8), $|x| = j$ and there exists a string y of length $j_0 - j$ such that $xy \in L(M_2) - L(M_1)$.

THEOREM 4.9. *There exists a deterministic polynomial-time algorithm that, given unambiguous finite automata M_1 and M_2 such that $L(M_2) - L(M_1) \neq \emptyset$, outputs a string $w \in L(M_2) - L(M_1)$ of minimal length.*

Proof. The theorem follows from Lemma 4.5 and Theorem 4.8 since $L(M_2) - L(M_1) \neq \emptyset$ if and only if the containment of $L(M_1) \cap L(M_2)$ in $L(M_2)$ is proper. \square

An example. We present an example of an unambiguous finite automaton M , together with the computation of the difference equation for ACC-SEQ $_M$.

Problem. Find the difference equation for the function ACC-SEQ $_M$ for the unambiguous finite automaton M without λ -transitions, with the start state A , input alphabet $\{0, 1\}$ and set of accepting states $\{C\}$ whose state transition function δ is given by

$$\begin{aligned}\delta(A, 0) &= \{A\}, & \delta(A, 1) &= \{A, B\}, & \delta(B, 0) &= \{C\}, & \delta(B, 1) &= \{C\}, \\ \delta(C, 0) &= \emptyset, & \delta(C, 1) &= \emptyset.\end{aligned}$$

It is easily verified that $L(M) = \{0, 1\}^* \cdot \{1\} \cdot \{0, 1\}$.

Step 1. Write the equations from Lemma 4.2.

$$\text{TRAN-SEQ}_M(A, k+1) = 2 \cdot \text{TRAN-SEQ}_M(A, k) + \text{TRAN-SEQ}_M(B, k).$$

$$\text{TRAN-SEQ}_M(B, k+1) = 2 \cdot \text{TRAN-SEQ}_M(C, k).$$

$$\text{TRAN-SEQ}_M(C, k+1) = 0.$$

Step 2. Obtain the equations for TRAN-SEQ $_M(A, k+1)$, TRAN-SEQ $_M(A, k+2)$ and TRAN-SEQ $_M(A, k+3)$ as in the proof of Lemma 3.1.

$$\text{TRAN-SEQ}_M(A, k+1) = 2 \cdot \text{TRAN-SEQ}_M(A, k) + \text{TRAN-SEQ}_M(B, k).$$

$$\begin{aligned}\text{TRAN-SEQ}_M(A, k+2) &= 2 \cdot \text{TRAN-SEQ}_M(A, k+1) + \text{TRAN-SEQ}_M(B, k+1) \\ &= 4 \cdot \text{TRAN-SEQ}_M(A, k) + 2 \cdot \text{TRAN-SEQ}_M(B, k) \\ &\quad + 2 \cdot \text{TRAN-SEQ}_M(C, k).\end{aligned}$$

$$\begin{aligned}\text{TRAN-SEQ}_M(A, k+3) &= 2 \cdot \text{TRAN-SEQ}_M(A, k+2) + \text{TRAN-SEQ}_M(B, k+2) \\ &= 2 \cdot [4 \cdot \text{TRAN-SEQ}_M(A, k) + 2 \cdot \text{TRAN-SEQ}_M(B, k) \\ &\quad + 2 \cdot \text{TRAN-SEQ}_M(C, k)] + 0 \\ &= 8 \cdot \text{TRAN-SEQ}_M(A, k) + 4 \cdot \text{TRAN-SEQ}_M(B, k) \\ &\quad + 4 \cdot \text{TRAN-SEQ}_M(C, k).\end{aligned}$$

Step 3. Eliminate TRAN-SEQ $_M(B, k)$ and TRAN-SEQ $_M(C, k)$.

In this case, we need only subtract twice the second equation from the third to obtain

$$\text{TRAN-SEQ}_M(A, k+3) - 2 \cdot \text{TRAN-SEQ}_M(A, k+2) = 0.$$

Since ACC-SEQ $_M(k) = \text{TRAN-SEQ}_M(A, k)$ for all $k \geq 0$, the difference equation for ACC-SEQ $_M$ is

$$\text{ACC-SEQ}_M(k+3) - 2 \cdot \text{ACC-SEQ}_M(k+2) = 0 \quad \text{for } k \geq 0.$$

By direct observation, ACC-SEQ $_M(0) = \text{ACC-SEQ}_M(1) = 0$ and ACC-SEQ $_M(k) = 2^{k-1}$ for $k \geq 2$, thus satisfying the derived difference equation.

5. Descriptions of ambiguity $\leq k$. For all $k \geq 1$, we show that there exists deterministic polynomially time-bounded algorithms for the equivalence and containment problems for regular expressions, regular grammars and nondeterministic finite automata of degree of ambiguity $\leq k$.

The basic idea of this section is that, although $ACC_M(k)$ is no longer equal to $ACC-SEQ_M(k)$, we can still find a difference equation for ACC_M of sufficiently low degree that the ideas of the previous section carry over.

We begin with two observations which help establish the difference equation for ACC_M . The observations are proved in Lemma 5.3.

Observation 5.1. Let $k \geq 1$. Let $M = (S, I, \delta, s_1, F)$ be a nondeterministic finite automaton of degree of ambiguity $\leq k$ with no λ -transitions. Let $|S| = n$. Let $1 \leq l \leq k$. Then there exists an $O(n^l)$ state nondeterministic finite automaton M_l constructable from M deterministically in polynomial-time such that, for all $m \in N$,

$$ACC-SEQ_{M_l}(m) = \sum_{j=1}^k \binom{j}{l} \cdot l! \cdot M(m, j),$$

where $M(m, j)$ is the number of words of length m accepted by M by exactly j distinct derivations.

Observation 5.2. Let M, k, l, m and M_l be as in Observation 5.1. Then there exist rational constants c_1, \dots, c_k , depending only on k and not on M , such that, for all $m \in N$, $ACC_M(m) = c_1 \cdot ACC-SEQ_{M_l}(m) + \dots + c_k \cdot ACC-SEQ_{M_k}(m)$.

LEMMA 5.3. *Observations 5.1 and 5.2 are correct.*

Proof. We first sketch the proof of Observation 5.1.

Let $M = (S, I, \delta, s_1, F)$, l, k and m be as in the statement of Observation 5.1. The nondeterministic finite automaton $M_l = (S', I, \sigma', s'_1, F')$, where

- (1) $S' = S \times S \times \dots \times S \times (0, 1) \times (0, 1) \times \dots \times (0, 1)$, i.e. S' consists of l copies of S , l^2 copies of $(0, 1)$;
- (2) $s'_1 = (s_1, s_1, \dots, s_1, a_{11}, \dots, a_{ll})$, where $a_{ij} = 1$, if $i = j$ and $a_{ij} = 0$, otherwise;
- (3) $F' = \{(s_{i_1}, \dots, s_{i_p}, a_{11}, \dots, a_{ll}) \mid \text{where } s_{i_1}, \dots, s_{i_p} \in F \text{ and each } a_{ij} = 1\}$; and
- (4) the function δ' is defined by, $\delta'((t_1, \dots, t_b, a_{11}, \dots, a_{ll}), a) = \{(u_1, \dots, u_b, b_{11}, \dots, b_{ll}) \mid u_i \in \delta(t_i, a), b_{rs} = 1 \text{ if and only if } a_{rs} = 1 \text{ or } u_r \neq u_s\}$.

Clearly, $|M_l| = O(2^{l^2} \cdot |M|^l) = O(|M|^l)$, since $l \leq k$ is a constant. Also the automaton M_l is constructable from M deterministically in polynomial time.

Intuitively, a state of S' represents the status of l derivations of M . The first l components represent the current state of the individual derivations and the subsequent l^2 components represent bits which indicate if two individual derivations were ever distinct. Bits a_{ii} are included so as to keep the notation from becoming too obscure and are always set to 1. The automaton starts with all derivations in the starting state and all distinct pairs marked as identical by setting the corresponding bits to 0. A sequence is accepted if all derivations end in an accepting state and all pairs have been marked as distinct.

By construction, $w \in L(M_l)$ if and only if there are $j \geq 1$ distinct accepting transition sequences of M for w . In which case, there are $\binom{j}{l} \cdot l!$ distinct accepting transition sequences of M_l for w . This follows since the accepting transition sequences of M_l for w correspond exactly to the $l!$ permutations of the $\binom{j}{l}$ different combinations of l different accepting transition sequences of M for w . Hence for all $n \in N$,

$$ACC-SEQ_{M_l}(n) = \sum_{j=1}^k \binom{j}{l} \cdot l! \cdot M(n, j)$$

as claimed.

To prove Observation 5.2, we first observe that

$$ACC_M(n) = \sum_{j=1}^k M(n, j)$$

since the ambiguity of M is bounded by K . This equation together with the k equations of Observation 5.1 can be used to eliminate the $M(n, j)$ and obtain an equation of the form required in Observation 5.2. \square

LEMMA 5.4. *There exists a deterministic polynomially time-bounded algorithm that, given as input a nondeterministic finite automaton $M = (S, I, \delta, s, F)$ outputs an equivalent nondeterministic finite automata M' with no λ -transitions having at most $|S|$ states and degree of ambiguity \cong the degree of ambiguity of M .*

Proof. The proof is identical to that of Lemma 4.1. \square

LEMMA 5.5. *Let $k \geq 1$. Let $n \in \mathbb{N}$. For any nondeterministic n state finite automaton M of degree of ambiguity $\cong k$ with no λ -transitions, the function $\text{ACC}_M(n)$ satisfies a difference equation of degree $O(n^k)$.*

Proof. By Observation 5.2, the function $\text{ACC}_M(n)$ is a linear combination of the functions $\text{ACC-SEQ}_M, \dots$, and ACC-SEQ_{M_k} . By Lemma 4.3 and Observation 5.1, the functions $\text{ACC-SEQ}_M, \dots$, and ACC-SEQ_{M_k} satisfy difference equations of degrees $O(n), \dots$, and $O(n^k)$, respectively. Thus by Lemma 3.3, the function $\text{ACC}_M(n)$ satisfies a difference equation of order $O(k \cdot n^k) = O(n^k)$. \square

LEMMA 5.6. *Let $k \geq 1$. There exists a deterministic polynomially time-bounded algorithm that, given as input a nondeterministic finite automaton $M = (S, I, \delta, s_1, F)$ of degree of ambiguity $\cong k$ with no λ -transitions and a string 1^n , outputs the first n values of the function ACC_M .*

Proof. The machines M_2 described in Observation 5.1 can be constructed, and the algorithm from the proof of Lemma 4.4 can be used to compute the first n values of ACC-SEQ_{M_1} for $1 \leq l \leq k$. These values are combined using Observation 5.2 to obtain the values for ACC_M . \square

LEMMA 5.7. *Let $k \geq 1$. There exists a deterministic polynomially time-bounded algorithm that, given as input an n_1 state nondeterministic finite automaton M_1 and an n_2 state nondeterministic finite automaton M_2 , both of degree of ambiguity $\cong k$ with no λ -transitions, outputs an $n_1 \cdot n_2$ state nondeterministic finite automaton M with no λ -transitions and of degree of ambiguity $\cong k^2$ such that*

$$L(M) = L(M_1) \cap L(M_2).$$

Proof. The proof is identical to that of Lemma 4.5. \square

THEOREM 5.8. *Let $k \geq 1$. There exists a deterministic polynomially time-bounded algorithm that, given as input an n_1 state nondeterministic finite automaton M_1 and an n_2 state nondeterministic finite automaton M_2 , both of degree of ambiguity $\cong k$ and such that $L(M_1) \subset L(M_2)$, the algorithm decides if the set containment is proper.*

Proof. By Lemma 5.4 we may assume that M_1 and M_2 have no λ -transitions. Since $L(M_1) \subset L(M_2)$ by assumption, the set containment is proper if and only if $\text{ACC}_{M_1}(m) \neq \text{ACC}_{M_2}(m)$ for some $m \in \mathbb{N}$. By Lemma 5.5 the functions ACC_{M_1} and ACC_{M_2} satisfy difference equations of degrees $\cong c \cdot n_1^k$ and $c \cdot n_2^k$, respectively, where c is a constant independent of M_1 or of M_2 . Thus by Lemma 3.2, the functions ACC_{M_1} and ACC_{M_2} are equal if and only if, for all natural numbers $m < c \cdot n_1^k + c \cdot n_2^k$, $\text{ACC}_{M_1}(m) = \text{ACC}_{M_2}(m)$. By Lemma 5.6 this can be checked deterministically in polynomial-time. \square

COROLLARY 5.9. *Let $k \geq 1$. The equivalence and containment problems for the nondeterministic finite automata of degree of ambiguity $\cong k$ are decidable deterministically in polynomial-time.*

Proof. For finite automata M_1 and M_2 , $L(M_1) \subset L(M_2)$ if and only if the language $L(M_1) \cap L(M_2)$ is not properly contained in the language $L(M_2)$. Thus the corollary follows immediately from Lemma 5.7 and Theorem 5.8. \square

Finally, we note the following:

THEOREM 5.10. *For all $k \geq 1$, there is a deterministic polynomially time-bounded algorithm to decide if a nondeterministic finite automaton is ambiguous of degree $\leq k$.*

Proof. Let $k \geq 1$. Let M be a nondeterministic finite automaton. Then M is ambiguous of degree $\leq k$ if and only if $L(M_{k+1}) = \emptyset$, where M_{k+1} is the finite automaton constructed from M as in the proof of Lemma 5.3. For a fixed k , this construction can be accomplished deterministically in polynomial time. It is well known that the emptiness problem for nondeterministic finite automata is decidable deterministically in polynomial-time. \square

6. Extensions that fail. In § 5, we extended the results of § 4 to k -bounded ambiguity. It is natural to ask if the results of § 4 can be extended further by considering other classes of finite automata that include the unambiguous automata but do not include all finite automata. Here we show that two extensions suggested by our earlier results fail (unless $P = NP$). This provides evidence that our deterministic polynomially time-bounded algorithms are close to being as generally applicable as possible.

First, we consider the class B of nondeterministic finite automata of bounded degree of ambiguity. The algorithms of § 5 are not directly applicable since the degrees of the polynomials that bound their runtimes grow unboundedly with k . Unless $P = NP = \text{CoNP}$, no deterministic polynomially time-bounded algorithm exists for the equivalence problem or for the containment problem for the class B .

THEOREM 6.1. *The equivalence and containment problems for the class B are CoNP-hard.*

Proof. In [HRS] we showed that the equivalence and containment problems for regular expressions containing only the operators \cup and \cdot are CoNP-complete. Clearly such expressions are of bounded degree of ambiguity. It is easy to show that there is a deterministic polynomially time-bounded algorithm that converts a regular expression containing only the operators \cup and \cdot into an equivalent nondeterministic finite automata of bounded degree of ambiguity. (The standard constructions for converting expressions preserve bounded degree of ambiguity.) \square

The next extension is based on the following definition.

DEFINITION 6.2. Let M be a nondeterministic finite automaton with input alphabet Σ . Then $\text{STATE}(M)$ equals the number of states of M ; and $\text{SHORT}(M)$ equals -1 , if $L(M) = \Sigma^*$, and equals the length of a shortest string in $\Sigma^* - L(M)$, otherwise.

In those cases where we have polynomial algorithms, STATE and SHORT are polynomially related.

THEOREM 6.3.

- (1) *For all unambiguous finite automata M , $\text{SHORT}(M) \leq \text{STATE}(M)$.*
- (2) *Let $k \geq 1$. Then there exists a polynomial p_k such that, for all nondeterministic finite automata M of degree of ambiguity $\leq k$, $\text{SHORT}(M) \leq p_k(\text{STATE}(M))$.*

Proof. These assertions follow from the proofs of Theorems 4.6 and 5.8 taking machine M_2 to be the one state machine which accepts all input sequences. \square

This result is in sharp contrast to the general case described by the following result.

THEOREM 6.4. *There exists $c > 0$ such that, for infinitely many nondeterministic finite automata M ,*

$$\text{SHORT}(M) > 2^{c \cdot \text{STATE}(M)}$$

Proof. This result is suggested by the work of other authors (see [H] and [N]). The construction in [SM] used to prove regular expression equivalence PSPACE-complete can be made into such an example by using an lba which accepts 1^n only

after making 2^n moves. In [H, Prop. 3.9] this is discussed. [N] shows that an exponential distinguishing sequence is required to distinguish two arbitrary automata, but not to distinguish one automaton from the automaton which accepts all sequences. However, we have found a variation on the example given in [N] which proves our result. Because of [H], we consider the proof to be already in the literature and omit further discussion. \square

It is also true that, even for machines with single letter alphabets, there is no polynomial relationship between $\text{SHORT}(M)$ and $\text{SIZE}(M)$. An easy variation on an example in [N] proves this. This is also implied by the proof in [SM] that the predicate " $L(R) = \{0\}^*$ " is CoNP-complete for regular expressions over the alphabet $\{0\}$.

Theorem 6.3 suggests that some condition on class C of nondeterministic finite automata such as:

$$(6.1) \quad \text{For all } M \in C, \quad \text{SHORT}(M) \cong \text{STATE}(M)$$

might suffice to insure that the equivalence and containment problems for C are decidable deterministically in polynomial-time. Again unless $P = NP = \text{CoNP}$, this is not the case.

THEOREM 6.5. *Let C be the class of nondeterministic finite automata that simultaneously are of bounded degree of ambiguity and satisfy condition (6.1). Then the equivalence and containment problems for C are CoNP-hard.*

Proof. The class C contains the class of all nondeterministic finite automata of bounded degree of ambiguity that recognize finite sets. (A sequence accepted by such an automaton cannot repeat a state and so an accepted sequence of n inputs must visit $n + 1$ distinct states and all accepted strings must be shorter than $\text{STATE}(M)$.) Therefore in particular, C contains the set of machines obtained from expressions involving \cup and \cdot as used in the proof of Theorem 6.1. But this same proof showed that the equivalence and containment problems for this latter class of finite automata were shown CoNP-hard. \square

7. Relative economy of description. We now contrast three descriptions of regular sets: the nondeterministic finite automata, the unambiguous finite automata and the deterministic finite automata. For each pair of descriptions, the economy of description is exponential. These results extend and complement the results on the relative economy of description of regular set descriptions in [MF].

THEOREM 7.1. *There exists $c > 0$ such that for infinitely many nondeterministic finite automata M and any equivalent unambiguous finite automaton N , $|N| > 2^{c \cdot |M|}$.*

Proof. The value of c and set of automata satisfying Theorem 6.4 also satisfy this theorem. To see this, let M be one of those automata and N an equivalent unambiguous automaton. Then by Theorems 6.3 and 6.4 and the definition of equivalence,

$$|N| \cong \text{STATE}(N) \cong \text{SHORT}(N) = \text{SHORT}(M) > 2^{c \cdot |M|}. \quad \square$$

THEOREM 7.2. *For all $n \geq 1$, there is an unambiguous finite automaton M_n with $n + 2$ states and of size $2 \cdot (n + 2)$ such that any equivalent deterministic finite automaton N_n has $\geq 2^{n+1}$ states and is of size $\geq 2^{n+2}$.*

Proof. For all $n \geq 1$, $M_n = (S_n, \{0, 1\}, \delta_n, s_0, \{s_{n+1}\})$, where

$$(1) \quad S_n = \{s_0, s_1, \dots, s_{n+1}\}; \text{ and}$$

$$(2) \quad \delta_n(s_0, 0) = \{s_0\}, \delta_n(s_0, 1) = \{s_0, s_1\}, \text{ for } 1 \leq i \leq n \quad \delta_n(s_i, 0) = \delta_n(s_i, 1) = \{s_{i+1}\}, \text{ and } \delta_n(s_{n+1}, 0) = \delta_n(s_{n+1}, 1) = \emptyset.$$

The only way M_n accepts is to stay in state s_0 for some initial string, move to state s_1 using input 1, and reach a single final state s_{n+1} after exactly n move inputs. Thus $L(M_n) = \{0, 1\}^* \cdot \{1\} \cdot \{0, 1\}^n$ and strings can be accepted in only one way.

Automaton M_n has $n+2$ states. An equivalent deterministic automaton must remember the last $n+1$ inputs so as to be able to give the right answer for the current input and the next n inputs. Hence the deterministic machine needs 2^{n+1} states. \square

The example in the above proof has been known since at least the early 1970's; here we add the observation that the M_n are in fact unambiguous.

The third exponential gap between deterministic and nondeterministic descriptors is implied by either of the above theorems and has been known for a long time. It is given in [MF, Prop. 1] where its early history is discussed. This exponential difference is also the maximum possible because it is achieved as an upper bound by the standard subset construction. Thus there is no class of regular sets which simultaneously satisfy the exponential gap between deterministic and unambiguous finite automata and the exponential gap between unambiguous and arbitrary nondeterministic finite automata. The possibility of simultaneous nonpolynomial gaps remains open.

Finally, we present one corollary of Theorem 7.1 for context-free grammars.

DEFINITION 7.3. Let $G = (N, \Sigma, P, S)$ be a context-free grammar defined as in [AU]. Then the *size* of G , denoted by $|G|$, equals

$$\sum_{A \rightarrow w \in P} (|w| + 1).$$

COROLLARY 7.4. *There exists $c > 0$ such that for infinitely many context-free grammars G and for any structurally equivalent structurally unambiguous context-free grammar H ,*

$$|H| > 2^{c \cdot |G|}.$$

Proof. The corollary follows from Theorem 7.1, by noting that:

- (1) two regular grammars are structurally equivalent if and only if they are equivalent [HRS];
- (2) a regular grammar is structurally unambiguous if and only if it is unambiguous [T]; and
- (3) there is an algorithm for converting a nondeterministic finite automaton M into an equivalent regular grammar G such that $|G|$ is $O(|M|)$ (see [AU]). \square

REFERENCES

- [AHU] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
- [AU] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation, and Compiling*, Vol. 1, Prentice-Hall, Englewood Cliffs, NJ, 1972.
- [B] J. A. BRZOWSKI, *Canonical regular expressions and minimal state graphs for definite events*, in Proc. Symposium on Mathematical Theory of Automata, Polytechnic Press of the Polytechnic Institute of Brooklyn, Brooklyn, NY, 1963, pp. 529-562.
- [CI] T. CHAN AND O. H. IBARRA, *On the finite-valuedness problem for sequential machines*, Theoret. Comput. Sci., 23 (1983), pp. 95-101.
- [GI] E. M. GURARI AND O. H. IBARRA, *A note on finite-values transducers*, Math. Systems Theory, 16 (1983), pp. 61-66.
- [HU] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.
- [H] H. B. HUNT, III, *Observations on the complexity of regular expression problems*, J. Comput. System Sci., 19 (1979), pp. 222-236.

- [HRS] H. B. HUNT, III, D. J. ROSENKRANTZ AND T. G. SZYMANSKI, *On the equivalence, containment, and covering problems for the regular and context-free languages*, J. Comput. System Sci., 12 (1976), pp. 222-268.
- [Mc] R. MCNAUGHTON, *The loop complexity of regular events*, Inform. Sci., 1, 1969, pp. 305-328.
- [McP] R. MCNAUGHTON AND S. PAPERT, *Counter-Free Automata*, MIT Press, Cambridge, MA, 1971.
- [MF] A. R. MEYER AND M. J. FISCHER, *Economy of description by automata, grammars, and formal systems*, in Conference Record, Twelfth Annual IEEE Symposium on Switching and Automata Theory, East Lansing, MI, October 1971, pp. 144-152.
- [N] A. NOZAKI, *Equivalence problems of non-deterministic finite automata*, J. Comput. System Sci., pp. 18 (1979), 8-17.
- [PU] M. C. PAULL AND S. H. UNGER, *Structural equivalence of context-free grammars*, J. Comput. System Sci., 2 (1968), pp. 427-463.
- [RH] D. J. ROSENKRANTZ AND H. B. HUNT, III, *Efficient algorithms for structural similarity of grammars*, in Proc. 7th ACM Symposium on Principles of Programming Languages, Las Vegas, NV, January 1980, pp. 213-219.
- [SM] L. J. STOCKMEYER AND A. R. MEYER, *Word problems requiring exponential time: preliminary report*, in Proc. 5th Annual ACM Symposium on Theory of Computing, Austin, TX, May 1973, pp. 1-9.
- [T] J. W. THATCHER, *Tree automata: an informal survey*, in Currents in the Theory of Computing, A. V. Aho, Ed., Prentice-Hall, Englewood Cliffs, NJ, 1973, pp. 143-172.
- [Z] Y. ZACKSTEIN, *On star-free events*, in Conference Record, the Eleventh Annual IEEE Symposium on Switching and Automata Theory, 1970, pp. 76-80.

AN $O(n \log^2 n)$ ALGORITHM FOR MAXIMUM FLOW IN UNDIRECTED PLANAR NETWORKS*

REFAEL HASSIN† AND DONALD B. JOHNSON‡

Abstract. A new algorithm is given to find a maximum flow in an undirected planar flow network in $O(n \log^2 n)$ time, which is faster than the best method previously known by a factor of $\sqrt{n}/\log n$. The algorithm constructs a transformation of the dual of the given flow network in which differences between shortest distances are equal, under suitable edge correspondences, to edge flows in the given network. The transformation depends on the value of a maximum flow. The algorithm then solves the shortest distances problem efficiently by exploiting certain structural properties of the transformed dual, as well as using a set of cuts constructible in $O(n \log^2 n)$ time by a known method which is also used to find the requisite flow value. The main result can be further improved by a factor of $\log n/\log^* n$ if a recently developed shortest path algorithm for planar networks is used in place of Dijkstra's algorithm in each step where shortest paths are computed.

Key words. flow, maximum flow, planar, network, duality, graph algorithm

1. Introduction. The best algorithms known for solving the maximum flow problem in capacitated networks with n vertices and m edges run in $O(\min \{n^{5/3} m^{2/3}, nm \log n\})$ computational steps [4], [12], [13]. On planar networks this bound reduces to $O(n^2 \log n)$, a bound known earlier for undirected networks [6], since $m = O(n)$.

The best bound for general planar networks is $O(n^{3/2} \log n)$. This bound is achieved by a divide-and-conquer algorithm, due to Johnson and Venkatesan [8], that operates on recursively subdivided regions of a planar representation of the given network. More efficient algorithms exist for (s, t) -planar networks, those that can be drawn in the plane with the source and the sink on a common face. These algorithms run in $O(n \log n)$ steps. One, due to Itai and Shiloach [6], derives from the "uppermost path" method of Ford and Fulkerson [3]. The other, due to Hassin [5], makes use of the properties of shortest paths in a planar dual network.

The algorithm of Itai and Shiloach [6] for flows in (general) undirected planar networks consists of two phases, each of which runs in $O(n^2 \log n)$ time. In the first phase a minimum (s, t) -cut is found; in the second a flow with value equal to the capacity of this cut is constructed. Reif [11] has shown how to find a minimum (s, t) -cut in an undirected planar network in $O(n \log^2 n)$ time. In this paper we show how to extend the ideas of [5] so that a shortest path computation in a derived network that depends on a given feasible flow value yields a flow function of this value for a general undirected planar network. We then show how to solve this shortest path problem quickly using a set of cuts which can be constructed by a modification of Reif's algorithm. Given a flow value and these cuts, our algorithm runs in $O(n \log n)$ time, thus giving a combined algorithm which solves the maximum flow problem in undirected planar networks in $O(n \log^2 n)$ time.

Our bounds cited above are derived using Dijkstra's shortest path algorithm (see [1], [7]) as a subroutine. If the algorithms of Frederickson [2] are used, these bounds can be improved as will be discussed later. It is interesting to observe that, in the case of finding flows in (s, t) -planar networks, the algorithm of reference [5] is amenable

* Received by the editors December 7, 1982, and in final revised form May 9, 1984.

† Department of Statistics, Tel Aviv University, Tel Aviv 69978, Israel.

‡ Department of Computer Science, Pennsylvania State University, University Park, Pennsylvania 16802.

The work of this author was partially supported by the National Science Foundation under grant MCS 80-02684.

to improvement using Frederickson’s algorithm, but the “uppermost path” algorithm is not since sorting can be reduced to the uppermost path computation [6].

2. Definitions and assumptions.

A flow network N is a quadruple (G, s, t, c) where

- (i) $G = (V, E)$ is an undirected graph with n vertices and m edges and, throughout this paper, is assumed to be given with a fixed planar embedding,
- (ii) s and t are distinct vertices, the source and sink, respectively, and
- (iii) $c: E \rightarrow \mathbb{R}^+$ is a capacity function assigning a positive real to each edge.

We denote an (undirected) edge with endvertices v and w as $(v - w)$. A flow is a function $f: V \times V \rightarrow \mathbb{R}^+ \cup \{0\}$ satisfying $f(v, w) = 0$ whenever $(v - w) \notin E$, $0 \leq f(v, w) + f(w, v) \leq c(e)$ for every edge $e = (v - w) \in E$, and $\sum_{(v-w) \in E} [f(v, w) - f(w, v)] = 0$ for every vertex $v \in V - \{s, t\}$. The value of a flow f is defined by $v(f) = \sum_{(v-t) \in E} [f(v, t) - f(t, v)]$. A flow f is a maximum flow if $v(f) \geq v(f')$ for every other flow f' . We denote the value of a maximum flow by v_{\max} where the network referred to is understood.

A cut $C \subseteq E$ is a minimal set of edges that disconnects t from s . The capacity of a cut is the sum of the capacities of its edges. A classical result is that the value of a maximum flow is equal to the minimum over the capacities of all cuts [3].

Without loss of generality we assume that G is triconnected. (If G were not, it could be triangulated in linear time.) Therefore G has a unique dual $G^d = (V^d, E^d)$ which is a graph without loops and multiple edges. (The uniqueness is by virtue of the fixed embedding.) Let F and F^d denote the set of faces of G and G^d , respectively.

The following one-to-one correspondences exist: $V \leftrightarrow F^d$, $F \leftrightarrow V^d$, and $E \leftrightarrow E^d$. For corresponding edges $(v' - w') \in E$ and $(v - w) \in E^d$, v' is taken to correspond to v and w' to w when w' follows v' in the clockwise direction in the face corresponding to v . For each $e \in E^d$ we define its length $l(e)$ to be equal to the capacity $c(e')$ of the corresponding primal edge $e' \in E$. These definitions give us a distance network $N^d = (G^d, l)$ corresponding to the given capacitated network N . The procedure for dualizing planar graphs, including the correspondence between the endpoints of the edges, is described, for instance, in [9].

3. Finding a minimum (s, t) -cut. We start with a brief description of Itai and Shiloach’s algorithm.

Let ϕ^s and ϕ^t denote the faces in N^d which correspond to s and t , respectively. Without loss of generality we assume that ϕ^s is the exterior face of N^d . A minimum cut in N corresponds therefore to a cycle of minimum length enclosing ϕ^t in N^d .

Let $\Pi = (\xi^s = \xi_1, \dots, \xi_k = \xi^t)$ be a shortest (ξ^s, ξ^t) -path in N^d where ξ^s is a dual vertex on ϕ^s and ξ^t is a dual vertex on ϕ^t , both chosen so as to minimize the length of Π over all shortest paths between such pairs. Call an edge $(\xi - \xi_i)$ Π -left if $\xi \notin \Pi$ and when traversing Π from ξ^s to ξ^t it is incident with ξ_i on the left. Define Π -right edges similarly. These definitions are extended to the edges incident with ξ^s and ξ^t by viewing Π as extended at each of its two ends by an edge to a new vertex situated properly within ϕ^s and ϕ^t , respectively. Figure 1 illustrates these concepts. Since N^d is triconnected and has a fixed embedding, every edge incident with Π is either Π -left or Π -right but not both.

Since Π is a shortest (ξ^s, ξ^t) -path, there exists a cycle of minimum length enclosing ϕ^t which intersects Π exactly once, and uses exactly one Π -right and one Π -left edge. The algorithm of Itai and Shiloach finds such a cycle and the corresponding minimum cut in the primal network as follows.

ALGORITHM MIN-CUT (N) [6]

{ N is a flow network with dual N^d }

{MIN-CUT (N) is a minimum (s, t) -cut of N }

for $i = 1, \dots, k$ do

 Direct every Π -left edge $(\xi_i - \xi)$ in N^d from ξ_i to ξ and every Π -right edge $(\xi_i - \xi)$ from ξ to ξ_i .

endfor

for $i = 1, \dots, k$ do

 Let C_i^d in N^d be a *minimum ξ_i -cycle*, a shortest cycle that uses exactly one Π -left and one Π -right edge, and its Π -left edge is incident with ξ_i . {It is easy to see that such a cycle encloses ϕ^i }

endfor

return (the minimum (s, t) -cut corresponding to C_j^d for which $l(C_j^d) = \min \{l(C_i^d) | i = 1, \dots, k\}$)

end MIN-CUT

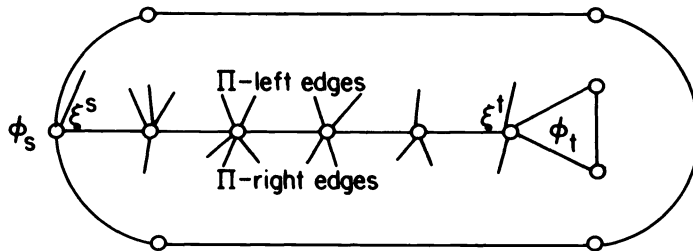


FIG. 1. A partial rendition of the dual N^d of some given network N showing a shortest (ξ^s, ξ^t) -path Π and Π -left and Π -right edges.

Given the directing of the edges in the first step, each cycle C_i^d in the second step can be found by splitting vertex ξ_i into ξ_i' , with the Π -left edges, and ξ_i'' with the other edges incident with ξ_i , and then finding a shortest (ξ_i', ξ_i'') -path. When Dijkstra's algorithm (see [1], [7]) is used, the time required for finding each of these paths is bounded by $O(n \log n)$. Thus, since $k \leq n$, it follows that MIN-CUT(\cdot) terminates in $O(n^2 \log n)$ time.

Dijkstra's algorithm has been improved upon recently by Frederickson [2] in the case where the shortest path problem to be solved is on a planar network. As with Dijkstra's algorithm, Frederickson's results apply when all edge lengths are nonnegative. He shows that a single shortest path computation can be done in $O(n\sqrt{\log n})$ time. With $O(n \log n)$ preprocessing, each of any number of single source computations can be done in $O(n \log^* n)$ time. Thus Itai and Shiloach's minimum-cut algorithm can be implemented to run in $O(n^2 \log^* n)$ time. Similar improvements are obtainable in Reif's algorithm (discussed below) and in ours.

For simplicity of presentation we shall assume the use of Dijkstra's algorithm in each of the results in what follows and then, where appropriate, we shall indicate how Frederickson's algorithms can be employed. (As is well known, when computing with edge lengths from some restricted domains, the running time of Dijkstra's algorithm can also be improved. We omit discussion of what can be done in such special cases except to note that each of our bounds can be improved when finding shortest paths from a single source to all other vertices is $o(n \log n)$.)

Reif [11] describes a more efficient implementation of Itai and Shiloach's minimum-cut algorithm. He observes that if C^d is a minimum ξ_i -cycle then, for $j = 1, \dots, i-1$,

there exist minimum ξ_j -cycles that enclose C^d (that is, have no vertices strictly within C^d) and, for $j = i + 1, \dots, k$, there exist minimum ξ_j -cycles which are enclosed by C^d . This observation allows the following divide-and-conquer algorithm, which we state for our purposes so that it generates a representation for each of the cuts corresponding to a minimum ξ_i -cycle for each $i = 1, \dots, k$.

ALGORITHM CUTS (N_j) [11]

```

{ $N_j$  is an undirected planar flow network with  $n_j$  vertices}
{CUTS ( $N_j$ ) is the set of cuts in  $N_j$  that correspond to the cycles in a set of
  minimum  $\xi_i$ -cycles in  $N_j^d$ , one for each  $i = 1, \dots, k_j$ }
if  $k_j = 1$  then return ({HIMID ( $N_j$ )})
else if  $k_j = 2$  then return ({LOMID ( $N_j$ )}  $\cup$  {HIMID ( $N_j$ )})
  else return ({HIMID ( $N_j$ )}  $\cup$  CUTS ( $N_s(N_j)$ )  $\cup$  CUTS ( $N_t(N_j)$ ))
end CUTS
    
```

Here

(i) HIMID (N_j) returns in $O(n_j \log n_j)$ time the cut corresponding to a minimum ξ_{mid} -cycle of N_j where $\Pi_j = (\xi^s = \xi_1, \dots, \xi_k = \xi^t)$ and $mid = \lceil k_j/2 \rceil$. (An algorithm to do this is obtained from Algorithm MIN-CUT by replacing “for $i = 1, \dots, k$ ” with “for $i = \lceil k/2 \rceil$ ”.) LOMID (N_j) is similarly defined with $mid = \lfloor k_j/2 \rfloor$. (The introduction of the two “MIDS” corrects a minor error in the original presentation where, in fact, termination is not assured.)

(ii) The networks $N_s(N_j)$ and $N_t(N_j)$ are obtained from $N_j - \text{HIMID}(N_j)$ by adding a second source vertex s_t and a second sink vertex t_s and then, for each edge $(v - w) \in \text{HIMID}(N_j)$ where v is connected by some path to s in $N_j - \text{HIMID}(N_j)$, adding $(v - t_s)$ and an edge $(s_t - w)$, replacing multiple edges with a single “super” edge of capacity equal to the sum of the capacities of the replaced edges. The resulting network has two connected components; the one containing s is the (s, t_s) -flow network $N_s(N_j)$ and the other, containing t , is the (s_t, t) -flow network $N_t(N_j)$.

Let a network N_j which is an argument to CUTS (\cdot) be at level l if it is generated as a result of l prior calls to CUTS (\cdot) applied to N , that is, if it is generated at level l in the recursion. Thus the given network N is at level 0 and no network is at a level greater than $\lceil \log(k - 1) \rceil$, since no (ξ^s, ξ^t) -path at level l has more than $|V(\Pi_{l-1})| + 1$ vertices, where $|V(\Pi_{l-1})|$ is the number of vertices in the longest such path at level $l - 1$, if the simple expedient is employed of inheriting, rather than recomputing, Π for each of $N_s(N_j)$ and $N_t(N_j)$ from Π for N_j .

From the construction it is evident that σ_l , the total number of vertices of networks at level l , must satisfy

$$\sigma_l \leq \sigma_0 + \sum_{q=1}^l 2^q < n + 2 \cdot 2^{\log n} = 3n.$$

Thus, since both LOMID (N_j) and HIMID (N_j) run in $O(n_j \log n_j)$ time, where a network N_j has n_j vertices, the running time of Algorithm CUTS (\cdot) on a given network N with n vertices, is

$$O\left(\sum_{l=0}^{\lceil \log(k-1) \rceil} \sigma_l \log n\right) = O(n \log n \log k) = O(n \log^2 n).$$

Then, to find a minimum cut of the given network N takes time equal to $O(|\text{CUTS}(N)|)$ which is surely $O(n \log n \log k)$ by the timing analysis above. The reader may find a more thorough exposition of a more complicated proof in the original reference [11].

As indicated above, the result can be improved to $O(\min\{n \log n + n \log^* n \log k, n\sqrt{\log n k}\})$ when Frederickson's shortest path algorithms are used.

In the original reference, the set CUTS (N) is not constructed. Instead, only a minimum cut is found. With respect to our generalization, it must be noticed that in general some cuts in CUTS (N) are described in terms of "super" edges that represent sets of edges in some network nearer the root in the execution tree, and not explicitly in terms of the original edges of N . However, we may keep in CUTS (N) the information necessary to expand any cut to a description in terms of the edges of N .

It in fact is possible to obtain a representation for CUTS (N), which is in terms of the original edges of N and is $O(n)$ in size, at no asymptotically significant increase in running time and from which one minimum cut corresponding to ξ_i -cycle C_i^d can be recovered for each i in the order $i = 1, \dots, k$, where a shortest (ξ^s, ξ^t) -path in the dual of the given network is $\Pi = (\xi^s = \xi_1, \dots, \xi_k = \xi^t)$. The first step is to record the execution tree of CUTS (\cdot) applied to N , assigning to the root the cut HIMID (N), to each of the two tree edges from the root the changes that need to be made to HIMID (N) to produce HIMID ($N_s(N)$) and HIMID ($N_i(N)$), respectively, etc., recording no cuts themselves at tree vertices other than the root. The edges to some leaves will need changes for both LOMID and HIMID.

This information can be recorded in terms of the "super" edges during the execution of CUTS (\cdot) applied to N . Then the second step is an inorder traversal of the labeled execution tree to produce the changes, in Π order, in terms of the original edges of N . This step can be done within the same running time as CUTS (\cdot) since "super" edges need be expanded at most twice, once when they enter some cut and once when they leave it or a later cut. If the edges of the cut C_1 , corresponding to the minimum ξ_1 -cycle, are placed initially in a search tree, ordered lexicographically on the edges taken as ordered pairs of vertices, the cuts can be constructed (though not output) in Π order in $O(n \log n)$ time by using the changes to modify the search tree containing the edges of one cut to obtain the next.

4. Finding a maximum (s, t) flow. An algorithm for constructing a flow of value D , if one exists, in a general (directed or undirected) planar network is described in [6]. This algorithm runs in $O(n^2 \log n)$ time and can be used to construct a maximum flow whenever the flow's value v_{\max} is known.

In this section we describe an alternative algorithm which can be applied to undirected networks if v_{\max} is known.

We first define a transformation of distance network N^d as follows:

- (i) For each vertex $\xi_i \in \Pi$, two vertices ξ_i' and ξ_i'' are created.
- (ii) Each edge $(\xi_i - \xi_{i+1})$ on Π is replaced by two edges $(\xi_i' - \xi_{i+1}')$ and $(\xi_i'' - \xi_{i+1}'')$ each with length equal to $l(\xi_i - \xi_{i+1})$.
- (iii) Edges directed from ξ_i'' to ξ_i' with length $-v_{\max}$ are added for $i = 1, \dots, k$. (These are the only directed edges.)
- (iv) Every Π -left edge $(\xi_i - \xi)$ is replaced by $(\xi_i' - \xi)$ with length equal to $l(\xi_i - \xi)$. Every Π -right edge $(\xi_i - \xi)$ is replaced by $(\xi_i'' - \xi)$ with length equal to $l(\xi_i - \xi)$.
- (v) Every vertex $\xi_i \in \Pi$ is now isolated and may be removed.

The transformation is illustrated in Fig. 2. We denote the transformed graph by $G^t = (V^t, E^t)$ and the transformed distance network by N^t .

Let $1 \leq r \leq k$ be an index for which it was found that the length of the minimum ξ_r -cycle was v_{\max} . For every vertex $v \in V^t$ let $u(v)$ be the length of a shortest (ξ_r', v) -path in N^t . Since the length of every minimum ξ_r -cycle in N^d was found to be at least v_{\max} , N^t has no negative cycles and $u(v)$ is defined for every v .

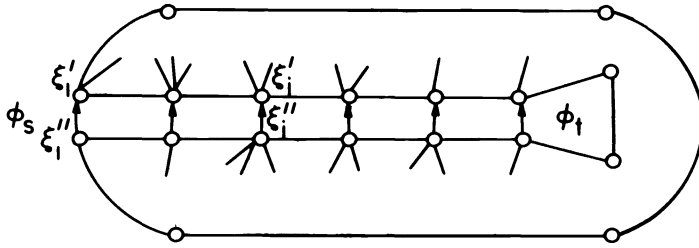


FIG. 2. A partial rendition of the transformed dual N^t of some given network N showing how the vertices ξ_i are split into ξ'_i and ξ''_i and new directed edges are introduced.

LEMMA 1. $u(\xi'_i) = u(\xi''_i) - v_{\max}$ for $i = 1, \dots, k$.

Proof. By construction, $u(\xi'_i) = 0$ and $u(\xi''_i) = v_{\max}$. Since Π is a shortest (ξ^s, ξ^t) -path there are only two possibilities concerning a shortest (ξ'_r, ξ'_i) -path:

- (i) It reaches ξ'_i from ξ''_i . In this case the lemma holds immediately.
- (ii) It terminates in a sequence of vertices $(\xi'_j | j \in J)$. If ξ'_p is the first vertex in this sequence, then $u(\xi'_p) = u(\xi''_p) - v_{\max}$, since either $p = r$ or a shortest (ξ'_r, ξ'_p) -path reaches ξ'_p from ξ''_p . This establishes that $u(\xi''_i) \leq u(\xi'_i) + v_{\max}$.

On the other hand, ξ'_i can be reached from ξ''_i along edge $(\xi''_i - \xi'_i)$ and, since $l(\xi''_i - \xi'_i) = -v_{\max}$, it follows that $u(\xi'_i) \leq u(\xi''_i) - v_{\max}$. The lemma in this case follows from the last two inequalities. \square

For every edge $(v - e) \in E^t$, let us define $u^t_{vw} = -u^t_{wv} = u(w) - u(v)$. By Lemma 1, $u(\xi'_i) - u(\xi'_j) = u(\xi''_i) - u(\xi''_j)$, so that u^d_{vw} is well defined for each edge $(v - w) \in E^d$ as follows.

- (i) For every Π -left edge $(\xi_a - w) \in E^d$, $u^d_{\xi_a w} = u^t_{\xi_a w}$.
- (ii) For every Π -right edge $(v - \xi_a) \in E^d$, $u^d_{v \xi_a} = u^t_{v \xi_a}$.
- (iii) For every Π -edge $(\xi_a - \xi_b) \in E^d$, $u^d_{\xi_a \xi_b} = u^t_{\xi_a \xi_b} = u^t_{\xi''_a \xi''_b}$.
- (iv) For every other edge $(v - w) \in E^d$, $u^d_{vw} = u^t_{vw} = u(w) - u(v)$.

THEOREM 1. A maximum flow f can be constructed as follows.

For each edge $(v - w) \in E^d$ and associated edge $(v' - w') \in E$ where v corresponds with v' and w corresponds with w' , let

$$f(v', w') = \max \{0, u_{vw}\} \text{ and } f(w', v') = \max \{0, u_{wv}\}.$$

Proof. The following observations show that f is a flow with value v_{\max} and thus a maximum flow.

- (i) $0 \leq \max \{0, u_{vw}\} \leq l(v, w) = c(v', w')$.
- (ii) For every dual face $\phi \in F^d - \{\phi^s, \phi^t\}$, with dual vertices $v_1, \dots, v_q, v_{q+1} = v_1$ in clockwise order and primal vertex $a \in V - \{s, t\}$ associated with ϕ ,

$$\sum_{i=1}^q u_{v_i v_{i+1}} = \sum_{(a-b) \in E} (f(b, a) - f(a, b)) = 0.$$

See Fig. 3.

- (iii) Let $\xi''_k = v_1, \dots, v_q = \xi''_k$ be the vertices belonging to ϕ^t in N^t in clockwise order (see Fig. 4). By Lemma 1,

$$u(v_1) - u(v_q) = u(\xi'_k) - u(\xi''_k) = -v_{\max},$$

so that

$$\sum_{i=1}^{q-1} u^t_{v_i v_{i+1}} - v_{\max} = 0.$$

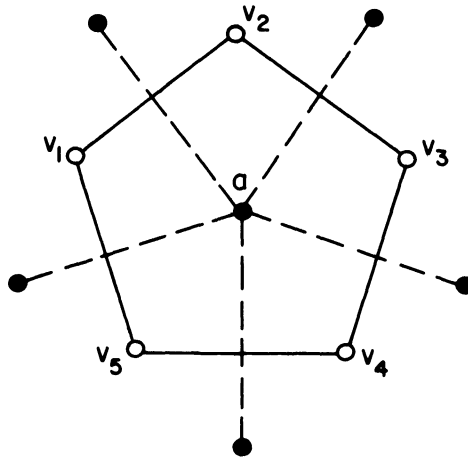


FIG. 3. Dual face $\phi = (v_1, v_2, v_3, v_4, v_5)$ corresponding to primal vertex a , for the case $\phi \in F^d - \{\phi^s, \phi^t\}$, where $q = 5$. Primal edges are shown dashed. As described in § 2, primal and dual edges that cross correspond.

This implies

$$\sum_{(t-v') \in E} (f(v', t) - f(t, v')) = v_{\max}. \quad \square$$

Once the labeling function $u(V^t)$ is obtained, the above construction can easily be seen to yield a flow function in $O(n)$ time. Thus, it remains to show how to compute $u(V^t)$ efficiently.

We note before proceeding to this discussion that, when $k = O(1)$, the bound in § 3 for finding CUTS(N) reduces to $O(n \log n)$. In the case when the network is (s, t) -planar (i.e. $k = 1$), the dual faces ϕ^s and ϕ^t have a common dual vertex which can be chosen as $\xi^s = \xi^t$ so that only one cycle need be found. In this case the algorithm is essentially the one described in [5] and the time needed to find both a minimum cut and a maximum flow is $O(n \log n)$ when Dijkstra's algorithm is used and $O(n\sqrt{\log n})$ when Frederickson's algorithm is used.

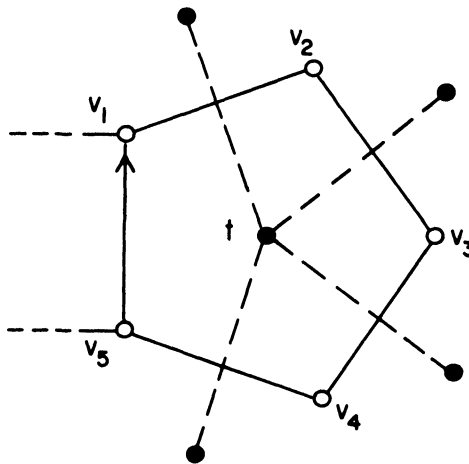


FIG. 4. Dual face $\phi = (v_1, v_2, v_3, v_4, v_5)$ corresponding to primal vertex t . Notice that there is no primal edge corresponding to dual edge $(v_5 - v_1)$, since this directed edge was introduced in the transformation of N^d .

5. Computing distances in the transformed dual network. As above, let a minimum cut cycle of N^d be a ξ_r -cycle for some r , $1 \leq r \leq k$. Shortest (ξ_r, v) -paths in N^t can be computed for every $v \in V^t$ in $O(n^{3/2})$ time [10] and thus a maximum flow can be obtained, as described in the previous section, within this bound. It is not known how to solve the shortest path problem faster in general in planar networks when $\Omega(n)$ of the edges have negative lengths. However, our network N^t has a structure which we exploit to obtain the required shortest distances in $O(n \log n)$ time, using repeated applications of Dijkstra's algorithm, when given a suitable representation of the minimum ξ_i -cycles in N^t that correspond to the (noncrossing) cuts in the set CUTS (N).

We denote the minimum ξ_i -cycles that we obtain in § 3 as C_i^t where, for each $i = 1, \dots, k$, cycle C_i^t corresponds with the primal cut C_i . We now define the sets Δ_i^+ and Δ_i^- for $i = 1, \dots, k-1$ by the relation

$$V(C_{i+1}^t) = (V(C_i^t) - \Delta_i^-) \cup \Delta_i^+,$$

where

- (i) $V(C_i^t)$ is the vertex set of C_i^t for $i = 1, \dots, k$, and
- (ii) the intersection $\Delta_i^+ \cap \Delta_{i+1}^-$ contains only those vertices in $C_i^t \cap C_{i+1}^t$ connected by an edge to some vertex in $C_i^t - C_{i+1}^t$. See Fig. 5. We note that the inverse relation also holds,

$$V(C_i^t) = (V(C_{i+1}^t) - \Delta_i^+) \cup \Delta_i^-.$$

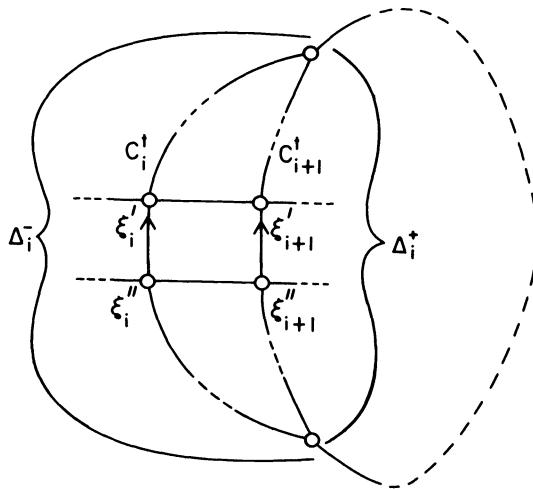


FIG. 5. Example of C_i^t and C_{i+1}^t in the general case where there may be vertices in common. The set Δ_i^- is comprised of all vertices of $C_i^t - C_{i+1}^t$ plus the first and last vertices on $C_i^t \cap C_{i+1}^t$. The set Δ_i^+ is comprised of all vertices of $C_{i+1}^t - C_i^t$ plus the first and last vertices on $C_i^t \cap C_{i+1}^t$.

The sets Δ_i^+ and Δ_i^- for $i = 1, \dots, k-1$ can be generated in order $i = 1, \dots, k-1$ in $O(n)$ time by a traversal of the Reif execution tree as described in § 3. This fact would be immediate if Δ_i^+ and Δ_i^- were defined so that $\Delta_i^+ \cap \Delta_i^- = \emptyset$. However, even though there is overlap, the bound of $O(n)$ can be seen to hold by observing that the sum over all vertices of the number of times a vertex can repeat within either all the Δ^+ sets or all the Δ^- sets is bounded by the number of edges in N^t , which is $O(n)$. Not only can these sets be used, as described in § 3, to recover the minimum length

cycles C_i^t , they are also of essential use in reducing the complexity of the computation of the labels $u(v)$ for $v \in V^t$ to $O(n \log n)$.

The cycles C_i^t for $i=1, \dots, k$ divide N^t into $k+1$ subnetworks N_0^t, \dots, N_k^t where, for $i=1, \dots, k-1$, N_i^t is the subnetwork bounded by and including C_i^t and C_{i+1}^t , N_0^t is everything outside and including C_1^t , and N_k^t is everything within and including C_k^t . See Fig. 6. Since the intersection of two adjacent subnetworks is a shortest cycle we obtain the following result.

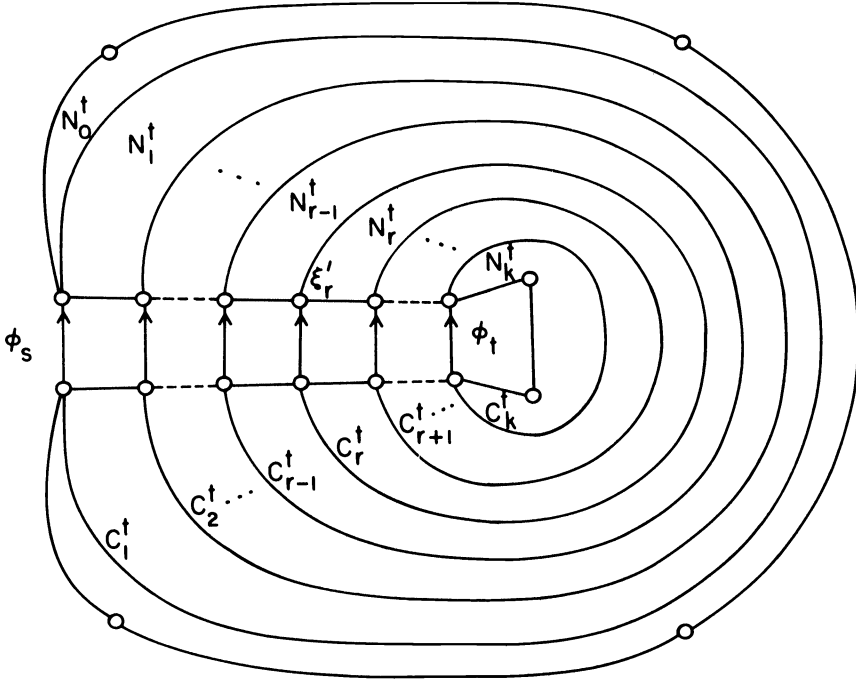


FIG. 6. Example of subnetworks of N^t induced by the cycles C_i^t , $i=1, \dots, k$. It may be that adjacent cycles have subpaths in common as shown in Fig. 5.

LEMMA 2. Let v be a vertex in N_i^t . Then if $i < r$ there exists a shortest (ξ'_r, v) -path in N^t which is contained in $\cup_{j=0, r-1} N_j^t$. Similarly, if $i \geq r$ there exists a shortest (ξ'_r, v) -path in N^t which is contained in $\cup_{j=r, k} N_j^t$.

This lemma implies that the computation of $u(v)$ for $v \in V^t$ can be restricted to the subnetwork $\cup_{j=0, r-1} N_j^t$ for v in this subnetwork, and similarly for v in $\cup_{j=r, k} N_j^t$. We confine our detailed discussion to the latter case. The former case is treated similarly.

Let $P(v)$ be a shortest (ξ'_r, v) -path. An example is given in Fig. 7, where P_v is shown as a concatenation of subpaths P_1, P_2, P_3, P_4 , and P_5 . In general, for any vertex $v \in V^t$ where v is in N_i^t let a normal path be a simple (ξ'_r, v) -path $P(v) = (P_r, \dots, P_q, \dots, P_{2q-i})$ such that, for $j=r, \dots, q$, subpath P_j is in N_j^t , and, for $j=q+1, \dots, 2q-i$, subpath P_j is in N_{2q-j}^t and uses no edges of negative length. We require also that q be minimal subject to these conditions. Call q the index of reversal of $P(v)$. As the following lemma states, for every v there exists a shortest (ξ'_r, v) -path that is normal.

LEMMA 3. For any $i=r, \dots, k$, for every vertex v in N_i^t there exists a normal shortest path $P(v)$.

Proof. (It may be helpful to consider the example in Figure 7.) Assume that a shortest path P touches some shortest cycle C^t corresponding to some member of CUTS(N). If the last shortest cycle it touched was also C^t , then there is a subpath of the cycle that can be used to replace the subpath of P whose endpoints are on the cycle. Then, once P departs from some cycle C_j^t into N_{j-1}^t , P cannot use a negative edge because, to do so, it would cross itself and the embedded cycle thus created (which could not have negative length) could be removed. \square

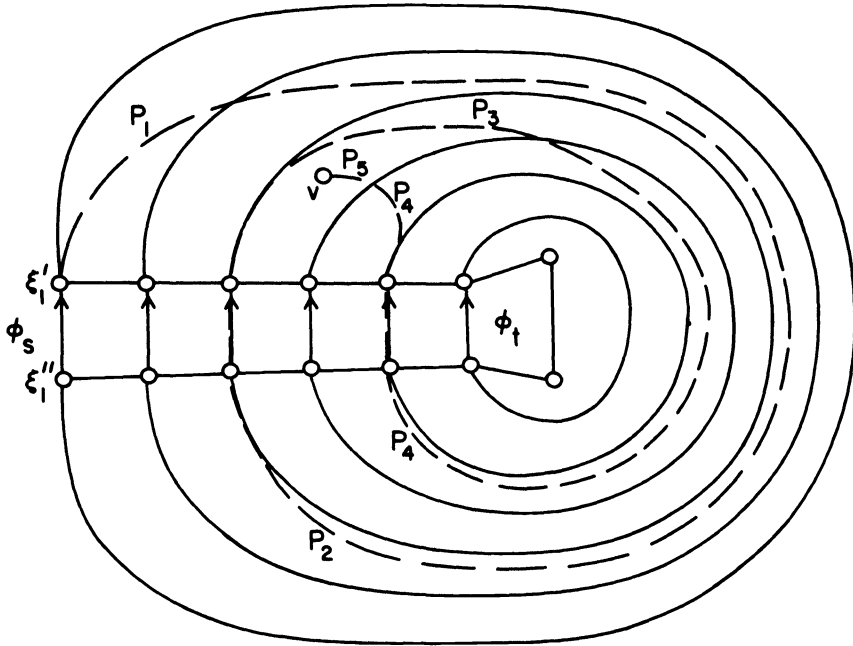


FIG. 7. Example of shortest (ξ_r', v) -path $P(v) = (P_1, P_2, P_3, P_4, P_5)$. In this example $r = 1$, the index of reversal $q = 4$, and v is in N_3^t .

We now give an algorithm to compute $u(v)$ for all v in $\cup_{i=r,k} N_i^t$.

ALGORITHM INSIDE-LABELS (N^t)

```

{Given the transformed dual  $N^t$ , INSIDE-LABELS ( $\cdot$ ) produces the labels
  $u(v)$  for all the vertices  $v$  in  $\cup_{i=r,k} N_i^t$ }
{Initialize the labels}
for  $v$  in  $\cup_{i=r,k} N_i^t$  do  $u(v) \leftarrow \infty$  endfor
 $u(\xi_r') \leftarrow 0$ 
{Compute shortest paths in  $C_r^t$ }
SP( $C_r^t, \{\xi_r'\}$ )
{Compute shortest paths in the forward direction}
for  $i \leftarrow r$  until  $k$  do
    {Compute shortest paths in  $N_i^t$  from start vertices in  $\Delta_i^-$ }
    SP( $N_i^t, \Delta_i^-$ )
endfor
{Extend shortest paths in the backward direction}
SP( $(\cup_{i=r,k} N_i^t), \Delta_{k-1}^+$ )
end INSIDE-LABELS
    
```

In Algorithm INSIDE-LABELS (\cdot), SP (X, W) is a two-step algorithm. The first step is Dijkstra's shortest path algorithm applied to network X from which all edges of negative length have been removed and with whatever u -labels its vertices have, starting the candidate set with the vertices in W . This is equivalent to running the usual version of the algorithm from w after deleting the edges of negative length and augmenting X with a new vertex w and edges $(w - v)$ of length $u(v)$ for each $v \in W$. The second step is to treat the edges of negative length individually as follows. For edge $(\xi'' - \xi')$ execute the assignment

$$u(\xi') = \min \{u(\xi'), u(\xi'') - v_{\max}\}.$$

LEMMA 4. Algorithm INSIDE-LABELS (\cdot) computes $u(v)$ correctly for all $v \in \cup_{i=r,k} N_i^t$. Whenever a vertex $v \in V(C_i^t) - \Delta_i^-$ is expanded in a shortest path computation on N_i^t in the forward loop, no labels change.

Proof. Observe that Dijkstra's algorithm is applied always to subnetworks without edges of negative length. The effect of the edges of negative length is obtained explicitly following each application of Dijkstra's algorithm.

Give C_r^t the name N_{r-1}^t . First, it can be seen that all labels $u(v)$ for $v \in N_{r-1}^t$ are computed correctly by the first three lines of the algorithm. Then, let V_h be the set of all vertices v for which there is a normal shortest (ξ_r^t, v) -path $P(v) = (P_r, \dots, P_{h-1})$.

Consider first such paths for which $h - 1$ is the index of reversal (that is, there is no reversal). Assume that $u(v)$ has been correctly computed by INSIDE-LABELS (\cdot) for all $v \in V_h$ when $i = h \leq k$ before the beginning of some iteration of the loop that starts with "for $i \leftarrow r$ until k do". Consider a vertex $w \in N_h^t$ for which there is a normal shortest (ξ_r^t, w) -path $P(w) = (P_r, \dots, P_h)$. If $w \in C_h^t$, then $u(w)$ is already correct by assumption. Otherwise, $w \notin C_h^t$, and the last vertex x on $P(w)$ that is in N_{h-1}^t and therefore with correct label $u(x)$, is in Δ_h^- . It follows that the iteration with $i = h$ must compute $u(w)$ correctly and that no label is changed by an expansion of a vertex in $V(C_i^t) - \Delta_i^-$. By induction, then, it is shown that $u(v)$ is computed correctly for all $v \in V_h, r \leq h \leq k$.

A simpler argument is applicable to the segment (P_q, \dots, P_b) , of any normal shortest path $P(v) = (P_r, \dots, P_q, \dots, P_b)$ with index of reversal $q < b$, given as we have just proved that $u(w)$ is correct for all w for which there exists a normal shortest path $P(w) = (P_r, \dots, P_q)$ upon completion of the loop that starts with "for $i \leftarrow r$ until k do". These segments contain no edges of negative length. Thus a single application of Dijkstra's algorithm suffices and, in fact, the second step in SP can be omitted. The desired results follow from Lemma 2. \square

When INSIDE-LABELS (\cdot) is combined with a similar procedure to calculate labels on $\cup_{i=1,r-1} N_i^t$ we have a correct procedure for the entire problem. Call this combined procedure LABELS (\cdot).

From earlier discussions it follows that the running time of LABELS (\cdot) is $O(s \log n)$, where $s < n^t + \sum_{i=1,k} |C_i^t|$ for $n^t = |V(N^t)| = O(n)$. There exist networks N for which $\sum_{i=1,k} |C_i^t| = \Omega(n^2)$, so LABELS (\cdot) as stated above has a running time of $O(n^2 \log n)$.

To improve this bound we give the following refinements in which shortest paths are computed on the subnetworks $N_i^t - (V(C_i^t) - \Delta_i^-)$, as Lemma 4 permits, and these subnetworks are produced explicitly by removing and replacing edges in N^t . In INSIDE-LABELS (\cdot) replace

{Compute shortest paths in C_r^t }
 SP ($C_r^t, \{\xi_r^t\}$)

with

```
{Isolate  $C_r^t$  in  $N^t$ }
 $N^t \leftarrow N^t - \{(v-w) | v \in C_r^t, w \notin C_r^t\}$ 
{Compute shortest paths in  $C_r^t$ }
 $SP(N^t, \{\xi_r^t\})$ 
{Disconnect  $C_r^t$ }
 $N^t \leftarrow N^t - \{(v-w) | v, w \in C_r^t\}$ ,
```

and replace

```
for  $i \leftarrow r$  until  $k$  do
  {Compute shortest paths in  $N_i^t$  from start vertices in  $\Delta_i^-$ }
   $SP(N_i^t, \Delta_i^-)$ 
endfor
```

with

```
for  $i \leftarrow r$  until  $k$  do
  {At each iteration,  $C_i^t$  is disconnected}
  {Put into  $N^t$  the edges that enter the proper interior of  $N_i^t$  from  $C_i^t$ , and delete
  the edges that are incident on some vertex properly within the region bounded
  by  $C_{i+1}^t$ . Thus, since no edges come into  $N_i^t$  from the two regions surrounding
  it,  $SP(\cdot)$  will compute on  $N_i^t$ }
   $N^t \leftarrow (N^t \cup \{(v-w) | v \in \Delta_i^-, w \notin C_i^t \text{ and within } C_i^t\})$ 
   $\quad - \{(v-w) | v \in \Delta_i^+, w \notin C_{i+1}^t \text{ and within } C_{i+1}^t\}$ 
  {Compute shortest paths in  $N_i^t$  from start vertices in  $\Delta_i^-$ }
   $SP(N^t, \Delta_i^-)$ 
  {Maintain  $C_{i+1}^t$  disconnected}
   $N^t \leftarrow N^t - \{(v-w) | v, w \in \Delta_i^+\}$ 
endfor.
```

Now, the sum over all forward shortest path computations of the number of vertices processed in each computation is $O(n)$ since $\sum_{i=r,k} |\Delta_i^-|$ is $O(n)$, so these computations are of complexity $O(n \log n)$. The manipulations of N^t can also be implemented to run in $O(n \log n)$ time over all. This gives us our theorem.

THEOREM 2. *Shortest (ξ_r^t, v) -paths can be computed for every $v \in V^t$ in $O(n \log n)$ time, given the set CUTS (N) .*

Frederickson's shortest path algorithm without preprocessing [2] can be used in each instance where Dijkstra's algorithm is used in our algorithm, giving a bound of $O(n\sqrt{\log n})$ overall when given the set CUTS (N) .

Theorems 1 and 2, together with the results of § 3, imply an algorithm for maximum flows in undirected planar networks that runs in $O(n \log^2 n)$ time and $O(n)$ space. With Frederickson's improvements, the time bound is $O(n \log n \log^* n)$ when $k = \Omega(n)$ and as small as $O(n\sqrt{\log n \log k})$ when k is small.

6. Acknowledgment. We are grateful to a referee for pointing out an error, now corrected, in the result of § 5.

REFERENCES

- [1] E. W. DIJKSTRA, *A note on two problems in connexion with graphs*, Numer. Math., 1 (1959), pp. 269-271.
- [2] G. N. FREDERICKSON, *Shortest path problems in planar graphs*, Proc. 24th Annual Symposium on Foundations of Computer Science, 1983, pp. 242-247.

- [3] L. R. FORD AND D. R. FULKERSON, *Maximal flow through a network*, *Canad. J. Math.*, 8 (1956), pp. 399-404.
- [4] Z. GALIL, *A new algorithm for the maximal flow problem*, Proc. 19TH Annual Symposium on Foundations of Computer Science, 1978, pp. 231-245.
- [5] R. HASSIN, *Maximum flow in (s, t) planar networks*, *Inform. Proc. Lett.*, 13 (1981), p. 107.
- [6] A. ITAI AND Y. SHILOACH, *Maximum flow in planar networks*, this Journal, 8 (1979), pp. 135-150.
- [7] D. B. JOHNSON, *Efficient algorithms for shortest paths in sparse networks*, *J. Assoc. Comput. Mach.*, 24 (1977), pp. 1-13.
- [8] D. B. JOHNSON AND S. M. VENKATESAN, *Using divide and conquer to find flows in directed planar networks in $O(n^{3/2} \log n)$ time*, Proc. Twentieth Annual Allerton Conference on Communication, Control, and Computing, Univ. Illinois, Urbana, October 1982, pp. 898-905.
- [9] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York, 1976.
- [10] R. J. LIPTON, D. J. ROSE, AND R. E. TARJAN, *Generalized nested dissection*, *SIAM J. Numer. Anal.*, 16 (1979), pp. 346-358.
- [11] J. H. REIF, *Minimum s - t cut of a planar undirected network in $O(n \log^2(n))$ time*, this Journal, 12 (1983), pp. 71-81.
- [12] D. D. SLEATOR, *An $O(nm \log n)$ algorithm for maximum network flow*, Ph.D. Dissertation, Comp. Sci. Dept., Stanford Univ., Stanford, CA, 1980.
- [13] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, Proc. 13th Annual ACM Symposium on Theory of Computing, 1981, pp. 114-122; *J. Comput. System Sci.*, 26 (1983), pp. 362-391.

AN $O(E \log E + I)$ EXPECTED TIME ALGORITHM FOR THE PLANAR SEGMENT INTERSECTION PROBLEM*

EUGENE W. MYERS†

Abstract. It is an open question in computational geometry as to whether there exists an $O(E \log E + I)$ algorithm to determine the I intersections of a collection of E line segments in the plane. An approach utilizing a work list bubble sort and a distribution-based search is presented. The resulting algorithm has $O(E \log E + I)$ expected time complexity. In the worst case the algorithm has the same complexity as the algorithm of Bentley and Ottmann [IEEE Trans. Comput., 28 (1979), pp. 643–647]: $O(E \log E + I \log E)$. The algorithm requires only $O(E)$ space and in contrast to prior work, no restrictions are placed upon the nature of the intersections.

Key words. concrete complexity, computational geometry, scan-line algorithm, work list bubble sort, distribution-based search

1. Introduction. A recent trend in computer science has been the study of geometric problems in terms of their algorithmic complexity [3], [12], [13], [17]. In particular, geometric intersection problems have been studied extensively [2], [5], [11], [18]. The results are applicable to many practical problems including computer graphics, automated printed circuit layout, and computer-assisted architectural design. Efficient algorithms for the intersection problem treated here provide the basis for faster hidden-line elimination in the “object-space” framework [8], [16], [19].

The problem to be examined is as follows. Given E line segments in the plane, list (count) all intersecting pairs. Shamos and Hoey [18] presented an $O(E \log E)$ algorithm to detect whether intersections exist. By extending their technique, Bentley and Ottmann [2] demonstrated an $O(E \log E + I \log E)$ time, $O(I)$ space algorithm for listing all intersecting pairs of segments where I is the number of intersections. A subsequent refinement by Brown [4] reduced the space requirement to $O(E)$. Although this appears to be an improvement over the naive $O(E^2)$ algorithm, it is not the case when I approaches E^2 . The problem of designing a definitively superior $O(E \log E + I)$ time, $O(E)$ space algorithm was first conjectured in [18] and is currently open.

An $O(E \log E + I)$ *expected time* algorithm is presented here. The necessary statistical hypothesis is that the $2E$ endpoints of the segments are uniformly distributed. The algorithm’s worst case performance is again $O(E \log E + I \log E)$. The algorithm requires only $O(E)$ space. By a direct refinement of the method of Ottmann and Bentley, A. Schmitt [15] has simultaneously designed an algorithm with the same time complexity as the one given here. However, his algorithm uses $O(I)$ space and requires the more stringent statistical hypothesis that both the segment endpoints and the I points of intersection be uniformly distributed. The algorithm here achieves the less restrictive assumption through the novel use of a work list bubble sort to detect intersections between segment endpoints.

In the next section, the problem statement is formalized and preliminary constructions and definitions are made. Section 3 illustrates the central role of a work list bubble sort and distribution-based searching in the algorithm presented in § 4. The discussion in § 5 highlights a number of subproblems that can be solved in $O(E \log E + I)$ worst-case time.

* Received by the editors July 19, 1982, and in final revised form May 3, 1984. This work was supported in part by the National Science Foundation under Grant MCS-8210096.

† Department of Computer Science, University of Arizona, Tucson, Arizona, 85721.

2. Preliminaries. The problem is to find all the intersections of a collection of E planar line segments, $SEGMENT = \{e_1, e_2, \dots, e_E\}$. Each line segment is specified by the x - and y -coordinates of its endpoints: $e = \langle \langle x_e, y_e \rangle, \langle \bar{x}_e, \bar{y}_e \rangle \rangle$. It is stipulated that in an $O(E)$ preprocess the segment endpoints have been arranged so that the start-point $\langle x_e, y_e \rangle$ is to the "left" of the final-point $\langle \bar{x}_e, \bar{y}_e \rangle$, i.e. $x_e < \bar{x}_e$ or $x_e = \bar{x}_e$ and $y_e \leq \bar{y}_e$.

As in the algorithms of Shamos and Hoey [18] and Bentley and Ottmann [2], the key conceptualization is to imagine a vertical *scan line* sweeping from left to right along the x -axis. The critical events are those moments at which this scan line reaches the abscissa of a line segment endpoint. Let $EVENT = \langle x_1, x_2, \dots, x_{EV} \rangle$ be the list obtained by sorting the set of abscissas of segment endpoints into ascending order. Observe that $EV \leq 2E$ as each segment has two endpoints.

For each event, x_i , there may be more than one segment having an endpoint at x_i . Such segments will be distinguished on the basis of whether x_i is the abscissa of their start-point, final-point, or both (i.e. vertical segments). Formally, let $BEG(i) = \langle e_1, e_2, \dots, e_{B(i)} \rangle$ be the list of segments for which $x_e = x_i \neq \bar{x}_e$; let $END(i) = \langle e_1, e_2, \dots, e_{E(i)} \rangle$ be the list of segments for which $x_e \neq x_i = \bar{x}_e$; and let $VERT(i) = \langle e_1, e_2, \dots, e_{V(i)} \rangle$ be the list of segments for which $x_e = x_i = \bar{x}_e$. It is further stipulated that the segments in each of these lists occur in descending order of their start-point ordinates y_e . Figure 1 illustrates these lists.

Algorithmically, all the lists above can be constructed with a single sort. First form an auxiliary list consisting of the following 4-tuples. For each nonvertical segment e introduce the 4-tuples $\langle e, x_e, 0, -y_e \rangle$ and $\langle e, \bar{x}_e, 2, -y_e \rangle$. For each vertical segment e introduce the 4-tuple $\langle e, x_e, 1, -y_e \rangle$. In $O(E \log E)$ time, heapsort this auxiliary list according to the lexicographical order of the second, third, and fourth components.

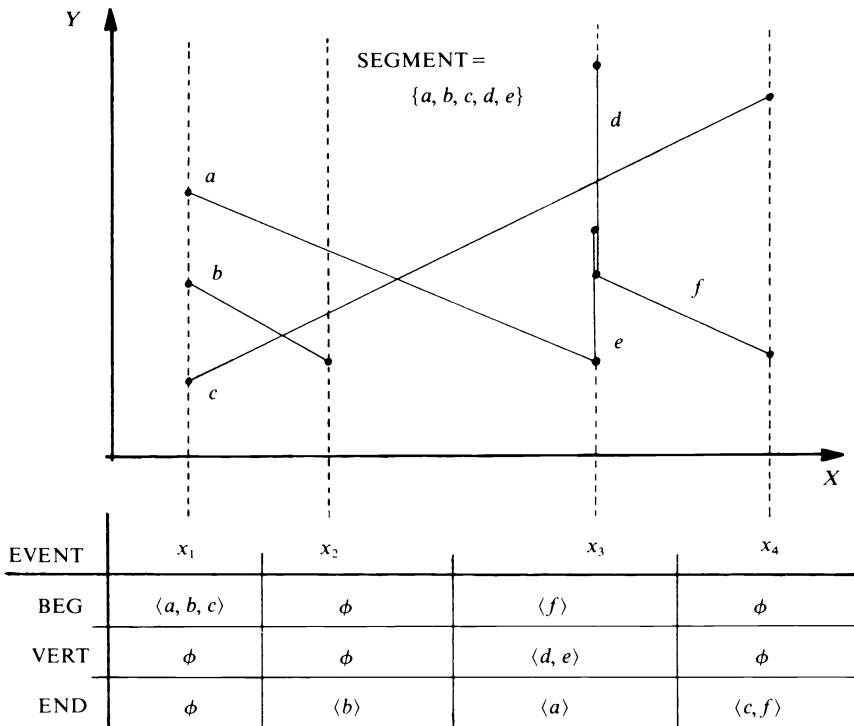


FIG. 1. Fundamental structures.

Then note that *EVENT* is the list of second components with duplicate entries removed. Further note that the list of first components is the concatenation of the lists *BEG*(1), *VERT*(1), *END*(1), *BEG*(2), ..., *END*(*EV*) in the order given. The boundaries of this partition are easily identified by examining the second and third components. Thus all the desired lists can be extracted in a final $O(E)$ sweep over the sorted auxiliary list.

The elements of *EVENT* divide the range $[x_1, x_{EV}]$ into the *event intervals* $[x_i, x_{i+1}]$ for i between 1 and *EV*-1. By construction, the abscissa of a segment endpoint cannot lie strictly within any of the event intervals. Thus if a segment has a point whose abscissa is interior to an event interval then the segment is guaranteed to span the entire interval (i.e., have a point at abscissa x for every x in the interval). The set of segments spanning the i th event interval is formally defined with the recursive definition:

$$SPAN(i) = \text{ If } i=0 \text{ Then } \emptyset \text{ Else } SPAN(i-1) \cup (BEG(i) - END(i)).$$

Note that for all i , *SPAN*(i) does not contain any vertical segments.

The algorithm presented here centers on computing the *x-order*, $<_x$, of the segments for every event x . Intuitively, $<_x$ ranks segments according to the order in which they intersect a scan line at x . For nonvertical segments let m_e be the slope of segment e and let $y_e(x)$ be the ordinate value of segment e at abscissa x . For vertical segments let $m_e = \infty$ and let $y_e(x) = y_e$. Formally, the rank of a segment on a scan line at x is the lexicographical rank of the ordinate value/slope pair $\langle y_e(x), m_e \rangle$:

$$e <_x f \text{ iff } y_e(x) < y_f(x) \text{ or } y_e(x) = y_f(x) \text{ and } m_e < m_f.$$

Figure 2 gives an example of the *x-order* of a collection of segments. Clearly one can make analogous definitions for the other *x-relations*: $=_x$, \cong_x , $>_x$, \cong_x , and \neq_x .

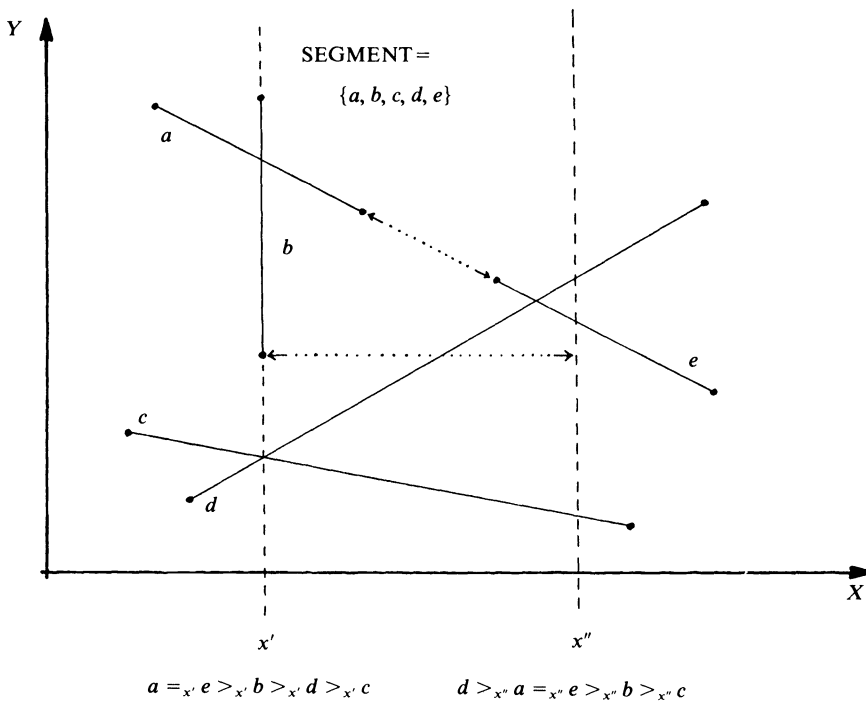


FIG. 2. *X-order example.*

In prior scan line algorithms [2], [15], [18] for the planar intersection problem, the following degenerate situations were explicitly excluded from the problem domain.

- (a) Vertical lines.
- (b) Multi-segment intersections (three or more segments intersecting at a point).
- (c) Infinite intersections (collinear segments that intersect).

In this paper the problem domain is unrestricted. The preceding definitions and all subsequent results have been designed with these anomalous cases in mind. For example, the inclusion of slope in the definition of \langle_x differs from prior work. This additional refinement insures that multi-segment intersections will be properly treated as collinear and only collinear segments are equal in any (and every) x -order.

A characterization of the conditions under which segments intersect in terms of the above constructions is now presented. Lemma 1 asserts that segments e and f intersect if there is an event interval in which e and f satisfy one of three mutually disjoint conditions. Condition 1.1 characterizes intersections involving a vertical segment. Condition 1.2 covers the case in which two nonvertical segments intersect at the start point of one (or both) of the segments. The main thrust of Lemma 1 is embodied in Condition 1.3, the *exchange predicate*, which reflects an observation first made in [18]. It asserts that any other intersection between nonvertical segments e and f is characterized by a reversal in the x -order of e and f at the endpoints of an event interval.

LEMMA 1. *Without loss of generality if e and f are not vertical then assume $\underline{x}_e \cong \underline{x}_f$, otherwise assume that e is vertical and if f is also vertical then further assume $\underline{y}_e \cong \underline{y}_f$. Segments e and f intersect if and only if there exists i such that*

$$(1.1) \quad e \in \text{VERT}(i) \text{ and } f \in \text{SPAN}(i-1) \cup \text{BEG}(i) \cup \text{VERT}(i) \text{ and } y_f(x_i) \in [\underline{y}_e, \bar{y}_e],$$

or

$$(1.2) \quad e \in \text{BEG}(i) \text{ and } f \in \text{SPAN}(i-1) \cup \text{BEG}(i) \text{ and } y_f(x_i) = \underline{y}_e,$$

or

$$(1.3) \quad e, f \in \text{SPAN}(i) \text{ and } (e <_{x_i} f \text{ and } e >_{x_{i+1}} f \text{ or } f <_{x_i} e \text{ and } f >_{x_{i+1}} e).$$

Proof. (\Rightarrow) Suppose e and f intersect at $\langle x, y \rangle$ and if they are collinear that this is the point of intersection with smallest abscissa (ordinate for vertical segments). First consider the case where e and f are not vertical and $\underline{x}_e \cong \underline{x}_f$. If $x = \underline{x}_e$ then let i be the integer for which $e \in \text{BEG}(i)$. Observe that $y_f(x_i) = y_f(x) = y_e(x) = y_e$ and that $x_i = x \in [\underline{x}_f, \bar{x}_f]$ implies $f \in \text{SPAN}(i-1) \cup \text{BEG}(i)$, i.e. Condition 1.2 holds. If $x \neq \underline{x}_e$ then let i be the integer for which $x \in (x_i, x_{i+1}]$. Observe that e and f cannot be collinear and that $x \in (\underline{x}_e, \bar{x}_e] \cap (\underline{x}_f, \bar{x}_f]$ implies $e, f \in \text{SPAN}(i)$. If $e <_{x_i} f$ then $y_e(x_i) < y_f(x_i)$. Moreover, e and f intersect at $x \in (x_i, x_{i+1}]$ implies $y_e(x_{i+1}) \cong y_f(x_{i+1})$ and $m_e > m_f$ implies $e >_{x_{i+1}} f$. Similarly $e >_{x_i} f$ implies $e <_{x_{i+1}} f$. Thus Condition 1.3 holds.

Now suppose that e is vertical. Let i be the integer for which $e \in \text{VERT}(i)$ and note that $x = x_i$ and $y \in [\underline{y}_e, \bar{y}_e]$. If f is not vertical then $y_f(x_i) = y$ and $x \in [\underline{x}_f, \bar{x}_f]$ implies $f \in \text{SPAN}(i-1) \cup \text{BEG}(i)$, i.e. Condition 1.1 holds. If f is also vertical then e and f must be collinear, $f \in \text{VERT}(i)$, and $\underline{y}_f \cong \underline{y}_e$ implies $y_f(x_i) = \underline{y}_f = y$ as y is the smallest ordinate of an intersection point. Thus Condition 1.1 holds.

(\Leftarrow) Conditions 1.1 and 1.2 imply e and f intersect at $\langle x_i, y_f(x_i) \rangle$. Now suppose Condition 1.3 is true. If $y_e(x_i) = y_f(x_i)$ or $y_e(x_{i+1}) = y_f(x_{i+1})$ then immediately e and f intersect. Otherwise $e <_{x_i} f$ and $e >_{x_{i+1}} f$ implies $y_e(x_i) < y_f(x_i)$ and $y_e(x_{i+1}) > y_f(x_{i+1})$. That is, e is below f at x_i , above f at x_{i+1} , and both segments span the interval. They must intersect in the interval $[x_i, x_{i+1}]$. Similarly e and f intersect when $f <_{x_i} e$ and $f >_{x_{i+1}} e$. \square

3. The key methods—work list bubble sort and distribution-based search. Consider the following simplification of the intersection problem. Suppose that $\text{EVENT} = \langle x_1, x_2 \rangle$ and that $\text{BEG}(1) = \text{END}(2) = \text{SEGMENT}$. There is only one event interval and no

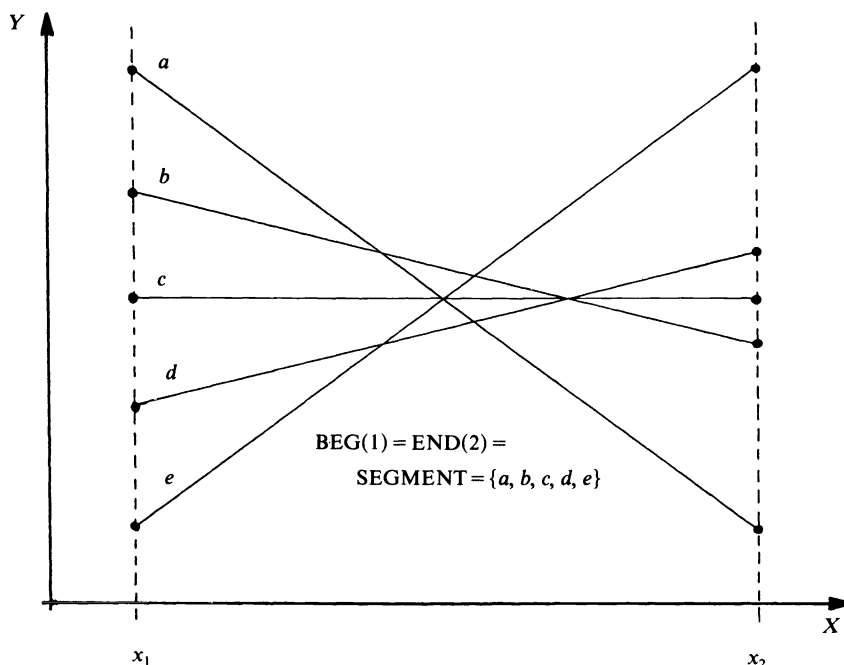


FIG. 3. A single interval intersection problem.

vertical segments. An instance of a “Single Interval Intersection Problem” is depicted in Fig. 3. By Lemma 1 it suffices to find the intersections satisfying Conditions 1.2 and 1.3 for the interval $[x_1, x_2]$. An algorithm is sketched that detects these intersections in $O(E \log E + I)$ time and $O(E)$ space.

As a first step, sort the segments into x_1 -order in $O(E \log E)$ time. Observe that segments with equal ordinate values, $y_e(x_1)$, occur in contiguous sub-lists of this ordering. Every pair of segments from such a sub-list and only these pairs satisfy Condition 1.2 of Lemma 1. The sub-lists can be detected in $O(E)$ time and the set of segment pairs from each sub-list can then be listed in $O(I)$ time where I is the number of intersections.

It remains to find all the intersections satisfying the exchange predicate (1.3). Consider the application of a bubble sort to the x_1 -ordered segments to obtain the x_2 -ordering of the segments. The key observation is that the set of segment pairs exchanged in performing the sort is exactly the set of segment pairs satisfying the exchange predicate.

The naive version of bubble sort requires $O(E^2)$ time. What is needed is an $O(E + I)$ time algorithm where I is the number of intersections or equivalently the number of exchanged pairs. This can be accomplished with the following *work list* variant of bubble sort. Let the current order of segments be the x_1 -order computed in the first step. Initialize the work list to be the set of all segments e for which e and its successor are *not* in x_2 -order. While the work list is not empty, perform the following steps.

- (3A) Pop a segment e from the work list. Let f be its successor in the current order.
- (3B) Remove f and the predecessor of e from the work list if present.
- (3C) Exchange e and f in the current order and report that they intersect.
- (3D) Push the current predecessor of f onto the work list if it and f are not in x_2 -order. Push e onto the work list if it and its current successor are not in x_2 -order.

The validity of the algorithm follows from the fact that at the start of each iteration the work list consists of those segments for which it and its current successor are not in x_2 -order. The algorithm terminates as each iteration strictly reduces the “exchange distance” to the x_2 -ordered arrangement of segments.

Steps 3A through 3D can be performed in constant time with the following structures. Model both the current order and work list as doubly-linked lists in which each cell contains a pointer to the record modeling the appropriate segment. Let each segment record contain a pointer to its cell in the current order and a pointer to its cell in the work list if present, the nil pointer otherwise. In constant time, cells can be popped and pushed from the work list and an arbitrary cell within the list can be deleted. Given a cell in the work list (current order) one can reach the corresponding cell in the current order (work list) via the segment record. The links of the current order list give current successors and predecessors. Segments are exchanged in this order by interchanging just the segment record pointers of the relevant adjacent cells.

With these models the work list bubble sort requires $O(E + I)$ time and $O(E)$ space. The initial work list is formed in an $O(E)$ sweep over the x_1 -order of the segments. Steps 3A through 3D are repeated exactly I times as an intersection is reported in each iteration. Thus the sort proper takes $O(I)$ time. Only $O(E)$ space is required at any given moment as each segment occurs in the work list at most once. Note that intersections are not necessarily reported in order of increasing abscissa. For example, when applied to Fig. 3, the algorithm reports intersections out of order regardless of the initial state of the work list and its implementation as a stack of queue.

The general intersection problem can be viewed as $EV - 1$ separate single interval problems. The algorithm presented here solves these individual problems in increasing event order. One can imagine the scan-line as jumping from event to event while the work list bubble sort detects the intersections between jumps. The $O(E \log E)$ cost of determining the initial x -order for each interval problem readily distributes across the computation: the bubble sort for the i th interval delivers the initial order for the $(i + 1)$ st interval. Distributing the $O(E)$ cost of computing the initial work list for each interval problem is more difficult. If during the processing of some interval, a segment comes to have a successor in the current x -order with which it satisfies the exchange predicate then the segment must immediately be placed in the work list of the interval in which the exchange predicate is satisfied. This requires a search for the interval containing the abscissa of the point of intersection. An $O(\log E)$ bisecting search on the ordered $EVENT$ list could be used. However, a distribution-based search reduces the cost to $O(1)$ expected time under the assumption that the elements of $EVENT$ are uniformly distributed in the interval $[x_1, x_{EV}]$.

Recently, much attention has been given to distribution-based sorting methods [1], [6], [7], [10], [20]. The technique sketched here is the search analogue of the sorting-by-partition method in [10]. Consider evenly dividing the interval $[x_1, x_{EV}]$ into EV subintervals (buckets) of size $\Delta = (x_{EV} - x_1) / EV$. Observe that an arbitrary x is in the bucket $[x_1 + \Delta H(x), x_1 + \Delta(H(x) + 1))$ where $H(x) = \lfloor (x - x_1) / \Delta \rfloor$. For each i in $[0, EV]$, let $MIN(i)$ be the ordinal position of the smallest event in the interval $[x_1 + \Delta i, x_{EV}]$.

$$MIN(i) = \min \{j \mid x_j \in EVENT \text{ and } H(x_j) \geq i\}.$$

For the limiting case, let $MIN(EV + 1) = MIN(EV)$. It is an elementary exercise to design an algorithm that constructs the vector MIN in an $O(E)$ sweep of the ordered $EVENT$ list. Under the hypothesis that the events in $EVENT - \{x_1, x_{EV}\}$ are uniformly distributed over the interval $[x_1, x_{EV}]$, the expected number of events in a bucket is

$(EV-2)/EV$ as there are EV buckets and the $EV-2$ events are found in a given bucket with equal probability. Thus the expected value of $MIN(i) - MIN(i-1)$ is also $(EV-2)/EV$ for all i .

Suppose segments e and f are known to intersect by the exchange predicate (1.3) in some event interval. This interval can be found as follows. First determine the abscissa of the segments' point of intersection, $XINTER(e, f)$, by analytic means. Let $\delta = H(XINTER(e, f))$. It follows from the construction of MIN that $x_{MIN(\delta)-1} < XINTER(e, f) \leq x_{MIN(\delta+1)}$. Thus the event interval, $[x_{i-1}, x_i]$, in which the segments satisfy the exchange predicate must have i in the range $[MIN(\delta), MIN(\delta+1)]$. The desired event interval can then be found by searching this limited subrange of event intervals. By the uniform distribution hypothesis the expected number of intervals in the subrange is $(MIN(\delta+1) - MIN(\delta)) + 1 < 2$. Thus by employing a bisecting search over this subrange, the expected search time is $O(1)$ and the worst case is guaranteed to be $O(\log E)$.

On a floating-point computer the calculation of real quantities is not exact due to the introduction of rounding errors. When realized on such a machine the search algorithm above is correct only if the computation of H is monotonic. That is, $x' \geq x \Rightarrow \hat{H}(x') \geq \hat{H}(x)$ where \hat{H} is the computed value of H (as opposed to its true value). If this condition is not met by the host hardware then the computed value of δ may differ from its true value. In such instances the index of the desired event interval will not be in the range $[MIN(\delta), MIN(\delta+1)]$ and the search will fail. This weakness can be removed by employing a finger-based search [9] for resolving bucket collisions. In this context the search would start at the $MIN(\delta)$ th event interval (the finger) and then search right or left for the correct interval in a bisecting fashion. The expected and worst-case times for the search remain unchanged.

The uniform distribution hypothesis was chosen for simplicity. More generally, the expected time for the search is $O(1) + E(\max(\log N_i))$ where E denotes "expected value" and N_i is the number of events in the i th bucket. From [1] it follows that the search is still $O(1)$ if the underlying density function is bounded, Riemann-integrable, and has compact support. The N -tree method of Ehrlich [7] could also be used here. It has the advantage of performing well over a wider class of density functions [20] (including those with exponentially vanishing tails). Its primary disadvantage is that it requires $O(E^2)$ space in the worst case.

4. The algorithm. The work list variation of bubble sort leads to an efficient algorithm for the single interval problem. The distribution-based search gives an $O(1)$ expected time method for determining the event interval in which a pair of segments intersect. With these methods the planar segment intersection problem can be solved efficiently. The simple data structures employed in the single interval problem must be enhanced to meet the more dynamic requirements of the general algorithm. The extension of these structures and the primitive operations that will be assumed are described in the paragraphs below.

In the general algorithm, the current x -order contains just those segments spanning the current event interval. This implies that one must be able to add and delete segments from this order efficiently. The doubly-linked list model of the current x -order is extended by superimposing a height-balanced tree (assume an AVL tree) upon it. Conversely, one may view the current x -order to be modeled as an AVL tree XOT (X -Ordered Tree) whose symmetric order is explicitly threaded with a doubly-linked list. As before, each cell in the tree points to its corresponding segment record and vice versa. The following primitives are assumed.

Add(e, i)—Add segment e to XOT under the x_i -order.

Delete(e)—Delete segment e from XOT .

Exchange(e)—Exchange e and Above(e) in the current ordering of XOT .

Above(e) (Below(e))—The segment immediately above (below) e in the current ordering of XOT if it exists, Λ otherwise.

The maintenance of the doubly-linked threads in XOT is an elementary exercise. With this model Add and Delete are $O(\log E)$ operations and Exchange, Above, and Below are $O(1)$ as before. Note that Exchange destroys any particular x -order. However, the operation Add(e, i) is only applied when XOT is x_i -ordered.

In the general algorithm there are $EV - 1$ distinct work lists, $WORK(i)$, one for each event interval $[x_i, x_{i+1}]$. The collection of work lists is modeled as an array of pointers to the first cell of a doubly-linked work list of the form described in § 3. The following primitives are assumed.

Push(e, i)—Add segment e to $WORK(i)$.

Pop(e, i)—Delete a segment from $WORK(i)$ and return it in e .

Remove(e)—Delete segment e from the work list containing it (if any).

In the previous section it was shown that the primitives in this repertoire can all be done in $O(1)$ time.

The final primitive is assumed to implement the distribution-based search algorithm sketched in § 3.

Hash(e, f)—A function returning the index of the interval in which e and f satisfy the exchange predicate (1.3).

The primitive Hash is only invoked with segments e and f that are known to satisfy the exchange predicate. It follows from the treatment at the end of § 3 that Hash requires $O(1)$ time in the expected case and $O(\log E)$ time in the worst case.

The complete algorithm is presented below as Algorithm 1. This algorithm begins by initializing XOT and every work list to be empty. It then proceeds by processing the event intervals in left to right order. This iteration constitutes the major loop of Algorithm 1 and at the start of its i th iteration it is claimed that

(4.1) XOT contains the segments in $SPAN(i-1)$ in x_i -order.

In each iteration, all the intersections in the current interval are detected and reported. This task is performed in four steps.

(4A) The segments in $BEG(i)$ are added to XOT . Intersections satisfying Condition 1.2 of Lemma 1 are reported. After this step XOT contains the segments in $SPAN(i-1) \cup BEG(i)$ in x_i -order.

(4B) The segments in $VERT(i)$ are checked for intersections satisfying Condition 1.1 of Lemma 1. This step has no net effect on XOT .

(4C) The segments in $END(i)$ are deleted from XOT . After this step XOT contains the segments in $SPAN(i)$ in x_i -order.

(4D) The intersections satisfying the event exchange predicate for the current interval are detected as XOT is bubble sorted into x_{i+1} -order. Note that after this step XOT satisfies Assertion 4.1 for the next iteration.

In Algorithm 1, each of Steps 4A through 4D appear as minor loops. It is claimed that before an iteration of any minor loop, the collection of work lists satisfies the condition:

(4.2) $e \in WORK(j)$ iff
 e and Above(e) satisfy the event exchange predicate in the j th event interval and $j \geq i$ where i is the index of the current interval.

The invariance of this minor loop predicate is maintained throughout the algorithm by pushing and removing work lists elements to correctly reflect the effect of every Add, Delete, and Exchange operation on *XOT*. The critical feature of Assertion 4.2 is that it implies $WORK(i) = \{e \in SPAN(i) \mid e \text{ and } Above(e) \text{ are not in } x_{i+1}\text{-order}\}$ just before Step 4D is about to be performed for the *i*th event interval. Thus the work list bubble sort performed in Step 4D is correct as $WORK(i)$ is correctly initialized at its outset.

The specification of Algorithm 1 contains several repeated code fragments that have been collected into *macro* definitions. References to these macros are underlined; their definition follows the algorithm. Keep in mind that macro parameters are passed by substitution.

ALGORITHM 1. The planar segment intersection algorithm

```

/* Initialize work lists and AVL tree */
For i ← 1 to EV Do
    WORK(i) ← ∅
    XOT ← ∅

/* For each event xi in increasing order do */
For i ← 1 to EV Do
    (4A) /* Add segments in BEG(i) and list their start point intersections */
        For e ∈ BEG(i) in order Do
            Add(e, i)
            f ← Below(e)
            If f ≠ ∆ Then
                Remove(f)
                Enter(f)
                Enter(e)
                Report(e, yg(xi) = ye)

    (4B) /* Find all intersections with segments in VERT(i) */
        For e ∈ VERT(i) in order Do
            Add(e, i)
            Report(e, yg(xi) ∈ [ye, ȳe])
        For e ∈ VERT(i) in order Do
            Delete(e)

    (4C) /* Delete segments in END(i) */
        For e ∈ END(i) in order Do
            f ← Below(e)
            Delete(e)
            Remove(e)
            If f ≠ ∆ Then
                Remove(f)
                Enter(f)

    (4D) /* Find all "event exchange" intersections in [xi, xi+1] */
        While WORK(i) ≠ ∅ Do
            Pop(e, i)
            "e and Above(e) intersect"
            f ← Below(e)
    
```

```

Exchange( $e$ )
Remove(Below( $e$ ))
If  $f \neq \Lambda$  Then
    Remove( $f$ )
    Enter( $f$ )
Enter( $e$ )

Macro Report( $e$ , cond)
 $g \leftarrow$  Below( $e$ )
While  $g \neq \Lambda$  and cond Do
    “ $e$  and  $g$  intersect”
     $g \leftarrow$  Below( $g$ )
 $g \leftarrow$  Above( $e$ )
While  $g \neq \Lambda$  and cond Do
    “ $e$  and  $g$  intersect”
     $g \leftarrow$  Above( $g$ )

Macro Enter( $e$ )
 $g \leftarrow$  Above( $e$ )
If  $g \neq \Lambda$  and  $e >_{\min(\bar{x}_e, \bar{x}_g)}$   $g$  Then
    Push( $e$ , Hash( $e$ ,  $g$ ))

```

Algorithm 1 is correct regardless of the class of intersections present. The correctness and generality of the algorithm follow directly from Lemma 1 and the discussion above. One subtle point: the macro *Report* in Step 4A (4B) correctly reports those segments that intersect with e by Condition 1.2 (1.1) as the segments are contiguous in *XOT*'s order and the intersecting pairs are reported only once as e is being entered into *XOT* for the first and only time. In Theorem 1, the algorithm is shown to have the time and space performance claimed at the outset of the paper.

THEOREM 1. *Algorithm 1 requires $O(E \log E + I)$ expected time when *EVENT* is uniformly distributed. Algorithm 1 requires $O(E \log E + I \log E)$ time in the worst case. Algorithm 1 requires $O(E)$ space.*

Proof. The initialization of *XOT* and the work lists at the outset of the algorithm require $O(E)$ time. As was shown at the time of their introduction, the construction of all the other auxiliary structures requires $O(E \log E)$ time.

The body of Step 4A is repeated once for each segment in a *BEG*-list. Thus it is repeated at most E times. The macro *Report* contains a **while** loop for which an intersection is reported in each iteration. The body of the while loop takes $O(1)$ time. The other primitives in Step 4A take either $O(1)$ or $O(\log E)$ time. Thus the total time taken by Step 4A is $O(E \log E + I_1)$ time in the worst case where I_1 is the number of intersection reported in the step. Similarly, the total times taken by Steps 4B and 4C are $O(E \log E + I_2)$ and $O(E \log E)$ respectively.

The body of Step 4D is repeated once for each intersecting pair satisfying the exchange predicate. All operators are $O(1)$ with the exception of Hash which is $O(1)$ in the expected case and $O(\log E)$ in the worst case. Thus the total cost of performing Step 4D is $O(I_4)$ in the expected case and $O(I_4 \log E)$ in the worst case. The total number of intersections, I , is the sum of I_1 , I_2 , and I_4 . Thus the overall performance of Algorithm 1 is $O(E \log E + I)$ in the expected case and $O(E \log E + I \log E)$ in the worst case.

With the exception of the work lists, it is clear that all the data structures involved require $O(E)$ space. Assertion 4.2 implies that there are at most $E - 1$ segments in all of the work lists at any time. This follows as there are at most $E - 1$ adjacent segments in any ordering and a given segment pair satisfies the exchange predicate in at most one event interval. Thus Algorithm 1 required only $O(E)$ space. \square

5. Discussion. The use of the work list bubble sort implies that Algorithm 1 does not report intersections in increasing order of abscissa. However, this is true only within each event interval; the intervals themselves are processed in increasing order. Informally, one may say that the intersections are sorted “with respect to the event intervals”. Thus the intersections can be totally ordered by simply sorting each collection of intersections reported in each execution of Step 4D. Although this requires $O(I \log E)$ time in the worst case, in practice it should result in some additional efficiency. Moreover, if appropriate a distribution-based method [6], [7], [10] could be used to solve each sorting subproblem in a total of $O(I)$ expected time.

If all segments are either vertical or horizontal, then Algorithm 1 performs in $O(E \log E + I)$ worst-case time. Simply observe that no pair of segments satisfies the exchange predicate. Consequently, no segment will ever be entered into a work list; the body of Step 4D will never be executed; and Hash will never be invoked. But the $O(\log E)$ worst-case performance of Hash is solely responsible for the $I \log E$ term in Algorithm 1’s performance. This result was first shown in Bentley and Ottmann’s paper [2] but was posed as a distinct algorithm. In this paper it is simply a direct consequence of the *general* algorithm.

A slight variation of Algorithm 1 gives an $O(E \log E + I)$ worst-case algorithm for yet another restricted intersection problem. Suppose that all segments are constrained to have their left endpoints at x_1 . Formally, assume $BEG(1) \cup VERT(1) = SEGMENT$; the *END* lists are unrestricted. To solve the “single start intersection problem” modify Algorithm 1 as follows. Replace all references to $WORK(i)$ with references to a *single* work list and modify all Pop and Push primitives to operate exclusively on this one work list. The second parameter of these primitives becomes superfluous and consequently the one and only call to Hash is removed. Since the primitive Hash is no longer employed, this modified algorithm must run in $O(E \log E + I)$ time in the worst case. Observe that while *XOT* is no longer reasonably ordered in later iterations of the major loop, the algorithm is correct as Add is not invoked after the first iteration.

In problem instances where the intersection density $\alpha = I/E^2$ is high, the following situation will frequently arise in the course of an event interval bubble sort. An intersecting pair of segments momentarily become adjacent in *XOT* and are entered into some work list only to be removed when another exchange separates them. The effort expended by Hash to find the appropriate work list was wasted. Such redundant searches can be eliminated by introducing a temporary work list, $WORK_T$, which is empty at the beginning of each bubble sort. During a sort, newly adjacent segments that intersect in an event interval other than the current one are placed in $WORK_T$. Only when the given sort is complete are the segments that remain in $WORK_T$ transferred via Hash to their respective work lists. This variation is not asymptotically superior to Algorithm 1 but does significantly reduce the number of searches for high density problems.

6. Conclusion. An $O(E)$ space, $O(E \log E + I)$ expected time algorithm for the planar segment intersection problem has been presented. The key techniques are the use of a work list bubble sort for solving individual event interval problems and the use of a distribution-based search to seed the work lists for these intervals. The expected time result still leaves open the question of whether or not a comparison based method must take $O(E \log E + I \log E)$ worst-case time. However, several restricted problems were observed to have $O(E \log E + I)$ worst-case algorithms.

The problem was treated in full generality. Vertical segments, multi-segment and infinite intersections were all permitted. As Sutherland et al. [19] have observed, these singularities must be carefully treated in order for the algorithm to be useful in graphics applications.

It was noted earlier that a sorted intersection list could be produced in $O(E \log E + I)$ expected time under the assumption that the abscissas of the intersection points are uniformly distributed in each event interval. Such a list readily provides the basis for a hidden-line computation. The method of Sechrest and Greenberg [16] suggests that with the use of coherence all computations can be done in $O(I)$ time except for the embeddings of locally minimum points.

The algorithm described in this paper performs a one-time analysis on a set of planar line segments. In many contexts it would be useful to incrementally obtain a solution. For example if several new edges are added to the problem or if the locations of some of the existing segments are perturbed, the new solution could be computed by simply detecting how it differs from the previous solution. This can obviously be done in $O(E)$ time per input modification by a direct extension of the naive $O(E^2)$ algorithm. The problem of arranging a more efficient incremental algorithm appears very difficult. If I_e is the number of intersection involving segment e , are $O(I_e)$, $O(I_e + \log E)$, or even $O(I_e \log E)$ incremental methods possible? Rosen [14] has noted in the context of data flow analysis that highly efficient one-time algorithms do not necessarily lead to efficient incremental algorithms.

Acknowledgments. The author would like to thank the referees for their careful and detailed reviews which lead to a greatly improved presentation. The author would also like to thank Peter J. Downey, Webb Miller, and Tim A. Budd for their many helpful suggestions.

REFERENCES

- [1] S. G. AKL AND H. MEIJER, *On the average-case complexity of "bucketing" algorithms*, J. Algorithms, 3 (1982), pp. 9-13.
- [2] J. L. BENTLEY AND T. A. OTTMANN, *Algorithms for reporting and counting geometric intersections*, IEEE Trans. Computers, 28 (1979), pp. 643-647.
- [3] J. L. BENTLEY AND M. I. SHAMOS, *Divide-and-conquer in multidimensional space*, Proc. 8th ACM Symposium on Theory of Computing, 1976, pp. 220-230.
- [4] K. Q. BROWN, *Comments on "algorithms for reporting and counting geometric intersections"*, IEEE Trans. Computers, 30 (1981), pp. 147-148.
- [5] D. P. DOBKIN AND B. CHAZELLE, *Detection is easier than computation*, Proc. 12th ACM Symposium on Theory of Computing, 1980, pp. 146-153.
- [6] W. DOBOSIEWICZ, *Sorting by distributive partitioning*, Inform. Proc. Lett., 7 (1978), pp. 1-6.
- [7] G. EHRLICH, *Searching and sorting real numbers*, J. Algorithms, 2 (1981), pp. 1-12.
- [8] W. R. FRANKLIN, *A linear exact hidden surface algorithm*, ACM Computer Graphics, 14 (1980), pp. 117-123.
- [9] L. J. GUIBAS, E. M. MCCREIGHT, M. F. PLASS AND J. R. ROBERTS, *A new representation for linear lists*, Proc. 9th ACM Symposium on Theory of Computing, 1977, pp. 49-60.
- [10] H. MEIJER AND S. G. AKL, *The design and analysis of a new hybrid sorting algorithm*, Inform. Proc. Lett., 10 (1980), pp. 213-218.
- [11] M. H. OVERMARS, *General methods for 'all elements' and 'all pairs' problems*, Inform. Proc. Lett., 12 (1981), pp. 99-102.
- [12] M. H. OVERMARS AND J. VAN LEEUWAN, *Dynamically maintaining configurations in the plane*, Proc. 12th ACM Symposium on theory of Computing, 1980, pp. 135-145.
- [13] F. P. PREPARATA, *A new approach to planar point location*, this Journal, 10 (1981), pp. 473-492.
- [14] B. K. ROSEN, *Linear cost is sometimes quadratic*, Proc. 8th ACM Conference on Principles of Programming Languages, 1981, pp. 117-124.

- [15] A. SCHMITT, *Reporting intersections of line segments: An improvement of the Ottman-Bentley algorithm*, Proc. 8th Conference on Graph Theoretic Concepts in Computer Science, H. J. Schneider and H. Gotter, Eds., Carl Hanser Verlag, Munchen, 1982, pp. 257-266.
- [16] S. SECHREST AND D. P. GREENBERG, *A visible polygon reconstruction algorithm*, ACM Trans. Graphics, 1 (1982), pp. 25-42.
- [17] M. I. SHAMOS, *Geometric complexity*, Proc. 7th ACM Symposium on Theory of Computing, 1975, pp. 224-233.
- [18] M. I. SHAMOS AND D. HOEY, *Geometric intersection problems*, Proc. 17th ACM Symposium on Foundations of Computer Science, 1976, pp. 208-215.
- [19] I. E. SUTHERLAND, R. F. SPROULL AND R. S. SCHUMACKER, *A characterization of ten hidden surface algorithms*, Comput. Surveys, 6 (1974), pp. 1-55.
- [20] M. TAMMINEN, *Analysis of N-trees*, Inform. Proc. Lett., 16 (1983), pp. 131-137.

SCHEDULING FLAT GRAPHS*

DANNY DOLEV† AND MANFRED WARMUTH‡

Abstract. The problem of scheduling a partially ordered set of unit length tasks on m identical processors is known to be NP-complete. There are efficient algorithms for only a few special cases of this problem. In this paper we analyze the effect of the structure of the precedence graph and the availability of the processors on the construction of optimal schedules. We prove that to find an optimal schedule it suffices to consider at each step only initial tasks which belong to the $m - 1$ highest components of the precedence graph. This result reduces the number of cases we have to check during the construction of an optimal schedule. Our method leads to polynomial algorithms if the number of processors is fixed and the precedence graph has a certain form. In particular, if the precedence graph contains only intrees and outtrees, this result leads to linear algorithms for finding an optimal schedule on two or three processors.

Key words. identical processors, profile, optimal schedule, intree and outtree

1. Introduction. The goal of deterministic scheduling is to obtain efficient algorithms under the assumption that all the information about the tasks to be scheduled is known in advance. One of the fundamental problems in deterministic scheduling is to schedule a set of unit length tasks, subjected to precedence constraints, on a system of identical processors. The precedence constraints between tasks are represented by a *precedence graph*, which is a directed acyclic graph. As in [GJ81] we allow the number of identical processors to vary with time. A *profile* is a sequence of natural numbers specifying how many processors are available at each time slot. A schedule for a given profile is a partitioning of all the tasks into a sequence of sets which does not violate the precedence graph. The i th set of the sequence is scheduled in the i th time slot (i.e. interval $[i - 1, i)$). Thus, the cardinality of the i th set cannot exceed the number of processors which are available in the i th time slot of the profile. A profile is *straight* if it has the same number of processors available at each time slot. The *breadth* of a profile is the maximum number of processors available at any time slot.

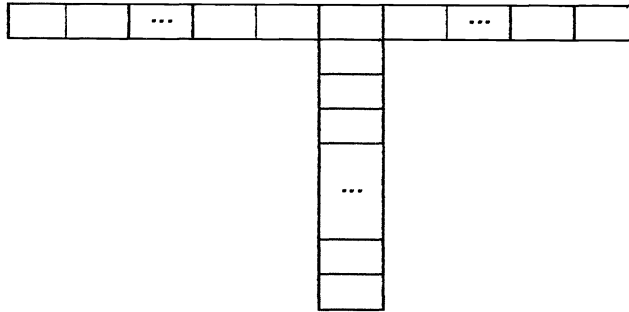
Various aspects of scheduling theory have been studied extensively in recent years [GL79] and many scheduling problems are known to be NP-complete [UI75], [GJ79], [LR78], [Wa81], [GJ81], [Ma81]. The first NP-completeness result on scheduling with precedence constraints was published by Ullman [UI75]. He showed that the existence of a schedule of a given length on a straight profile for a collection of unit length tasks subjected to some given precedence constraints is NP-complete, if the number of available processors is a variable of the problem. Notice that the breadth of the profile is not bounded by a constant. The problem remains NP-complete even for certain classes of precedence graphs [GJ81], [Ma81], [Wa81]. To support the idea that the breadth of the profile is the main source of NP-completeness we prove in [DW82b] that scheduling unit length tasks is NP-complete even if the precedence graph has height one and the profile has one processor available in each slot except for one slot that has an arbitrary number (see Table 1.1). Polynomial algorithms have been developed for only a few special cases. The first polynomial algorithm was developed

* Received by the editors December 23, 1980, and in final revised form March 23, 1984. This paper is a revision of IBM Research Report RJ3398, 1982.

† Institute of Mathematics and Computer Science, Hebrew University, Jerusalem, Israel. Part of this work was done while this author visited IBM Research, San Jose, California.

‡ Computer Science Department, University of California, Santa Cruz, California 95064. Part of this work was done while this author visited the Hebrew University, Jerusalem. The research was supported by the Fulbright Commission of West Germany, grants from Univac Corporation and Storage Technology Corporation, and by the United States-Israel Binational Science Foundation under grant 2439/82.

TABLE 1.1
 The question of existence of a schedule for a precedence graph of height one and a profile of the below form is NP-complete.



by Hu [Hu61]. It produces an optimal schedule for a straight profile of arbitrary breadth if the precedence graph is an inforest. Hu's algorithm produces a schedule according to the *Highest Level First* (HLF) strategy, meaning tasks of higher level are chosen over tasks of lower level and tasks of the same level are chosen arbitrarily. HLF also produces an optimal schedule for outforests and straight profiles of arbitrary breadth [Br81]. A restricted version of HLF provides an optimal schedule when the precedence graph is an interval order [PY79], [Ga82], or if the number of available processors is two [Ga81]. Recently, polynomial algorithms have been published [GJ81], [Wa81], [DW82c] for scheduling certain classes of precedence graphs on profiles of fixed breadth. In [Wa81], [DW82a] it was also shown that scheduling an arbitrary graph on a profile of fixed breadth is polynomial, if the height of the graph is bounded by a constant.

The major scheduling problem remaining open is whether the scheduling of an arbitrary graph of unbounded height is NP-complete or polynomial for a fixed number ($m \geq 3$) of processors.

Let m be the breadth of the profile. The median (see § 3) of the precedence graph is defined to be one plus the height of the m th highest component of the precedence graph (see Fig. 3.1) and if the precedence graph contains less than m components, then the median is zero. A task is *initial* if it does not have any predecessors. The *Elite* of the precedence graph is the set of all initial tasks that belong to components that are higher than the median.

Our main result, the Elite theorem (§ 4), states that it is enough to choose tasks from the Elite of the precedence graph for the first slot of an optimal schedule. If we do not have enough tasks in the Elite, then we choose tasks according to highest height from the set of initial tasks that are not in the Elite. After filling the first slot, the Elite theorem can be applied to the remaining precedence graph and the next slot, and so on. The theorem restricts the number of cases that need to be considered for constructing an optimal schedule. Variations of the Elite theorem are the basis for the algorithms in [Wa81], [DW82c].

In § 5, we generalize results of [Hu61], [Br81], and [GJ81] by applying the Elite theorem. We show that HLF produces an optimal schedule if the precedence graph is either an inforest or an outforest and the profile is of a certain type. In § 6 we prove some properties of graphs containing only inforest and outforest components (opposing forests). In § 7 we use these properties to develop a linear algorithm for scheduling an opposing forest on a straight profile of breadth three improving the $O(n \log n)$ time

bound of Garey et al. [GJ81].¹ Furthermore, we give an $O(n \log n)$ algorithm for scheduling an opposing forest on a profile that has two or three processors available at any time slot. The algorithm is essentially the one described in [Do80].

2. Basic definitions and properties. A precedence graph G is denoted by a tuple (V, E) , where V is the set of n tasks and E the set of edges of G . A (directed) path π of length r in G is a sequence of tasks x_0, \dots, x_r , such that the edge (x_i, x_{i+1}) , for $0 \leq i \leq r-1$, is in E . We assume that if a task x has to be executed before a task y , then there exists a (directed) path from x to y in G . Note that G is acyclic.

If there exists a path from x to y , then x is a *predecessor* of y , and y is a *successor* of x . In the case where the longest path from a task x to a task y is the edge (x, y) , we call x the *immediate predecessor* of y and y the *immediate successor* of x .

By $h(G)$ we mean the *height* of G , which is the length of the longest path in G . For a task $x \in G$ (i.e., $x \in V$) we denote by $h(x)$ the length of the longest path that starts at x . Note that a task with no successors has zero height. Tasks with identical height are said to be at the same *level*.

The graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$, denoted by $G' \subseteq G$, if $V' \subseteq V$ and for all x and y in G' , x is a predecessor of y in G' if and only if x is a predecessor of y in G . A subgraph G' of G is called a *closed subgraph* if every task in G' has the same successors in G' as it has in G . For two graphs $G = (V, E)$ and $G' = (V', E')$, $G \cup G'$ denotes the graph $(V \cup V', E \cup E')$. The graph $G = (V, E)$ is composed of $\{G_1, \dots, G_r\}$ if these subgraphs (called *components* of G) are a decomposition of G into its connected components, that is, each subgraph is a nonempty connected graph and there are no edges between tasks of different components; therefore, $G = \cup_i G_i$. A task of G is *initial* if it has no predecessors. Note that an initial task of G is not necessarily of maximum height in G . A set of k *highest initial tasks* is a subset of the set of initial tasks consisting of some k highest ones; when there are less than k initial tasks, it contains all of them. Let R be a set of initial tasks of a precedence graph G . Then $G - R$ is the subgraph of G obtained by removing the tasks of R .

We partition the time scale into time slots of length one. The time interval $[i-1, i)$ for $i \geq 1$ is the i th time slot. A *profile*, M , is a sequence of positive integers, (m_1, m_2, \dots, m_d) , specifying the number of identical processors, m_i , that are available in each time slot i , for $1 \leq i \leq d$ (see Table 2.1); d is the *length* of the profile M . The *breadth* of profile M is the maximum number of processors that is available at any time slot of M . Throughout the paper we denote the breadth of the given profile with the letter m . The profile of Table 2.1 has breadth three. We call a profile M *straight* if $m_i = m$, for all $1 \leq i \leq d$.

A *schedule* S for a precedence graph G is a sequence of sets $(S)_1 | \dots | (S)_k$ such that:

- i) the sets $(S)_i$, for $1 \leq i \leq k$, partition the tasks of G ;
- ii) if $x \in (S)_i$ and $y \in (S)_j$, for $1 \leq i \leq j \leq k$, then there is no path from y to x .

The *length* of a schedule S , denoted by $\lambda(S)$, is the index of the last nonempty set in the sequence. A minimum length schedule is called *optimal*. The schedule S *fits* the profile M if the length of S is not greater than the length of the profile and the cardinality of $(S)_i$ is not greater than m_i . The set of tasks $(S)_i$ gets executed in the i th time slot, that is $|(S)_i|$ of the m_i processors of slot i are executing the tasks of $(S)_i$ during the time interval $[i-1, i)$. Note that the length of a task equals the length of a time slot. We call the schedule S an *M-schedule* for G .

¹ In a revised version of [GJ81] Garey et al. also obtain a linear algorithm.

As an example assume we have a set of twelve tasks subject to the precedence graph G presented in Fig. 2.1. We name the tasks by numbers. Throughout the paper we always assume that the edges of the graphs are directed downwards. We look for a schedule for G that fits the profile $M = (2, 3, 3, 1, 3, 2)$. The following sequence S is a valid schedule:

$$\{1, 2\}\{3, 4, 5\}\{6, 7\}\{8\}\{9, 10, 11\}\{12\}.$$

The M -schedule S can be shown as in Table 2.1. Notice that $\lambda(S) = 6$. In Table 2.2 a schedule S' of length 5 is given for the same precedence graph, which is optimal.

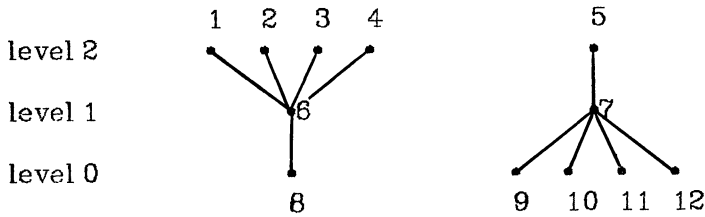


FIG. 2.1. The precedence graph G .

TABLE 2.1
The schedule S for the precedence graph G of Fig. 2.1 and the profile $M = (2, 3, 3, 1, 3, 2)$.

| slot | 1 | 2 | 3 | 4 | 5 | 6 |
|-------|---|---|---|---|----|----|
| P_1 | 1 | 3 | 6 | 8 | 9 | 12 |
| P_2 | 2 | 4 | 7 | | 10 | |
| P_3 | | 5 | | | 11 | |
| m_i | 2 | 3 | 3 | 1 | 3 | 2 |

TABLE 2.2
The schedule S' for G and M .

| slot | 1 | 2 | 3 | 4 | 5 | 6 |
|-------|---|---|----|---|----|---|
| P_1 | 1 | 2 | 4 | 6 | 8 | |
| P_2 | 5 | 3 | 9 | | 11 | |
| P_3 | | 7 | 10 | | 12 | |
| m_i | 2 | 3 | 3 | 1 | 3 | 2 |

The i th slot of a schedule S , $1 \leq i \leq \lambda(S)$, has $m_i - |(S)_i|$ idle periods. Such an idle period corresponds to a processor being idle during time slot i of S .

A schedule S is an HLF-schedule for G and M if $(S)_i$, $1 \leq i \leq \lambda(S)$, is a set of m_i highest initial tasks of the subgraph of G induced by all tasks scheduled in slot i of S or later. Note that in the above example S is a HLF-schedule, whereas S' is not. HLF-schedules have the following property. Assume task x is scheduled in slot i and y is scheduled in slot j . If $h(x) > h(y)$, then either $i \leq j$ or there is a predecessor of x in the j th slot. We say that HLF produces an optimal schedule if any HLF-schedule is

optimal; that is, if an optimal schedule can be constructed by choosing higher initial tasks before lower ones and choosing arbitrarily among initial tasks of the same height.

A schedule for G is greedy if whenever there is an idle period in some slot i then this slot contains all initial tasks of the subgraph of G induced by the tasks that appear in slot i or later. It is easy to see that any schedule can be made into a greedy one without increasing its length; thus there exists greedy schedules which are optimal.

3. The median. In this paper we study graphs that have more than m components ("flat" graphs), where m is the breadth of the profile. We use the notion of the median to characterize this property of a precedence graph.

DEFINITION. The *median* of a precedence graph G with respect to a given breadth m , denoted by $\mu(G)$, is one plus the height of the m th highest component of the precedence graph.

Thus the graph of Fig. 3.1 has median 3 with respect to $m = 3$. If the graph has fewer than m components the median is 0. For example, in the graph described by Fig. 2.1 the median with respect to $m = 3$ is 0.

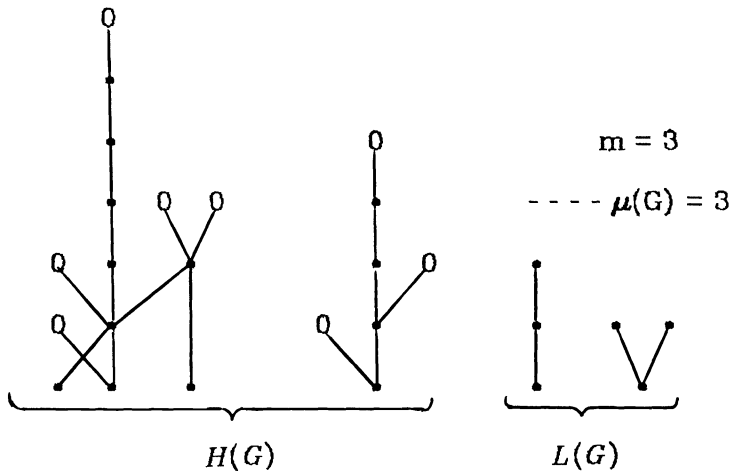


FIG. 3.1. The decomposition of a graph G into $H(G)$ and $L(G)$; 0 denotes tasks of $E(G)$.

We use the median to partition the components of G into two sets, $H(G)$ and $L(G)$ (see Fig. 3.1).

DEFINITION. The closed subgraph $H(G)$ consists of all components of height higher than the median, and $L(G)$ contains all the components that are at most as high as the median.² The set of all initial tasks of $H(G)$ is called the *Elite* of G , denoted by $E(G)$.

During the construction of a schedule, the median is a dynamic line. When a set of initial tasks is removed, the median might increase, because some components of the graph might split into several components. On the other hand, the median can drop at most by one. If it drops by one, then some initial tasks of $L(G)$ were removed. If only tasks of $H(G)$ are removed, then the median does not drop. This leads to the following properties which are used in the current paper and in [DW82c].

² All the results of this paper will still hold if we define $H(G)$ to be the closed subgraph that contains all initial tasks of height higher than the median plus all their successors, and $L(G)$ to be the remaining subgraph. See also [DW82b].

Properties of the median:

M1: There are at most $m - 1$ components of G having height at least $\mu(G)$.

M2: If $\mu(G) > 0$, then there are at least m components of G having height at least $\mu(G) - 1$.

M3: If G has at most $m - 1$ components of height at least h , then $\mu(G) \leq h$.

M4: If G has at least m components of height at least $h - 1$, then $\mu(G) \geq h$.

M5: Let R be a set of initial tasks of G . Then $\mu(G - R) \geq \mu(G) - 1$.

M6: Let R be a subset of $E(G)$. Then $\mu(G - R) \geq \mu(G)$. Furthermore, $H(G - R) \subseteq H(G) - R$ and $L(G - R) \supseteq L(G)$.

M7: Let T be a set of highest initial tasks of $L(G)$. Then $H(G - T) \subseteq H(G)$ and $L(G - T) \supseteq L(G) - T$.

M8: Let R be a subset of $E(G)$ and T be a set of highest initial tasks of $L(G)$. Then

$$H(G - (R \cup T)) \subseteq H(G) - R \text{ and } L(G - (R \cup T)) \supseteq L(G) - T.$$

The proofs of the properties M1 through M4 follow directly from the definition of median. The proof of M8 is a simple consequence of M6 and M7. To prove the remaining properties the following claim is needed.

CLAIM 3.1. *Let I be a component of G and let R be a set of initial tasks of G . Then*

$$h(I) \geq h(I - R) \geq h(I) - 1.$$

Proof. The claim trivially holds if $h(I) = 0$. Thus assume that $h(I)$ is positive. The set R contains only initial tasks of I ; therefore, the longest path in $I - R$ is by at most one shorter than the longest path of I . Thus, $h(I - R) \geq h(I) - 1$ and clearly, $h(I) \geq h(I - R)$, which completes the proof. \square

Proof of M5.

M5. Let R be a set of initial tasks of G . Then $\mu(G - R) \geq \mu(G) - 1$.

M5 is clearly true if $\mu(G) \leq 1$. Thus assume that $\mu(G) > 1$. By property M4 we only have to show that $G - R$ contains at least m components having height at least $\mu(G) - 2$. To do this observe that by property M2 the graph G contains at least m components of height at least $\mu(G) - 1$. By Claim 3.1, $h(I - R) \geq \mu(G) - 2$ for every component I of G that satisfies $h(I) \geq \mu(G) - 1$. Therefore, the subgraph $I - R$ of $G - R$ has at least one component of height at least $\mu(G) - 2$, and $G - R$ has at least m components of height at least $\mu(G) - 2$.

Proof of M6.

M6. Let R be a subset of $E(G)$. Then $\mu(G - R) \geq \mu(G)$. Furthermore, $H(G - R) \subseteq H(G) - R$ and $L(G - R) \supseteq L(G)$.

The second part of M6 is a simple consequence of the fact that $\mu(G - R) \geq \mu(G)$. Readily this inequality holds if $\mu(G) = 0$. So assume $\mu(G)$ is positive. To prove that $\mu(G - R) \geq \mu(G)$ we need to show that $G - R$ contains at least m components of height $\mu(G) - 1$ (see property M4). If I is a component with $h(I) \leq \mu(G)$, then I is in $L(G)$ and therefore $I = I - R$. Also, if $h(I) > \mu(G)$, then by Claim 3.1, $h(I - R) \geq \mu(G)$. This completes the proof, because by property M2, G contains at least m components I , satisfying $h(I) \geq \mu(G) - 1$, and for each such component I the corresponding subgraph $I - R$ of $G - R$ contains at least one component of height at least $\mu(G) - 1$.

Proof of M7.

M7: Let T be a set of highest initial tasks of $L(G)$. Then $H(G - T) \subseteq H(G)$ and $L(G - T) \supseteq L(G) - T$.

Assume that M7 does not hold for some G and T . Then $\mu(G - T) < \mu(G)$, and by M5, $\mu(G - R) = \mu(G) - 1$. For the median to drop, T must contain an initial task

of $L(G)$ of height $\mu(G) - 1$. Since T is a set of highest initial tasks of $L(G)$ it follows that T contains all tasks of $L(G)$ of height $\mu(G)$ (which are all initial). Therefore, we conclude that $L(G - T) \subseteq L(G) - T$ and thus $H(G - T) \supseteq H(G)$. But this contradicts the assumption. \square

4. The Elite theorem. In this section we present our main result, the Elite theorem. Let $M = (m_1, \dots, m_d)$ be a given profile of breadth m . The Elite theorem states that to find an optimal schedule for G it suffices to “look” at the Elite of G . In particular, if the cardinality of the Elite is larger than m_1 , then there exists an optimal M -schedule for G that starts with a subset of the Elite. Otherwise, there exists an optimal M -schedule starting with $E(G)$ and $m_1 - |E(G)|$ highest initial tasks from $L(G)$, choosing arbitrarily among tasks of the same height.

The Elite theorem enables us to ignore large portions of the graph at each step of the construction of an optimal schedule. As a special case, the Elite theorem also implies that if there is no initial task above the median, then HLF produces an optimal schedule.

Results similar to the Elite theorem were developed in [Wa81] and [DW82c]. They are the basis for several polynomial algorithms which find optimal schedules for certain restricted classes of precedence graphs and profiles of constant breadth. The Elite theorem is easily derived from the following theorem.

THEOREM 4.1. *Let S be a greedy M -schedule for $H(G)$ not longer than the length of an optimal M -schedule for G . Let f be the number of idle processors at the first slot of S . For any set T of f highest initial tasks of $L(G)$, there exists an optimal M -schedule S' for G with the properties:*

- i) $(S')_1 = (S)_1 \cup T$;
- ii) if $\lambda(S') > \lambda(S)$ then S' has idle periods only in its last slot.

Proof. The proof is by induction on r , the number of tasks of G of positive height. In the case $r = 0$ the graph does not contain any tasks of positive height and therefore, all the tasks of G are initial and the theorem obviously holds.

Assume that the theorem holds for every precedence graph of fewer than $r + 1$ tasks of positive height and let G be a graph with $r + 1$ such tasks. We distinguish between two cases, according to T' , the set of initial tasks in $L(G)$.

Case $|T'| < f$. Thus, the number of initial tasks in $L(G)$ is less than f which is the number of idle processors at the first slot of S . In this case $(S)_1$ contains all the initial tasks of $H(G)$, since we assumed that S is greedy. Furthermore, $T = T'$ and $(S)_1 \cup T$ is the set of all initial tasks of G . This implies that G contains fewer than $m_1 \leq m$ components; therefore, $\mu(G) = 0$, all the tasks in $L(G)$ are initial and $L(G) = T$. Thus, the schedule

$$S' = ((S)_1 \cup T) | (S)_2 | \cdots | (S)_{\lambda(S)}$$

for G has the same length as S , which implies the optimality of the schedule S' for G and M .

Case $|T'| \geq f$. Let λ be the length of an optimal M -schedule for G , then by assumption, $\lambda(S) \leq \lambda$. Let T be a set of f highest initial tasks of $L(G)$. Denote $\bar{G} = G - ((S)_1 \cup T)$ and let S^* be the schedule obtained from $(S)_2 | \cdots | (S)_{\lambda(S)}$ by removing all tasks not in $H(\bar{G})$ and making the resulting schedule greedy. Clearly $\lambda(S^*) \leq \lambda - 1$.

By M8, $H(\bar{G}) \subseteq A(G) - (S)_1$, which assures that S^* contains all the tasks of $H(\bar{G})$. Denote by $\bar{\lambda}$ the length of an optimal schedule for \bar{G} that fits $\bar{M} = (m_2, \dots, m_d)$. Since

λ is the length of an optimal schedule for G , $\lambda - 1 \leq \bar{\lambda}$, and therefore $\lambda(S^*) \leq \bar{\lambda}$. Moreover, the graph \bar{G} contains fewer than $r+1$ tasks of positive height, because $(S)_1 \cup T$ contains at least one task of positive height. Thus, the inductive hypothesis can be applied to S^* , ensuring the existence of \bar{S} , an optimal \bar{M} -schedule for \bar{G} , with the property of having idle processors in its last slot in the case $\lambda(\bar{S}) > \lambda(S^*)$.

Define $S' = ((S)_1 \cup T) | \bar{S}$. We have to prove that S' is an optimal M -schedule for G , and that it satisfies ii). Consider the following two subcases:

a) If $\lambda(S) \leq \lambda(S')$, then S' is clearly an optimal M -schedule for G , since by assumption S is not longer than the length of an optimal M -schedule for G .

b) Otherwise, $\lambda(S') = \lambda(\bar{S}) + 1 > \lambda(S)$. By the definition of S^* we get $\lambda(S) \geq \lambda(S^*) + 1$, which implies $\lambda(\bar{S}) > \lambda(S^*)$. Now, the inductive assumption guarantees that \bar{S} has idle processors only in its last slot. Since $(S)_1 \cup T$ has no idle processors, this implies that S' is an optimal M -schedule for G with idle processors only in its last slot. This completes the proof. \square

Now we are ready to prove the basic result of the paper.

THEOREM 4.2. The Elite theorem. *Let G be a precedence graph and M be a profile of breadth m . Then*

i) *If $E(G)$ contains more than m_1 tasks, then there exists an optimal schedule for G and M that starts with m_1 initial tasks of $E(G)$.*

ii) *If $E(G)$ contains m_1 tasks or fewer, then any set of m_1 highest initial tasks of G is a first slot of some optimal M -schedule for G .*

iii) *If $E(G) = \emptyset$, then HLF produces an optimal schedule for G and M that has idle periods only in its last slot.*

Proof. The proof follows from Theorem 4.1 and the fact that if $|E(G)| \leq m_1$, then there exists an optimal schedule for $H(G)$ and M that starts with $E(G)$; otherwise there exists an optimal schedule which starts with a subset of $E(G)$ of size m_1 . If $E(G)$ is empty, then $H(G)$ is empty and the empty sequence is an optimal schedule for $H(G)$. \square

We will demonstrate the Elite theorem on a few examples. Assume we need to schedule the precedence graphs of Figs. 4.1 and 4.2 on the profile $M = (3, 3, 2, 1, 3)$, which has breadth three.

In Fig. 4.1, $\mu(G_1) = 0$ and therefore, in finding an optimal schedule, we start by choosing $m_1(3)$ of the four tasks that are above the median. These four tasks are the Elite of the graph. Not every subset of three tasks of the Elite begins an optimal schedule. For example, if we choose $\{1, 2, 3\}$ as the first slot we would not get an optimal schedule. By the Elite theorem we know that there exists a set of three tasks, among the tasks in the Elite, that starts an optimal schedule. In G_1 , $\{1, 2, 5\}$ are such tasks.

If the precedence graph is the one given in Fig. 4.2, the situation is much simpler: there exists an optimal schedule starting with $E(G_2)$, and if we remove this set from the graph we obtain the graph described in Fig. 4.3. The Elite of this graph is empty ($E(G'_2) = \emptyset$) and thus, as we proved in the Elite theorem, HLF produces an optimal schedule for this graph. Therefore, the tableau T'_2 of Table 4.1 describes an optimal schedule for G'_2 , and T_2 describes an optimal schedule for the whole graph G_2 .

LEMMA 4.1. *Let G' be a closed subgraph of G . If the length of any HLF-schedule for G and M is bounded by λ , then the length of any HLF-schedule for G' and M is also bounded by λ . \square*

The following results follow from the Elite theorem and Theorem 4.1, and are useful in showing that HLF produces an optimal schedule for certain classes of precedence graphs and profiles.

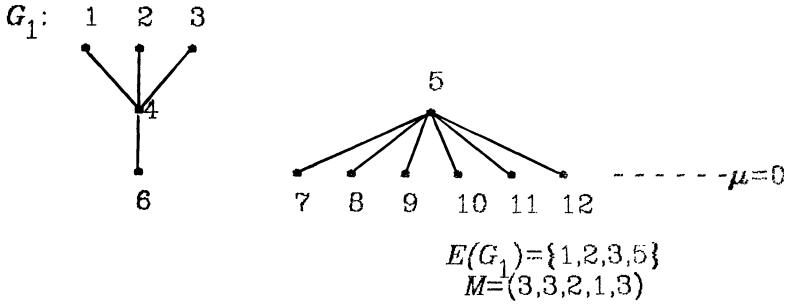


FIG. 4.1

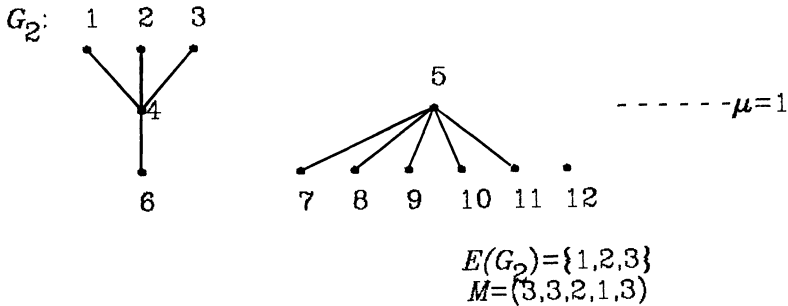


FIG. 4.2

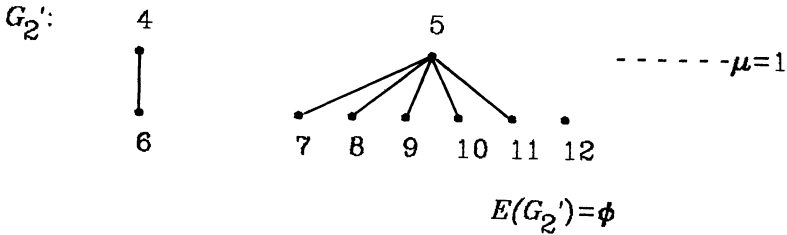


FIG. 4.3

TABLE 4.1.
The profile schedules for the graph G_2' and G_2 .

| slot | 2 | 3 | 4 | 5 |
|-------------|----|---|---|----|
| $T_2': P_1$ | 4 | 6 | 8 | 9 |
| P_2 | 5 | 7 | | 10 |
| P_3 | 12 | | | 11 |

| slot | 1 | 2 | 3 | 4 | 5 |
|------------|---|----|---|---|----|
| $T_2: P_1$ | 1 | 4 | 6 | 8 | 9 |
| P_2 | 2 | 5 | 7 | | 10 |
| P_3 | 3 | 12 | | | 11 |

LEMMA 4.2. Let λ be the length of an optimal M -schedule for G . If λ bounds the length of every HLF-schedule for $H(G)$ and M , then HLF produces an optimal M -schedule for G . In particular, if HLF is optimal for $H(G)$, then HLF is optimal for G .

Proof. The proof is by induction on the number of tasks of positive height in G . The case of no task of positive height is trivial. Assume that the lemma holds for every precedence graph of up to k tasks of positive height and let G be one with $k+1$ such tasks. Let T be any set of m_1 highest initial tasks of G . Denote by $G = G - T$ the remaining subgraph, and by M' the remaining profile. It is enough to show that;

- a) There exists an optimal M -schedule for G starting with T .

b) HLF produces an optimal M' -schedule for G' .

Proof of a). If $|E(G)| \leq m_1$, then by the Elite theorem there exists an optimal M -schedule for G starting with T . Otherwise $T \subseteq E(G)$, and by assumption it starts a schedule for $H(G)$ and M of length at most λ . Theorem 4.1 implies that it also starts an optimal schedule for G and M .

Proof of b). The subgraph G' contains fewer than $k+1$ tasks of positive height because T contains some. Since T starts an optimal M -schedule for G the length of an optimal schedule for G' is $\lambda - 1$. λ bounds the length of every HLF-schedule for $H(G)$ and M . Therefore $\lambda - 1$ bounds the length of every HLF-schedule for $H(G) - T$ and M' . By property M8 of the median, $H(G') \subseteq A(G) - T$, and thus, by Lemma 4.1, the length of every HLF-schedule for $H(G')$ and M' is bounded by $\lambda - 1$. Using the inductive assumption we conclude that HLF produces an optimal schedule for G' and M' . \square

5. HLF for inforest and outforest. In this section we study inforests and outforests, and analyze whether HLF produces an optimal schedule for such precedence graphs. An *inforest* (respectively *outforest*) is a graph in which each task has at most one immediate successor (respectively one immediate predecessor). HLF is optimal for specific types of profiles.

A profile M is *nondecreasing* (resp. *nonincreasing*) if $m_i \leq m_{i+1}$ (resp. $m_i \geq m_{i+1}$), for $1 \leq i \leq d - 1$; that is, the number of available processors does not decrease (resp. not increase) along the profile.

The results obtained in this section hold for more general types of profiles. We will see that if the profile is nondecreasing or nonincreasing but its amplitude of variation is bounded by one, then the complexity of the algorithms does not change.

We say that M is a *zigzag profile* if the following two conditions hold:

- i) $m_i + 1 \geq m_j$, for all ij such that $1 \leq i \leq j \leq d$.
- ii) $m_j + 1 \geq m_i$, for all ij such that $1 \leq i \leq j \leq d$.

If condition i) holds, then M is called a *nonincreasing zigzag profile*; if ii) holds, then it is called *nondecreasing zigzag profile*.

The profile of Table 2.1 is neither a nonincreasing nor a nondecreasing zigzag profile, since $m_4 - m_5 = -2$ and $m_4 - m_3 = -2$, respectively. In Table 5.1 we give an example of a nonincreasing zigzag profile; the profile is $M = (5, 4, 5, 2, 3, 3, 1)$. It is a nonincreasing but not a nondecreasing zigzag profile, since $m_7 - m_3 = -4$.

TABLE 5.1
A nonincreasing zigzag profile.

| slot | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-------|---|---|---|---|---|---|---|
| P_1 | | | | | | | |
| P_2 | | | | | | | |
| P_3 | | | | | | | |
| P_4 | | | | | | | |
| P_5 | | | | | | | |

Three HLF results for forests have appeared in the literature. The first and basic one is by Hu [Hu61] who showed that HLF produces an optimal schedule for inforests and straight profiles. Bruno [Br81] proved that HLF is optimal for outforests and straight profiles. The third result is by Garey et al. [GJ81]; they proved that HLF works for inforests and nonincreasing profiles in which $m_1 - m_d = 1$. We prove that HLF is

optimal if the precedence graph is an inforest (resp. outforest) and the profile is nondecreasing zigzag (resp. nonincreasing zigzag). Note that scheduling an inforest (resp. outforest) on a nonincreasing (resp. nondecreasing) profile is NP-hard if the breadth of the profile is arbitrary [GJ81], [Ma81], [Wa81] and polynomial if the breadth of the profile is constant [GJ81], [Wa81], [DW82c].

THEOREM 5.1. *Let G be an outforest and M be a nonincreasing zigzag profile of breadth m . Then HLF produces an optimal M -schedule for G .*

Proof. It suffices to prove the following: let H be an outforest, M be a nonincreasing profile of breadth m ; then for any set of m_1 highest initial tasks of H , there exists an optimal M -schedule starting with this set. Note that if H contains less than m_1 initial tasks, then the set of all initial tasks are the only set of m_1 highest initial tasks.

We prove the theorem by applying the Elite theorem and using the inequality

$$(*) \quad |E(G)| \leq m - 1 \leq m_1.$$

The inequality $m - 1 \leq m_1$ holds because M is a nonincreasing zigzag profile of breadth m . By the definition of $H(G)$, it has fewer than m components. Each component of $H(G)$ is an outtree having one initial task, and this task is the only task from that outtree in the Elite of G . This implies that the Elite of G has fewer than m tasks and $(*)$ holds. \square

We now prove a similar result for inforests. Let G be a precedence graph; denote by G_z the subgraph of G obtained by removing all tasks of height zero. Observe that an optimal schedule for G_z is at least by one shorter than an optimal schedule for G , since the last slot of any schedule can only have tasks of height zero.

THEOREM 5.2. *Let G be an inforest and M be a nondecreasing zigzag profile of breadth m . Then HLF produces an optimal M -schedule for G .*

Proof. Assume to the contrary that the theorem does not hold, and let G be an inforest with a minimal number of tasks, such that HLF is not optimal (for some nondecreasing zigzag profile of breadth m). If $L(G) \neq \emptyset$, then $H(G)$ contains less tasks than G . Thus, HLF is optimal for $H(G)$, and by Lemma 4.2 it is also optimal for G . This proves that the minimality of G requires that $L(G) = \emptyset$, and that G contains at most $m - 1$ components.

Let M be any nondecreasing zigzag profile of breadth m and let λ be the length of an optimal schedule for G and M . Assume that M has length λ . Notice that $m_\lambda \geq m - 1$, i.e., all tasks of G of height zero fit into the last slot of M . The minimality of G implies that HLF is optimal for G_z and M . But every HLF-schedule for G and M can be obtained from an HLF-schedule for G_z and M by scheduling all tasks of height zero in the last slot of M (which was empty) and making the resulting schedule greedy. The HLF-schedules for G and M are of length λ and therefore optimal. This is a contradiction. \square

Theorems 5.1 and 5.2 can be further improved using Lemma 4.2. Lemma 4.2 proves that it is enough to require that only $H(G)$ will be an outforest (resp. inforest) for Theorem 5.1 (resp. Theorem 5.2) to hold.

The following examples show that Theorems 5.1 and 5.2 are tight. Table 5.2 presents an HLF-schedule for the inforest G_1 of Fig. 5.1 that fits the profile $M_1 = (3, 3, 1, 1, 1)$ but is not optimal. Notice that M_1 is not a nondecreasing zigzag profile. Table 5.3 presents a nonoptimal HLF-schedule for the outforest G_2 of Fig. 5.2 that fits the profile $M_2 = (1, 2, 3, 3, 3, 3, 3)$. Notice that M_2 is not a nonincreasing zigzag profile. If one starts with task 3, then it is easy to find an optimal schedule, which is shorter than the schedule of Table 5.3.

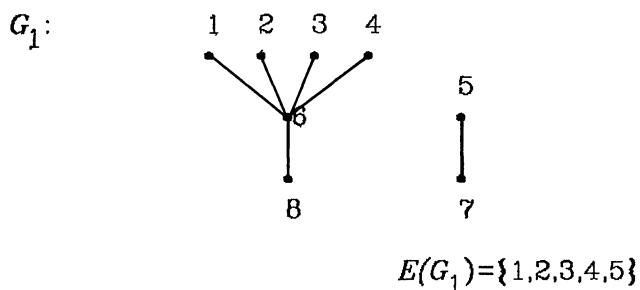


FIG. 5.1

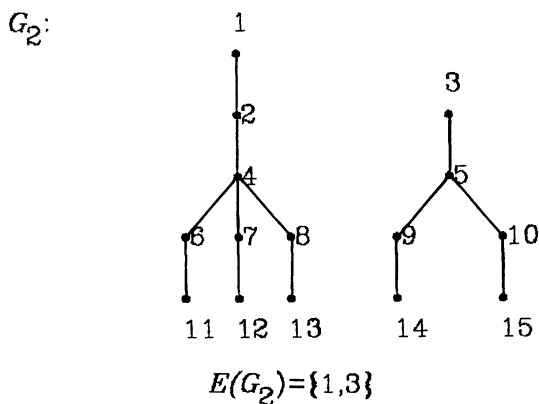


FIG. 5.2

TABLE 5.2
A HLF-schedule for the graph of G_1
of Fig. 5.1 and the profile $M = (3, 3, 1, 1, 1)$.

| slot | 1 | 2 | 3 | 4 | 5 |
|-------|---|---|---|---|---|
| P_1 | 1 | 4 | 6 | 7 | 8 |
| P_2 | 2 | 5 | | | |
| P_3 | 3 | | | | |

TABLE 5.3
A HLF-schedule for the graph G_2 of Fig. 5.2 and the profile
 $M_2 = (1, 2, 3, 3, 3, 3, 3)$.

| slot | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|-------|---|---|---|---|----|----|----|
| P_1 | 1 | 2 | 4 | 6 | 9 | 12 | 15 |
| P_2 | | 3 | 5 | 7 | 10 | 13 | |
| P_3 | | | | 8 | 11 | 14 | |

6. Opposing forests. We say that a graph is an *opposing forest graph* if all its components are intrees or outtrees, that is, it is composed of an inforest and an outforest. It was shown that HLF is optimal for scheduling an inforest [Hu61] or an outforest [Br81] on a straight profile with arbitrary breadth. On the other hand scheduling an opposing forest on straight profiles with arbitrary breadth is NP-hard [GJ81], [Ma81], [Wa81].

If the profile is straight and its breadth m is fixed, then scheduling an opposing forest is polynomial [GJ81], [Wa81], [DW82c]. These algorithms have rather complex time bounds (m appears in the exponent). We use the results of this section, which are derived from the Elite theorem, to obtain a linear algorithm for the special case of straight profiles of breadth three and opposing forests. This improves the $O(n \log n)$ time bound of an algorithm presented in [GJ81] for this case. Our approach also leads to an $O(n \log n)$ algorithm for scheduling an opposing forest on a zigzag profile of breadth three (either two or three processors in each time slot).

Goyal [Go76] proved that HLF is optimal for series-parallel graphs [LT79] and straight profiles of breadth two. Since opposing forests are series-parallel graphs, HLF is also optimal for opposing forests and straight profiles of breadth two. In the case of scheduling opposing forests we can generalize Goyal's result to any profile of breadth two.

THEOREM 6.1. *Let G be an opposing forest³ and M be a profile of breadth two. Then HLF produces an optimal M -schedule for G .*

Proof. In case of breadth two $H(G)$ contains at most one component. Therefore, $H(G)$ is either an intree or an outtree. Note that any profile of breadth two is a zigzag profile. By Theorem 5.1, Theorem 5.2, and Lemma 4.2 we conclude that HLF produces an optimal schedule for G . \square

Note that Theorem 6.1 holds also for graphs that are not opposing forests whose highest component is either an intree or an outtree. It is easy to see that for arbitrary graphs and zigzag profiles of breadth two the Coffman–Graham algorithm [CG72] produces an optimal schedule. This algorithm corresponds to a restricted version of HLF. For profiles of breadth three choosing tasks according to highest height does not necessarily lead to optimal schedules.

For example, HLF does not produce optimal schedules for the graph of Fig. 6.1 and the straight profile of breadth three. Task 8 must be scheduled in an earlier slot than task 7. But always preferring the outtree tasks does not lead to an optimal schedule either. We need a criterion that tells us when to prefer outtree tasks over intree tasks. Such a criterion will be provided by a theorem proven below: If there is an initial outtree task x of the same height as the whole opposing forest G , then any set of three highest tasks of G that contains x starts an optimal schedule for G .

Notice that the above criterion does not apply to the graph of Fig. 6.1, since task 8 is not a task of maximum height. In this case, we “flip” the graph. Let G^R denote the graph obtained by reversing all the edges of G . Note that for an opposing forest, either G or G^R contains an outforest task of height $h(G)$.

In the example (Fig. 6.1 and Table 6.1) we can remove the set {12, 17, 18}. Since we remove this triplet from the reversed graph, we schedule it in the last slot of the schedule. The remaining subgraph is shown in Fig. 6.2.

Now both the top and the bottom contain an outtree of maximum height. We can choose either of the sides to continue the algorithm. Assume we choose {9, 15, 16} and

³ Actually the theorem holds for any series-parallel graph. This can be proven by a simple induction on the size of the graph using Lemma 4.2.

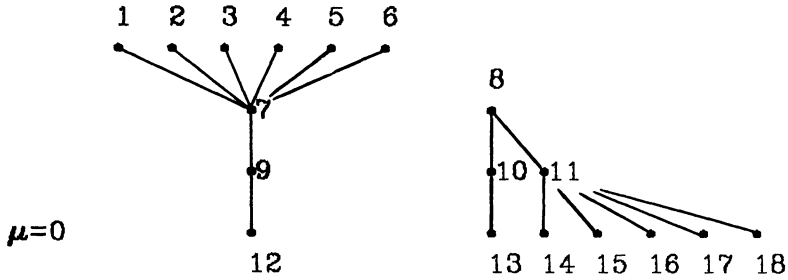


FIG. 6.1

TABLE 6.1
First step.

| slot | 1 | 2 | 3 | 4 | 5 | 6 |
|-------|---|---|---|---|---|----|
| P_1 | | | | | | 12 |
| P_2 | | | | | | 17 |
| P_3 | | | | | | 18 |

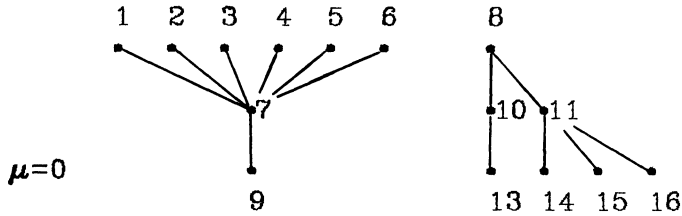


FIG. 6.2

TABLE 6.2
Second step.

| slot | 1 | 2 | 3 | 4 | 5 | 6 |
|-------|---|---|---|---|----|----|
| P_1 | | | | | 9 | 12 |
| P_2 | | | | | 15 | 17 |
| P_3 | | | | | 16 | 18 |

put them in the next slot from the bottom. We are left with the subgraph of Fig. 6.3. At this step we have to switch back to the top, because the reversed graph does not have an outtree of maximum height.

We remove $\{8, 1, 2\}$ from the graph and insert them in the first slot of the schedule. We now obtain the graph in Fig. 6.4.

The Elite of this graph is empty and therefore by the Elite theorem we know that HLF produces an optimal schedule for the resulting graph. We complete the schedule according to HLF filling slots 2, 3 and 4 and obtain Table 6.4.

The above example demonstrates the Flip-Flop algorithm which will be presented in the next section. A high-level description of the algorithm is given at the beginning of the next section.

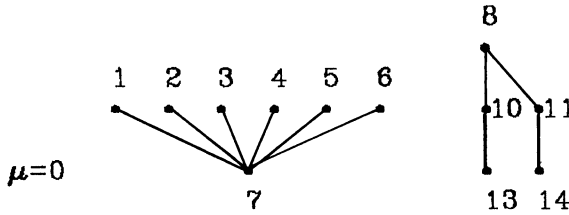


FIG. 6.3.

TABLE 6.3
Third step.

| slot | 1 | 2 | 3 | 4 | 5 | 6 |
|-------|---|---|---|---|----|----|
| P_1 | 8 | | | | 9 | 12 |
| P_2 | 1 | | | | 15 | 17 |
| P_3 | 2 | | | | 16 | 18 |

$\mu=2$ - - - - -

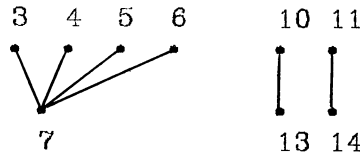


FIG. 6.4.

TABLE 6.4
Last step.

| slot | 1 | 2 | 3 | 4 | 5 | 6 |
|-------|---|---|----|----|----|----|
| P_1 | 8 | 3 | 6 | 7 | 9 | 12 |
| P_2 | 1 | 4 | 10 | 13 | 15 | 17 |
| P_3 | 2 | 5 | 11 | 14 | 16 | 18 |

The following lemma proves that among tasks of the same height, we can always schedule tasks of an outtree first.

LEMMA 6.1. *Let S be an optimal M -schedule for G . Assume that y is an initial task of an intree component and that x is an initial task of an outtree component, such that $h(x) \geq h(y)$. If $y \in (S)_1$ and $x \notin (S)_1$ then there exists an optimal M -schedule, \hat{S} , for G that starts with x instead of y , that is,*

$$(\hat{S})_1 = ((S)_1 - \{y\}) \cup \{x\}.$$

Proof. Let π_x be a longest path in G that starts with x , and π_y the longest path that starts with y . The path π_x is not shorter than π_y , because $h(x) \geq h(y)$. The order of the tasks on these paths is the same order in which they appear in the schedule S . Let k be the first slot in S at which the total number of tasks along π_x in slot k and in all previous slots, is equal to the corresponding number of tasks along π_y . There

must be such a slot, because S starts with π_y and not with π_x and the latter is not shorter than the former. Moreover, no task of π_y exists in the slot $(S)_k$, otherwise, k would not be the first.

Since the component of x is an outtree, every task in π_x has at most one immediate predecessor which is the previous task along the path π_x . Similarly, every task along π_y has at most one immediate successor, which is the next task along π_y . Therefore, we can exchange the tasks of π_x and π_y which appear in the first k slots, one by one, respectively. The new schedule obtained is of the same length as S , and none of the precedence constraints represented by G are violated, simply because every task along π_x moves upward following its only immediate predecessor, and every one along π_y moves downward preceding its only immediate successor. Since in the k th slot there is no task of π_y , we can exchange the member of π_x , which is in $(S)_k$, with the last member of the path π_y that is in a slot of lower index than k . Thus, we have obtained the desired optimal schedule that starts with the set $\{(S)_1 - \{y\}\} \cup \{x\}$. \square

In the following theorem we use the Elite theorem to show that the inforest tasks can be chosen according to height.

THEOREM 6.2. *Let O be an outtree and I be an inforest. Let x be the root of O and let M be a zigzag profile of breadth three. Then i) or ii) holds.*

i) *For any set T_1 of $m_1 - 1$ highest initial tasks of I there exists an optimal M -schedule for $O \cup I$ starting with $T_1 \cup \{x\}$.*

ii) *For any set T_2 of m_1 highest initial tasks of I there exists an optimal M -schedule for $O \cup I$ starting with T_2 .*

Proof. If $H(O \cup I)$ is an outtree (resp. inforest), then by Theorem 5.1 (resp. Theorem 5.2) HLF is optimal for $H(O \cup I)$ and M . Furthermore, Lemma 4.2 implies that HLF is optimal for the whole graph $O \cup I$ and M , and the theorem holds.

Assume that $O \cup I$ is a counterexample with the fewest number of vertices. By the above remarks we know that $H(O \cup I)$ consists of the outtree O and an intree which we call N . Theorem 4.1 guarantees that if i) or ii) holds for $H(O \cup I)$ and M then it also holds for $O \cup I$ and M . Thus $O \cup I$ is not a minimal counterexample unless $L(O \cup I) = \emptyset$, i.e. $I = N$.

Let y be the task of N of height zero, and let S be any optimal schedule for $O \cup N$ and M . Assume y appears in slot k of S and let G' be the subgraph of $O \cup N$ induced by the tasks which are in the first $k - 1$ slots of S . Observe that G' contains less tasks than $O \cup N$. Thus i) or ii) must hold for G' and M . Since $h_{G'}(x) = h_N(x) - 1$, for any task x of N in G' , a set of highest initial tasks from N in G' is a set of highest initial tasks of N . We conclude that i) or ii) holds for $O \cup N$ and M , which contradicts the minimality of $O \cup N$. \square

The following theorem, which is a consequence of the previous two theorems, leads to the Flip-Flop algorithm. Note that for any opposing forest G , either G or G^R contains an outtree task of height $h(x)$.

THEOREM 6.3. *Let G be an opposing forest that contains an outtree of height $h(G)$. Let M be a zigzag profile of breadth three; let x be the initial task of an outtree with height $h(G)$; and let A be any set of m_1 highest initial tasks of G that contains x . Then there exists an optimal schedule that starts with A .*

Proof. If $H(G)$ is an outforest, then the theorem holds by Theorem 5.1 and Lemma 4.2. Theorem 4.1 implies that it is sufficient to prove the theorem for the case when $L(G)$ is empty. Thus G consists of an intree and an outtree and we can apply the previous theorem. If case i) holds, then we are done. Assume ii) holds and let R be any set of m_1 highest initial tasks of the intree. This starts an optimal schedule for G and M . Let z be a minimum height task of R . Then $A = R - \{z\} \cup \{x\}$ is an arbitrary

set of m_1 highest initial tasks of G which contains the root of the outtree. By Theorem 6.1 this set starts an optimal schedule for G and M . \square

7. The Flip-Flop algorithm. We first give a high-level description of the Flip-Flop algorithm that produces an optimal schedule for an opposing forest, G , and a zigzag profile, M , of breadth three. The algorithm has two phases, a *Flip-Flop phase* and an *HLF phase*. In the Flip-Flop phase we deal with the case when G has an intree and an outtree above the median. We iteratively remove sets of initial tasks from G or from the reverse graph G^R to fill up slots in the schedule we are constructing. To do this we apply Theorem 6.3, i.e. we remove from G if there is an outtree in G of height $h(G)$ and from G^R if there is an outtree in G^R of height $h(G)$. Notice that one of the two cases must hold and that we always choose a set of highest initial tasks that contains the root of the highest outtree. The sets of initial tasks removed from G are put in the first, second, \dots , time slot, and those removed from G^R are put in the last, second to last, \dots , time slot.

We stop the Flip-Flop phase and enter the HLF phase as soon as the HLF condition holds (given below). In the HLF phase we schedule the remaining graph according to highest-level-first. Our ability to stop the Flip-Flop phase assures the linearity of the algorithm, because we do not have to keep track of too many components.

The Flip-Flop algorithm produces two sequences of sets. One consists of the sets which were removed from the top (from G) and the other consists of the set removed from the bottom (from G^R). If the two sequences do not overlap, then we get a valid schedule for G . In the case of a straight profile we run the Flip-Flop algorithm on a very long straight profile and then just omit the empty slots between the two sequences to get an optimal schedule. In this case the Flip-Flop algorithm is linear. For zigzag profiles another factor of $\log n$ is required to find a minimum length valid schedule.

DEFINITION. We say that the HLF-condition holds for the opposing forest G if there are either only outtree components or only intree components above the median.

LEMMA 7.1. *If the HLF-condition holds for an opposing forest G , then HLF produces an optimal schedule for G and any zigzag profile of breadth three.*

Proof. Follows from Lemma 4.2, Theorem 5.1 and Theorem 5.2. \square

The only remaining case in which the HLF-condition does not hold is when the opposing forest has exactly two components above the median: an outtree, and an intree with more than one initial task. Note that an intree that contains only one initial task is a chain of tasks and a chain is also an outtree.

We are now ready to present the Flip-Flop algorithm. The variables i_{top} and i_{bot} will point to the next empty slot in the corresponding end of the profile M . Initially they point to the first and last slots of the profile M , respectively. The variable \hat{G} represents the opposing forest remaining at the current step of the algorithm. Initially, \hat{G} is equal to G , and after each removal of a set of tasks from the graph the variable \hat{G} becomes the resulting subgraph.

THE FLIP-FLOP ALGORITHM

(*Flip-Flop phase*)

REPEAT

 WHILE not HLF? (\hat{G}) and an outtree component is the highest component of \hat{G} DO:

 BEGIN

 Remove the initial task of the highest outtree and any other $m_{i_{\text{top}}} - 1$ highest initial tasks from \hat{G} . Add these tasks as slot i_{top} to the schedule S . Increase i_{top} by one.


```

END
IF not HLF? ( $G$ ) THEN
  WHILE not HLF? ( $\hat{G}^R$ ) and an outtree component is the highest component
  of  $\hat{G}^R$  DO:
    BEGIN
      Remove the initial task of the highest outtree and any other  $m_{i_{\text{bot}}} - 1$  highest
      initial task from  $\hat{G}^R$ . Add these tasks as slot  $i_{\text{bot}}$  to the schedule  $S$ . Decrease
       $i_{\text{bot}}$  by one.
    END
  UNTIL HLF? ( $G$ ) or HLF? ( $G^R$ )
(*HLF phase*)
  IF HLF? ( $\hat{G}$ ) THEN continue to fill the schedule  $S$ , updating  $i_{\text{top}}$  by applying
  the HLF strategy to  $\hat{G}$  FI.
  If HLF? ( $\hat{G}^R$ ) THEN continue to fill the schedule  $S$ , updating  $i_{\text{bot}}$  by applying
  the HLF strategy to  $\hat{G}^R$  FI.
(*check for overlapping of the two sequences*)
  IF  $i_{\text{top}} < i_{\text{bot}}$  THEN the constructed schedule fits the profile ELSE return failure.

```

THEOREM 7.1. *The Flip-Flop algorithm produces a schedule for an opposing forest G and a zigzag profile of breadth three. If there is no feasible schedule it returns failure.*

Proof. To prove the correctness of the Flip-Flop algorithm it is enough to prove that if there exists a feasible schedule, then the algorithm will find one. It suffices to show that all sets of initial tasks that are removed from \hat{G} (resp. \hat{G}^R) start some optimal schedule for \hat{G} (resp. \hat{G}^R) and the corresponding profiles. Theorem 6.3 assures this for the Flip-Flop phase and Lemma 7.1 for the HLF phase. \square

THEOREM 7.2. *The Flip-Flop algorithm can be implemented in time $O(n)$.*

Proof. We just sketch how to do this. A more detailed description is given in [DW2b]. First we outline that with some simple data structures an inforest or an outforest (and therefore an opposing forest) can be scheduled in time $O(n)$. We keep the initial tasks of the forest in an array of lists. The i th list contains all initial tasks of level i . To facilitate the removal of up to three highest initial tasks we remember the highest three levels that contain initial tasks. After each removal of an initial set we update the array of lists and redetermine the highest three levels. We conclude that HLF is linear for any kind of forest. In our algorithm we sometimes remove initial tasks from the reversed graph according to HLF. This can be handled by keeping dual data structures for G^R . So far we reasoned that the HLF phase is linear.

In the Flip-Flop phase we remove from \hat{G} as well as from \hat{G}^R . Thus the heights of the tasks in \hat{G} and \hat{G}^R might change and we need some more insights in the algorithm to assure linearity. Observe that if the HLF condition does not hold then there are at least three tasks in the Elite. Thus during the entire Flip-Flop phase we only remove tasks from the highest two components of G which must be an intree and an outtree. As soon as the outtree splits into several outtrees then at most one of them will remain above the median and then this outtree will be a highest one. Components that drop below the median do not need to be considered any more during the Flip-Flop phase.

It is easy to keep track of the highest outtree vertex in $H(\hat{G})$ and $H(\hat{G}^R)$. But we also need to remove sets of highest initial tasks from the intree of $H(\hat{G})$ and $H(\hat{G}^R)$, even though the heights of the tasks of the intree in $H(\hat{G})$ and $H(\hat{G}^R)$ are changing during the Flip-Flop algorithm. If we remove the highest outtree vertex of $H(\hat{G})$ (resp. $H(\hat{G}^R)$), then the heights of all intree tasks in $H(\hat{G}^R)$ (resp. $H(\hat{G})$) drop

by one. Thus their relative heights in $H(\hat{G})$ and $H(\hat{G}^R)$ remain the same and we do not need to update the heights. This completes the sketch of the linear implementation. \square

In the following theorems we show how we can use the Flip-Flop algorithm for finding optimal schedules.

THEOREM 7.3. *The Flip-Flop algorithm implies a linear algorithm to find an optimal schedule for an opposing forest and a straight profile of breadth three.*

Proof. We choose the initial profile to be of length n and run the Flip-Flop algorithm. The resulting schedule might contain some empty slots in the middle, that is, after the algorithm terminates $i_{\text{top}} < i_{\text{bot}}$. By removing the empty slots we get an optimal schedule. \square

The proof of Theorem 7.2 does not hold for general zigzag profile, because the difference in the slots size prevents us from just squeezing the schedule to obtain an optimal one.

THEOREM 7.4. *The Flip-Flop algorithm implies an $O(n \log n)$ algorithm to find an optimal schedule for an opposing forest and a zigzag profile of breadth three.*

Proof. We run the Flip-Flop algorithm for different schedule lengths. The legitimate lengths are between $n/3$ and n ; therefore by binary search we can find the shortest schedule that fits the profile in time $O(n \log n)$. \square

Observe that in Flip-Flop phase only tasks of $E(\hat{G})$ and $E(\hat{G}^R)$, respectively, are removed. Therefore, the algorithm can be easily extended to work for every graph G for which $H(G)$ is an opposing forest. The basic reason is that tasks that are under the median will not “pop up” to be above the median, if we remove tasks according to height from $L(G)$ and $L(G^R)$, respectively (see properties M7 and M8 of the median). The correctness proof for this extended algorithm remains the same, using Lemma 4.2. For the sake of simplicity we restrict ourselves to the case where the whole graph is an opposing forest. The time complexity for running the Flip-Flop algorithm on a graph G for which $H(G)$ and $H(G^R)$ are opposing forests will be $O(n + e)$, where e is the number of edges in G . Note that in opposing forests e is $O(n)$.

Acknowledgments. We would like to thank Barbara Simons for many useful suggestions and thorough proofreading of this paper. We are also very grateful to the referee for simplifying some of our proofs.

REFERENCES

- [AH74] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
- [Br81] J. BRUNO, *Deterministic and stochastic scheduling problems with treelike precedence constraints*, NATO Conference, Durham, England, July 1981.
- [Co76] E. G. COFFMAN, JR., ed. *Computer and Job Shop Scheduling Theory*, John Wiley, New York, 1976.
- [CD73] E. G. COFFMAN, JR AND P. J. DENNING, *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, NJ, 1973.
- [CG72] E. G. COFFMAN, JR. AND R. L. GRAHAM, *Optimal scheduling for two-processor systems*, Acta Inform., 1 (1972), pp. 200–213.
- [Do80] D. DOLEV, *Scheduling wide graphs*, Technical Report STAN-CS-80-832, Dept. Computer Science, Stanford Univ., Stanford, CA, December 1980.
- [DW82a] D. DOLEV AND M. K. WARMUTH, *Scheduling precedence graphs of bounded height*, J. Algorithms, 5 (1984), pp. 48–59.
- [DW82b] ———, *Scheduling flat graphs*, IBM Research Report RJ3398, May 1982.
- [DW82c] ———, *Profile scheduling of opposing forests and level orders*, SIAM J. Alg. Discr. Meth., 6 (1985), to appear.

- [Ga81] H. N. GABOW, *A linear-time recognition algorithm for interval dags*, Inform. Proc. Lett., 12 (1981), pp. 20-22.
- [Ga82] ———, *An almost linear algorithm for two processor scheduling*, J. Assoc. Comput. Mach., 29 (1982), pp. 766-780.
- [GJ79] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman, San Francisco, 1979.
- [GJ81] M. R. GAREY, D. S. JOHNSON, R. E. TARJAN AND M. YANNAKAKIS, *Scheduling opposing forests*, Technical Report, Bell Laboratories, Murray Hill, NJ, 1981; SIAM J. Alg. Discr. Meth., 4 (1983), pp. 72-93.
- [GL79] R. L. GRAHAM, E. L. LAWLER, J. K. LENSTRA AND A. H. G. RINNOOY KAN, *Optimization and approximation in deterministic sequencing and scheduling: A survey*, Ann. Discrete Math., 5 (1979), pp. 287-326.
- [Go76] D. K. GOYAL, *Scheduling series parallel structured tasks on multiprocessor computing systems*, Technical Report CS-76-034, Dept. Computer Science, Washington State Univ., Pullman, September 1976.
- [Hu61] T. C. HU, *Parallel sequencing and assembly line problems*, Operations Res., 9 (1961), pp. 841-848.
- [LR78] J. K. LENSTRA AND A. H. G. RINNOOY KAN, *Complexity of scheduling under precedence constraints*, Oper. Res., 26 (1978), pp. 22-35.
- [LT79] E. L. LAWLER, R. E. TARJAN AND J. VALDES, *The recognition of series parallel digraphs*, Proc. the 11th Annual ACM Symposium on Theory of Computing, Atlanta, GA, 1979, pp. 1-12.
- [Ma81] E. W. MAYR, *Well structured parallel programs are not easier to schedule*, Technical Report STAN-CS-81-880, Dept. Computer Science, Stanford Univ., Stanford, CA, September 1981.
- [PY79] C. H. PAPADIMITRIOU AND M. YANNAKAKIS, *Scheduling interval-ordered tasks*, this Journal, 8 (1979), pp. 405-409.
- [U175] J. D. ULLMAN, *NP-complete scheduling problems*, J. Comput. System Sci., 10 (1975), pp. 384-393.
- [Wa81] M. K. WARMUTH, *Scheduling on profiles of constant breadth*, Ph.D. thesis, Dept. Computer Science, Univ. Colorado, Boulder, August, 1981.

ON COMPARABILITY AND PERMUTATION GRAPHS*

JEREMY SPINRAD†

Abstract. This paper presents a technique for orienting a comparability graph transitively in $O(n^2)$ time. The best previous algorithm for this problem required $\Omega(n^3)$ time. When combined with a result in [SP], we can recognize permutation graphs in $O(n^2)$ time, and determine in the same time complexity whether two permutation graphs are isomorphic. The orientation algorithm can also be used to reduce the problem of recognizing comparability graphs to that of recognizing transitive graphs. This gives an upper bound of $O(n^{2.49+})$ for comparability graph recognition, while the fastest previous algorithms required $\Omega(n^3)$ time.

Key words. complexity, algorithms, comparability graphs, transitive orientation, permutation graphs

Background. The transitive orientation problem takes an undirected graph G as input, and assigns directions to each edge in G so that the resulting graph G' is transitive, if such an orientation exists. Previous algorithms for the problem [GH], [GOL], [PEL] work by orienting a single edge of the graph, and finding other edges whose orientation is forced. These algorithms have a running time of $O(dm)$, where d is the maximum vertex degree in the graph, and m is the number of edges in the graph. This paper presents an $O(n^2)$ algorithm for the orientation problem, where n is the number of vertices in the graph. The new algorithm orients all edges between two sets of vertices, and uses a partition refinement scheme to orient the rest of the graph.

Some problems seem to be easier to solve for directed graphs than for their undirected counterparts: For instance, recognizing transitive digraphs seems easier than recognizing transitively orientable graphs, and recognizing transitive series-parallel digraphs seems easier than recognizing cographs. However, the problems on undirected graphs can be solved by applying the transitive orientation algorithm, and then testing whether the resulting directed graph is in the appropriate class of graphs. The algorithm presented in this paper reduces the time complexity of the following algorithms: comparability graph recognition reduces to transitive digraph recognition, from $\Omega(n^3)$ [PEL], [GH], [GOL] to $O(n^{2.49+})$ [CW], and permutation graph recognition and isomorphism reduce to the problems for two-dimensional partial orders, from $\Omega(n^3)$ [COL], [PEL] to $O(n^2)$ [SP].

Definitions. Definitions of terms not found in this section are standard, and may be found in [AHU].

A directed graph is *transitive* if the existence of the edges (x, y) and (y, z) in E implies the existence of (x, z) in E .

If there are three vertices x, y and z such that (x, y) and (y, z) are edges but (x, z) is not an edge, there is a *transitivity violation* involving x, y and z . A directed graph is transitive if and only if there are no transitivity violations between any trio of vertices.

An undirected graph can be *oriented* into a directed graph by transforming each undirected edge (x, y) in E into a directed edge (x, y) or (y, x) .

An undirected graph is a *comparability graph* if it can be oriented into a transitive graph.

* Received by the editors October 19, 1982, and in final revised form April 15, 1984.

† School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332.

An undirected graph G is a *permutation graph* if there is some pair of permutations of the vertex set P_1, P_2 such that there is an edge between x and y in G if and only if x precedes y in P_1 and P_2 , or y precedes x in P_1 and P_2 . Every permutation graph is a comparability graph; the converse is not true.

We say that two vertices u, v are *related* if there is an edge between u and v . This edge may be directed or undirected. If there is no edge between the two vertices, then u and v are *unrelated*.

Modular decomposition. The algorithms presented in this paper rely on a technique called “modular decomposition”, which was developed in [SP]. This section gives a definition of the modular decomposition of a graph. The definitions apply to both directed and undirected graphs.

In the remainder of the present section, definitions apply to a simple graph $G = (V, E)$, where V is the set of vertices, and E is the set of edges.

A *module* is a subset M of vertices of V with the property that every vertex in $V - M$ is related to all vertices in M or no vertices in M . Note that between any two modules there are either no edges or all possible edges (as in a complete bipartite graph).

Let M' be the graph with vertex set M , and an undirected edge edge between u and v if and only if there is an edge between u and v in G . M is a *connected module* if and only if M' is a connected graph.

Let M'' be the graph with vertex set M , and an undirected edge between u and v if and only if there is no edge between u and v in G . M is *complement-connected* if and only if M'' is a connected graph.

Suppose a module M is not connected. M can be partitioned into two modules M_1, M_2 such that no vertex in M_1 is related to a vertex in M_2 . Therefore, unconnected modules are called *parallel modules*, due to the similarity between an unconnected module and a parallel connection in a series-parallel graph. If M is not complement-connected, M can be partitioned into M_1, M_2 such that every vertex in M_1 is related to every vertex in M_2 . Thus, a module which is not complement-connected is called a *series module*. A module which is both connected and complement-connected is called a *neighborhood module*.

In Fig. 1, module J is a series module, module K is a parallel module, and module L is a neighborhood module. Every module is exactly one of the types series, parallel or neighborhood. By definition, a module cannot be both a neighborhood module and a series or parallel module. A module cannot be a series module and a parallel module; if the module is not connected, every vertex in one component will be connected to every vertex which is not in that component in the complement graph, so the module will be complement-connected.

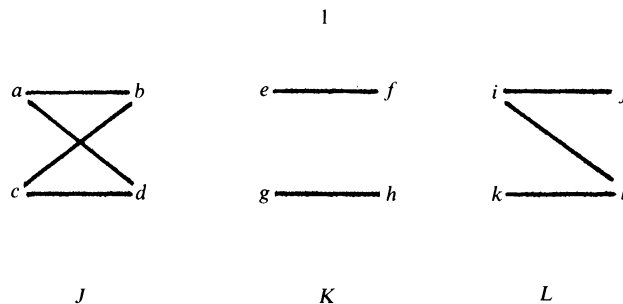


FIG. 1

It is easy to see that a series or parallel module M can be decomposed into the connected components of M' or M'' . It is more difficult to see how a neighborhood module can be decomposed.

A module M is a *maximal submodule* of a neighborhood module N if no proper submodule of N contains M . In [SP], there is a proof that every vertex contained in a neighborhood module N is in a unique maximal submodule of N . Thus, the maximal modules partition the vertex set of N .

The following recursive algorithm can be used to divide any graph G into a unique tree structure which will be called the *modular decomposition* of G .

First, consider the module M consisting of the entire graph. If M is a single vertex, halt; the vertex itself is the modular decomposition of G .

M is either a series, parallel or neighborhood module. Create a node in the structure labeled with S , P or N , depending on the type of the module.

Let K be a node representing a parallel (series) module, and let M_1, M_2, \dots, M_i be the connected components of $M'(M'')$. Find the modular decomposition of each M_j , and make these the children of node K .

Let K be a node representing a neighborhood module M , and let M_1, M_2, \dots, M_i be the maximal submodules of M . Find the modular decomposition of each M_j , and make these the children of node K .

The result of this process is a tree with the vertices of G as leaves. The structure is an extension of the tree representation of series-parallel graphs given in [VTL]; series-parallel partial orders are exactly those graphs which have no neighborhood nodes in their modular decomposition. There are $O(n)$ vertices in the modular decomposition of a graph.

Figure 2 gives a sample graph with its modular decomposition.

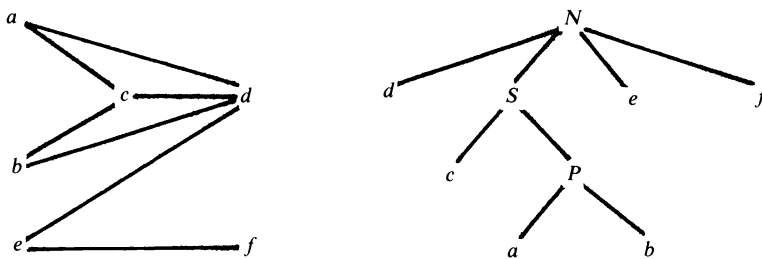


FIG. 2

A *representative graph* of a module M in the modular decomposition is a graph induced by a subset consisting of a single vertex from each maximal submodule of M . The representative graph is independent of the choice of representative vertices from the maximal submodules.

In [SP] and [MS], algorithms are developed which find the modular decomposition of a graph in $O(n^2)$ time. The algorithms in this paper will assume the existence of the modular decomposition of the graph.

Orienting comparability graphs: An overview. This section presents the outline of an algorithm which orients comparability graphs into transitive graphs in $O(n^2)$ time. The algorithm performs a postorder traversal of the modular decomposition tree, orienting each module transitively when given orientations for all submodules. Parallel and series modules prove to be easy to orient; orientation of neighborhood modules is accomplished by a partition refinement scheme.

Lemma 1 allows the problem to be simplified. It says that we only need to find transitive orientations of each representative graph in the modular decomposition of G .

LEMMA 1. *There is a transitive orientation of G if and only if there is a transitive orientation of every representative graph in the modular decomposition of G .*

Proof. Let T be any transitive orientation of G . We can construct a transitive orientation for the representative graph of any module M as follows. For each vertex v in the representative graph of M , let $r(v)$ be any representative vertex in the submodule corresponding to v . An edge is directed from x to y in the representative graph of M if and only if the edge $(r(x), r(y))$ is directed from $r(x)$ to $r(y)$ in T . If there were three vertices x, y, z of the representative graph which had edges directed from x to y , and from y to z , with no edge directed from x to z , then in T there would be edges from $r(x)$ to $r(y)$ and from $r(y)$ to $r(z)$, but there would be no edge from $r(x)$ to $r(z)$. Since T has a transitive orientation, this cannot occur. Therefore, T defines a transitive orientation of each representative graph in a modular decomposition of G .

Suppose we are given transitive orientations of each representative graph in the modular decompositions of G . There is a transitive orientation T' of G defined as follows: for each pair of vertices x, y of G , let M be the smallest module in the modular decomposition of G which contains both x and y ; i.e., the first common ancestor in the modular decomposition tree. Let X be the vertex in the representative graph of M which corresponds to the submodule of M that contains x , and let Y be the vertex in the representative graph of M which corresponds to the submodule that contains y . If there is an edge (x, y) in G , direct the edge from x to y in T' if and only if the edge (X, Y) is directed from X to Y in the representative graph of M .

We will show that T' is transitive. Let u, v and w be any three vertices of G . Let M be the smallest module of the modular decomposition which contains at least two of $\{u, v, w\}$, and let M' be the smallest module of the modular decomposition which contains all three of $\{u, v, w\}$. Suppose M contains exactly two of these vertices, let us say u and v , but M does not contain w . Since u and v are in a module which does not contain w , both u and v are in the same submodule of M' . There is an edge from (to) w to (from) v in T' if and only if there is an edge from (to) w to (from) u in T' . Therefore, there cannot be a transitivity violation involving u, v and w . If M contains all three vertices, then $M = M'$, and any transitivity violation between u, v and w would imply a transitivity violation in the representative graph of M' , as edges between these vertices are directed in the same direction as the corresponding vertices in the representative graphs. Since there cannot be any transitivity violations, T' must be transitive. \square

It is important to note that the procedure described in Lemma 1 to construct a transitive orientation of G from transitive orientations of every representative graph in the modular decomposition can be implemented to run in $O(n^2)$ time. This can be done by attaching to each vertex in every representative graph a list of all vertices in the corresponding module. Each edge (x, y) in each representative graph is examined, and edges between the two submodules are oriented in the appropriate direction. An edge between x and y can only be directed once, while examining the representative graph of the smallest module containing x and y in the modular decomposition, so the whole procedure takes $O(n^2)$ time.

The orientation algorithm works by orienting each representative graph in the modular decomposition. Vertices and parallel modules are handled trivially; there are no edges to orient. Series modules are also easy to orient; choose any permutation of the vertex set of the representative graph, treat it as a total order and orient the edges

in the appropriate directions. The orientation of neighborhood modules is more difficult, and is covered in the next section.

Orientation of neighborhood modules. The only remaining problem in the orientation of comparability graphs is in the orientation of a neighborhood module in which all submodules can be thought of as single vertices. A neighborhood module M with no submodules has a single transitive orientation, up to reversal of all edges, as will be seen later; this is what makes the modular decomposition useful for the orientation problem. The algorithm works in two stages; first, the algorithm finds a subset S of M and orients all edges between S and $M - S$ in directions which are consistent with a transitive orientation of M . Lemma 2 proves that a transitive orientation of M can be constructed from this in $O(k^2)$ time, where k is the number of vertices in M .

LEMMA 2. *Let M be a neighborhood module of comparability graph with k vertices and only trivial submodules. Let S be a subset of M such that every edge between S and $M - S$ is oriented in directions compatible with a transitive orientation of M . M can be transitively oriented in $O(k^2)$ time.*

Proof. Partition M into two blocks, S and $M - S$. We will refine these blocks, maintaining the property that all edges between blocks are oriented in a manner compatible with a transitive orientation of M , until all blocks consist of a single vertex. When the algorithm halts, between each two blocks of the partition are either no edges or all possible edges, implying that the blocks are modules; therefore, each block must be trivial when the algorithm halts, and all edges must be oriented. The algorithm for refinement is presented in more detail in Appendix 1; we will present the general idea in this section.

A vertex v splits a block B if there are vertices $b_1, b_2 \in B$ such that $(v, b_1) \in E$, and $(v, b_2) \notin E$.

At any time during this algorithm, a vertex v has attempted to split blocks containing some set of vertices, and has not attempted to split blocks containing some other set of vertices. For each vertex v , M can be divided into three sets:

SAMEBLOCK (v). The vertices currently in the same block as v . The algorithm does not attempt to split the block it is in.

ALREADYSPPLIT (v). The vertices in blocks which have been split sometime previously in the algorithm. These blocks would not be split by v , since v has already split the block into a subblock of vertices related to v , and a subblock of vertices unrelated to v .

ACTIVE (v). The vertices which are not in the same block as v , and not in blocks which have been split already by v . The algorithm will take some vertex v which has active vertices, and try to split some block which contains a vertex in ACTIVE (v).

Originally, for each vertex $s \in S$, SAMEBLOCK (s) = S , ALREADYSPPLIT (s) = \emptyset , and ACTIVE (s) = $M - S$. Similarly, for any $m \in M - S$, SAMEBLOCK (m) = $M - S$, ALREADYSPPLIT (m) = \emptyset , and ACTIVE (m) = S . All vertices for which ACTIVE (v) $\neq \emptyset$ are kept in a list called SPLITTER; originally, all vertices in M are placed in SPLITTER.

Any vertex v from SPLITTER is chosen, and is used to split a block which contains vertices in ACTIVE (v). The algorithm chooses any vertex in ACTIVE (v) and finds its block B . We divide B into $B_1 = \{b: b \in B, (b, v) \in E\}$ and $B_2 = \{b: b \in B, (b, v) \notin E\}$. It is easy to orient all edges between B_1 and B_2 ; edges from v to B_1 are already oriented, so an edge $(b_1 \in B_1, b_2 \in B_2)$ is oriented from b_1 to b_2 if and only if the edge (b_1, v) is oriented from b_1 to v . If the edge were oriented in the other direction, there would be a path from b_2 to v , while there is no edge between b_2 and v . Therefore, we maintain

the property that all edges between blocks are oriented consistently with a transitive orientation of M .

The algorithm must keep track of which vertices are active for each other vertex. This requires the following operations after the block B is split by v . All vertices in B are moved from ACTIVE(v) to ALREADYSPPLIT(v). If ACTIVE(v) = \emptyset , v is removed from SPLITTER. If B_1 and B_2 are both nonempty, the vertices in B_1 are moved from SAMEBLOCK(b_2) to ACTIVE(b_2) for each vertex $b_2 \in B_2$, and the vertices in B_2 are moved from SAMEBLOCK(b_1) to ACTIVE(b_1) for each $b_1 \in B_1$. Any vertex w which had an active set = \emptyset and has some vertex inserted into ACTIVE(w) is put into SPLITTER.

The algorithm continues choosing a vertex v from SPLITTER and splits a block containing vertices in ACTIVE(v) until SPLITTER becomes empty. Note that for a vertex v , any other vertex can only move from SAMEBLOCK(v) to ACTIVE(v), or from ACTIVE(v) to ALREADYSPPLIT(v).

The adjacency matrix entry (x, y) is examined at most three times. The first time (x, y) can be examined is when the block containing x and y is split into a block containing x , and another containing y , and at this time x is moved from SAMEBLOCK(y) to ACTIVE(y) and y is moved from SAMEBLOCK(x) to ACTIVE(x). The entry can also be examined once when the block containing x is split by y ; at this time, x is moved from ACTIVE(y) to ALREADYSPPLIT(y). Finally, the entry can be examined when the block containing y is split by x , and y is moved from ACTIVE(x) to ALREADYSPPLIT(x). Since each entry is examined a constant number of times, this procedure takes $O(k^2)$ time.

Finally, we must prove that every block after the call REFINE($S, M - S$) consists of a single vertex; this will show that all edges have been oriented. Suppose a block B has more than one vertex, and that there are no vertices in SPLITTER. This means that for every vertex in $M - B$, B is a subset of ALREADYSPPLIT(v). Thus, no vertex outside B splits B , so B is a submodule. Since M has no nontrivial submodules, B cannot exist. \square

The problem has been reduced to finding a set S such that all edges between S and $M - S$ are oriented in directions consistent with a transitive orientation of M . The algorithm creates a "skeleton" of the graph, where each vertex in $M - S$ has an oriented edge to some vertex x which is unrelated to every vertex in S . This makes it easy to orient edges between S and $M - S$, as the orientation must not allow a path between x and the vertex in S .

We start by choosing an initial edge to orient. Let a and z be any pair of unrelated vertices in M . There must be some vertex which has an edge to exactly one of $\{a, z\}$, or a and z would form a submodule. Without loss of generality, assume there is an edge (a, b) such that z is unrelated to both a and b . The edge (a, b) is oriented in an arbitrary direction. We then divide M into three sets:

U . Initially, the subset of $M - \{a, b\}$ which is unrelated to both a and b . In general, this is the set of vertices which are unrelated to all vertices which have already been examined.

R . Initially, the subset of $M - \{a, b\}$ which is related to both a and b . In general, this is the set of vertices which are related to all vertices which have already been examined.

B . Initially, $\{a, b\} \cup$ the subset of $M - \{a, b\}$ which is related to exactly one of a, b . In general, this is the set of vertices which are related to some, but not all, of the vertices which have already been examined.

S will be a subset of U , and we orient an edge from a vertex when it is brought

into B . This edge will be oriented to a vertex which has already been examined, so the oriented edge will always be to a vertex unrelated to every vertex in S .

We orient the edges between any initial vertex v of $B \neq a$ or b and that vertex of $\{a, b\}$ it is related to in the direction which does not allow a path between v and that vertex of $\{a, b\}$ which is unrelated to v . Vertices in B are then put in a first in, first out queue; each vertex is examined in order to bring other vertices into B . We say a vertex is examined when it is removed from the queue, and a and b are presumed to be removed from the queue at the beginning of the algorithm.

While the set U is not empty, i.e., there is still some vertex unrelated to all vertices examined thus far, we perform the following operations. The vertex v at the front of B 's queue is removed from the queue. We note that there is always an oriented edge between v and some vertex x such that x has already been examined.

We try to use v to bring new vertices into B . We first look at the relationships between v and the set R ; if any vertex $w \in R$ is unrelated to v , we bring w into B . There is an oriented edge between v and x , and x was examined before v and must be related to w . The edge between x and w can be oriented in the direction which does not create a path between v and w .

We then look at the relationships between v and U . If there is any vertex $w \in U$ which is related to v , we bring w into B , unless there is no vertex in U which is unrelated to v ; in the latter case, U becomes the set S . There is an oriented edge between x and v , and x was examined before v and must be unrelated to w . The edge between v and w can be oriented in the direction which does not create a path between x and w .

Eventually, we find a set S , which would have been the last set of vertices of U to be brought into B . Note that every vertex in B at this time has an oriented edge to some vertex which has already been examined, and therefore must be unrelated to every vertex in S . The only remaining problem is to orient an edge from each vertex in R to a vertex which is unrelated to every vertex in S .

We call the vertex which removed the set S from U $e(S)$. We note that $e(S)$ cannot be a or b , since we chose our initial edge (a, b) so that some vertex z was unrelated to both a and b . Therefore, there must be some vertex which was examined before $e(S)$ which was unrelated to $e(S)$, as well as some vertex which was examined before $e(S)$ which has an oriented edge to $e(S)$. Let $u(e(S))$ be a vertex examined before $e(S)$ which is unrelated to $e(S)$, and let $o(e(S))$ be the vertex which was removed from the queue before $e(S)$, and has an oriented edge to $e(S)$. Note that we can orient all edges between S and $e(S)$ so as to avoid any path between S and $o(e(S))$.

The vertices in S are now put on the queue along with the remaining vertices of B . We continue as before, examining the vertex v at the front of the queue, and bringing into B any vertices of R which are unrelated to v . If v is not in S , we can orient an edge from any vertex w which is brought into B as we did before; there is an oriented edge between v and a vertex x examined before $e(S)$, and we can orient the edge between x and w so that there is no path between v and w . If v is in S , we must use a different procedure, which is described in the following paragraph.

Let v be a vertex in S , and let w be a vertex which has no edge to v , but has edges to all vertices examined before v . We want to orient an edge between w and some vertex which was examined before $e(S)$; this will allow us to orient all edges between w and S . We can orient the edge between $e(S)$ and v so that there is no path between v and $o(e(S))$. Once this edge has been oriented, we can orient the edge between w and $e(S)$ so that there is no path between v and w . Finally, we orient the edge between

$u(e(S))$ and w so that there is no path between $u(e(S))$ and $e(S)$. See Fig. 3 for an example of this orientation.

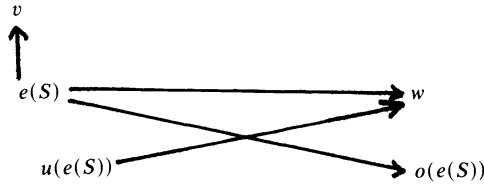


FIG. 3

After this procedure, there will be an oriented edge between each vertex in $M - S$ and a vertex v which was examined before $e(S)$; v must be unrelated to every vertex in S . This allows us to orient all edges between S and $M - S$, and we can use the procedure REFINE of Appendix 1, described in Lemma 2, to develop a transitive orientation for the module. Given the orientation algorithm for a module, it is easy to orient the graph, as was described in the previous section. Since the orientation of the module can be done in $O(k^2)$ time, the entire graph can be oriented in $O(n^2)$ time.

A version of this procedure written in pseudocode is given in Appendix 2. Figure 4 shows a sample module M , and shows the skeleton derived from M using this algorithm. In the example, $S = \{s\}$, and vertices are removed from the queue in the following order with the vertex that brought each into B in parenthesis: $a(b)$, $b(a)$, $d(a)$, $e(b)$, $c(d)$, $s(e)$, $r(c)$.

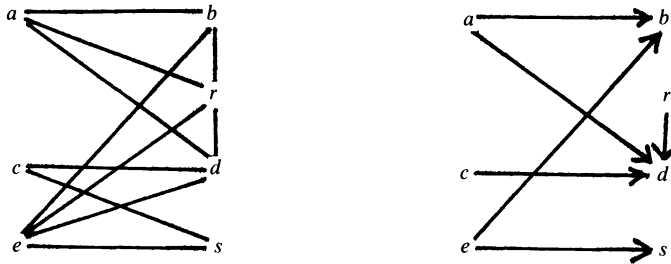


FIG. 4

Recognizing comparability graphs. The $O(n^2)$ comparability graph orientation algorithm does not automatically imply the existence of an $O(n^2)$ comparability graph recognition algorithm. The orientation algorithm assigns an orientation to any undirected graph G , and says that this orientation is transitive if and only if G is a comparability graph. The best way known to the author to recognize a transitive graph is to compare the graph to its transitive closure. The transitive closure can be computed in slightly less than $O(n^{2.49+})$ time [CW]; this is the dominant cost in a comparability graph recognition algorithm.

Using the reduction described in the previous paragraph, a new algorithm for recognizing transitive graphs in time $f(n)$ would allow us to recognize comparability graphs in $O(\max \{f(n), n^2\})$ time. In this section, we will show that the reduction holds in both directions, i.e., a faster comparability graph recognition algorithm would give a faster algorithm for recognizing transitive graphs. More precisely, any algorithm for recognizing comparability graphs in time $f(n)$ would allow us to recognize transitive graphs in $O(\max \{f(n), m\})$ time.

Let $G = (V, E)$ be a digraph to be tested for transitivity. Assume $|E| > 0$. Construct G' , an undirected graph whose vertices are three copies, V_1 , V_2 and V_3 , of V , and two distinguished vertices s and t . The edges of G' will be the edges (s, v_1) and (v_3, t) for each $v \in V$, and the edges (v_1, w_2) , (v_1, w_3) and (v_2, w_3) for each edge $(v, w) \in E$. We will prove that G is transitive if and only if G' is a comparability graph.

Suppose that G is transitive. G' can be directed in the following fashion: Each edge (s, v) is directed from v to s , each edge (t, v) is directed from t to v , all edges between V_1 and V_2 are directed from V_1 to V_2 , all edges between V_1 and V_3 are directed from V_1 to V_3 , and all edges between V_2 and V_3 are directed from V_2 to V_3 . Suppose there are edges from x' to y' and from y' to z' in the directed version of G' . Any vertex v' which has edges both entering and leaving v' must be in V_2 ; therefore y' must be in V_2 . By the orientation of G' , x' must be a vertex of V_1 such that $(x, y) \in E$, and z' must be a vertex of V_3 such that $(y, z) \in E$. Since G is transitive, $(x, z) \in E$, and there will be an edge directed from x' to z' in the directed version of G' . There cannot be any transitivity violations in this orientation of G' , so G' is a comparability graph.

Suppose that G' is a comparability graph. Let v' be any vertex $\in V_1$. There must be some transitive orientation T of G' in which the edge (v', s) is directed from v' to s , since the reversal of a transitive orientation is also transitive. There cannot be a path in T between v' and any vertices $\in V_1$, so all edges between s and V_1 must be directed from V_1 to s in T . No vertex in V_2 or V_3 is adjacent to s in G' , so all edges between V_1 and V_2 must be directed from V_1 to V_2 in T , and all edges between V_1 and V_3 must be directed from V_1 to V_3 . Since $|E| > 0$, there must be some $w' \in V_3$ such that there is an edge directed from a vertex in V_1 to w' in T . There cannot be a path between t and a vertex in V_1 in T , so the edge (w', t) must be directed from t to w' . There cannot be a path in T between w' and any vertices $\in V_3$, so all edges between t and V_3 must be directed from t to V_3 in T . There cannot be a path between any vertex $\in V_2$ and t in T , so all edges between V_2 and V_3 must be directed from V_2 to V_3 in T . Consider any three vertices x, y, z of V . If the edges (x, y) and (y, z) are in E , there must be edges (x_1, y_2) and (y_2, z_3) in T . Since T is a transitive orientation of G' , there must be an edge (x_1, z_3) in T ; this can only happen if there is an edge (x, z) in E . If (x, y) and (y, z) are in E , (x, z) must be in E , so G is transitive.

It should be noted that comparability graph recognition has been used as a subroutine in recognition algorithms for other classes of graphs. Comparability graph recognition is used as part of the standard algorithm for permutation graph recognition (for example, see [GOL2]); a new permutation graph recognition algorithm is described in the next section. Recognition of comparability graphs is also the slowest part of the best known algorithm for recognizing circular permutation graphs [RU], so the complexity of circular permutation graph recognition is reduced from $O(n^3)$ to $O(n^{2.49+})$ using the algorithm described in this paper.

Permutation graphs. There is a more direct result using the orientation algorithm in the recognition of permutation graphs. A permutation graph is an undirected graph which can be represented by two permutations P, Q of the vertex set such that there is an edge between u and v if and only if one of the following two conditions is true:

- (1) u precedes v in both P and Q .
- (2) v precedes u in both P and Q .

Permutation graphs have been studied by Pnueli, Even and Lempel [PEL], Colbourne [COL], and others. Golumbic [GOL2] demonstrates that permutation graphs can be used to model a number of problems dealing with memory management and other areas. Previous algorithms to recognize permutation graphs are based on the

characterization that a graph is a permutation graph if and only if the graph and its complement are comparability graphs [PEL]. In [DM], there is a proof that a transitive graph is a two dimensional partial order if and only if the complement graph is a comparability graph. Therefore, a transitive orientation of an undirected graph G is a two dimensional partial order if and only if G is a permutation graph.

In [SV], an $O(n^2)$ algorithm is presented for recognizing two dimensional partial orders. From the argument above, the same algorithm can be used to recognize permutation graphs in $O(n^2)$ time. The algorithm for recognition of two dimensional partial orders works by taking a pair of unrelated vertices a and b , and setting a in front of b in one list. This forces a particular ordering of the vertices in a single neighborhood module, using a procedure similar to the REFINER algorithm in Appendix 1. In fact, the transitive orientation algorithm in this paper can also be used directly to solve the permutation graph recognition problem in $O(n^2)$ time, as is indicated by Golumbic [GOL2, Ex. 8, p. 169].

The isomorphism algorithm for two dimensional partial orders which is presented in [SP] can also be adapted to permutation graphs, to get an $O(n^2)$ isomorphism algorithm. The fastest previous solution to this problem is an $O(n^3)$ algorithm presented in [COL].

The new algorithm makes a postorder traversal of the modular decomposition of the graphs which are being tested for isomorphism, determining for each module M which modules in the other decomposition are isomorphic to M . Assuming that all isomorphisms of the children of M are known, it is easy to determine which modules are isomorphic to a series or parallel module M ; a module of the same type is isomorphic if we can pair off isomorphic children, and all children of both modules are paired. For neighborhood modules, we rely on the fact that there are only two permutations of the vertex set which represent the module; this reduces the problem to pairing off children in the same spot in the permutations.

Summary and open problems. We have presented algorithms which improve the time complexity of a number of problems on comparability graphs; transitive orientation of comparability graphs, recognition of comparability graphs, recognition of permutation graphs and isomorphism of permutation graphs. These algorithms run in $O(n^2)$ time, with the exception of the comparability graph recognition algorithm, which runs in $O(n^{2.49+})$ time.

We have proved that any improvements in algorithms for recognizing transitive graphs will give us faster algorithms for recognizing comparability graphs, and vice-versa. It is not at all clear that these problems are as hard as computing the transitive closure of the graph. Further research is needed to determine whether the recognition problems are easier than the computation of transitive closure, or if they have the same complexity.

In addition, other uses of the modular decomposition of a graph may be interesting. Modular decomposition seems to be a natural way of decomposing transitive graphs, and can be used both as an aid in developing algorithms, and as a method for storing large transitive graphs.

Appendix 1. This section gives a detailed specification of the partition refinement scheme outlined in Lemma 2. The input to this procedure is a pair of sets of vertices B_1, B_2 such that every edge between B_1 and B_2 is oriented in a manner consistent with a transitive orientation of $B_1 \cup B_2$. If $B_1 \cup B_2$ is the vertex set of a representative graph of a neighborhood module in a comparability graph, the output of the procedure will be a transitive orientation of $B_1 \cup B_2$.

```

REFINE ( $B_1, B_2$ );
  for each  $b_1 \in B_1$  do begin
    SAMEBLOCK ( $b_1$ ) :=  $B_1$ ;
    ACTIVE ( $b_1$ ) :=  $B_2$ ;
    ALREADYSPPLIT ( $b_1$ ) :=  $\emptyset$ ;
  end;
  for each  $b_2 \in B_2$  do begin
    SAMEBLOCK ( $b_2$ ) :=  $B_2$ ;
    ACTIVE ( $b_2$ ) :=  $B_1$ ;
    ALREADYSPPLIT ( $b_2$ ) :=  $\emptyset$ ;
  end;
  SPLITTER :=  $B_1 \cup B_2$ ;
  while SPLITTER  $\neq \emptyset$  do begin
     $v$  := any vertex from SPLITTER;
    splittee := any vertex in ACTIVE ( $v$ );
     $B$  := the block which contains splittee;
     $B_R$  :=  $\{b \in B : (b, v) \in E\}$ ;
     $B_U$  :=  $\{b \in B : (b, v) \notin E\}$ ;
    for each  $x \in B_U$  do
      for each  $y \in B_R$  do
        if  $(x, y) \in E$  then
          orient  $(x, y)$  so there is no path between  $x$  and  $v$  through  $y$ ;
    for each  $x \in B_U$  do begin
      SAMEBLOCK ( $x$ ) := SAMEBLOCK ( $x$ ) -  $B_R$ ;
      if ACTIVE ( $x$ ) =  $\emptyset$  and  $B_R \neq \emptyset$  then
        add  $x$  to SPLITTER;
      ACTIVE ( $x$ ) := ACTIVE ( $x$ )  $\cup B_R$ ;
    end;
    for each  $x \in B_R$  do begin
      SAMEBLOCK ( $x$ ) := SAMEBLOCK ( $x$ ) -  $B_U$ ;
      if ACTIVE ( $x$ ) =  $\emptyset$  and  $B_U \neq \emptyset$  then
        add  $x$  to SPLITTER;
      ACTIVE ( $x$ ) := ACTIVE ( $x$ )  $\cup B_U$ ;
    end;
    ACTIVE ( $v$ ) := ACTIVE ( $v$ ) -  $B$ ;
    ALREADYSPPLIT ( $v$ ) := ALREADYSPPLIT ( $v$ )  $\cup B$ ;
    if ACTIVE ( $v$ ) =  $\emptyset$  then SPLITTER := SPLITTER -  $\{v\}$ ;
  end;

```

Appendix 2. The following algorithm finds a set S and orients all edges between $M - S$ in directions consistent with a transitive orientation of M . Input to the algorithm is a neighborhood module M of the modular decomposition, and a pair of vertices a, b such that a and b are related, and some vertex is unrelated to both a and b .

```

ALGORITHM SKELETON ( $a, b$ );
  for each  $v \in M$  do directed ( $v$ ) :=  $\emptyset$ ;
  orient edge  $(a, b)$  as  $a \rightarrow b$ ;
  directed ( $a$ ) :=  $b$ ;
  directed ( $b$ ) :=  $a$ ;
   $U$  :=  $\{m \in M : (m, a) \notin E, (m, b) \notin E\}$ ;
   $R$  :=  $\{m \in M : (m, a) \in E, (m, b) \in E\}$ 

```

```

 $N(b) := \{m \in M - R : (m, b) \in E\};$ 
 $N(a) := \{m \in M - R : (m, a) \in E\};$ 
for each  $m \in N(b)$  do begin
  orient edge  $(b, m)$  as  $m \rightarrow b$ ;
  directed  $(m) := b$ ;
end;
for each  $m \in N(a)$  do begin
  orient edge  $(a, m)$  as  $a \rightarrow m$ ;
  directed  $(m) := a$ ;
end;
QUEUE :=  $N(a) \cup N(b)$ ;
while  $U \neq \emptyset$  do begin
   $v :=$  vertex at front of QUEUE;
  QUEUE := QUEUE -  $\{v\}$ ;
  for each vertex  $w \in R$  do
    if  $(v, w) \notin E$  do begin
       $R := R - \{w\}$ ;
      orient  $(w, \text{directed}(v))$  so that there is no path between  $w$  and  $v$ ;
      directed  $(w) := \text{directed}(v)$ ;
      append  $w$  to QUEUE;
    end;
   $S := \emptyset$ ;
  for each vertex  $w \in U$  do
    if  $(v, w) \in E$  do begin
       $U := U - \{w\}$ ;
      orient  $(w, v)$  so that there is no path between  $w$  and directed  $(v)$ ;
      directed  $(w) := v$ ;
      append  $w$  to QUEUE;
       $S := S \cup \{w\}$ ;
    end;
  end;
   $e(S) := v$ ;
  orient all edges between  $e(S)$  and  $S$  so that there is no path between  $S$  and directed  $(e(S))$ ;
   $u(e(S)) :=$  any  $x$  s.t.  $(x, e(s)) \notin E$ ,  $x$  already removed from QUEUE
  while  $R \neq \emptyset$  do begin
     $v :=$  vertex at front of QUEUE;
    QUEUE := QUEUE -  $\{v\}$ ;
    if  $v \notin S$  do begin
      for each vertex  $w \in W$  do
        if  $(v, w) \notin E$  do begin
           $R := R - \{w\}$ ;
          orient  $(w, \text{directed}(v))$  so that there is no path between  $w$  and  $v$ ;
          directed  $(w) := \text{directed}(v)$ ;
          append  $w$  to QUEUE;
        end;
      end;
    end;
    if  $v \in S$  do begin
      for each vertex  $w \in R$  do
        if  $(v, w) \notin E$  do begin

```

```

    R := R - {w};
    orient (w, e(S)) so that there is no path between v and w (through e(S));
    orient (u(e(S)), w) so that there is no path between u(e(S)) and e(S)
      (through w);
    directed (w) := u(e(S));
    append w to QUEUE;
  end;
end;
end;
for each vertex v ∈ S do
  for each vertex w ∈ M - S do
    if (v, w) ∈ E then
      orient (v, w) so there is no path between v and directed (w);

```

Acknowledgments. I would like to thank Jacobo Valdes for his help in the development of the modular decomposition algorithm, and his advice and support in this research. I am also indebted to the referees for improving the structure of this paper, and particularly to one of the referees for pointing out a much simpler reduction of transitive graph recognition to comparability graph recognition than was contained in an earlier draft of this paper.

REFERENCES

- [AHU] A. AHO, J. HOPCROFT AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
- [COL] C. J. COLBOURNE, *On testing isomorphism of permutation graphs*, Networks, 11 (1981), pp. 13-21.
- [CW] D. COPPERSMITH AND S. WINOGRAD, *On the asymptotic complexity of matrix multiplication*, this Journal, 11 (1982), pp. 472-492.
- [DM] B. DUSCHNIK AND E. W. MILLER, *Partially ordered sets*, Amer. J. Math., 63 (1941), pp. 600-610.
- [GH] GILMORE AND HOFFMAN, *A characterization of comparability and interval graphs*, Canad. J. Math., 16 (1964), pp. 539-548.
- [GOL] M. GOLUBIC, *The complexity of comparability graph recognition and coloring*, Computing, 18 (1977), pp. 199-208.
- [GOL2] ———, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.
- [G-H] GOUILA-HOURI, *Caracterisation des graphes nonorientés dont on peut orienter les arêtes à obtenir le graphe d'une relation d'ordre*, CR Acad. Sci. Paris, 254 (1962), pp. 1370-1371.
- [MS] J. MULLER AND J. SPINRAD, *On-line modular decomposition*, Georgia Institute of Technology, Tech. Rep. GIT-ICS-84/11.
- [PEL] A. PNUELI, S. EVEN AND A. LEMPEL, *Transitive orientation of graphs and identification of permutation graphs*, Canad. J. Math., 23 (1971), pp. 160-175.
- [RU] D. ROTEM AND J. URRUTIA, *Circular permutation graphs*, Networks, 12 (1982), pp. 429-437.
- [SP] J. SPINRAD, *Two dimensional partial orders*, Ph.D. Thesis, Princeton Univ., Princeton, NJ, 1982.
- [SV] J. SPINRAD AND J. VALDES, *Recognition and isomorphism of two dimensional partial orders*, 10th Colloquium on Automata, Languages and Programming, in Lecture Notes in Computer Science 154, Springer-Verlag, Berlin, 1983, pp. 676-686.
- [VTL] J. VALDES, R. TARJAN AND E. LAWLER, *Recognition of series-parallel digraphs*, this Journal, 11 (1982), pp. 298-314.

THE IMPLICATION PROBLEM FOR FUNCTIONAL AND INCLUSION DEPENDENCIES IS UNDECIDABLE*

ASHOK K. CHANDRA† AND MOSHE Y. VARDI‡

Abstract. The implication problem for a class of dependencies is the following: given a finite set of dependencies, determine if they logically imply another dependency. We show that the implication problem is undecidable for the class of functional and inclusion dependencies. This holds true even if the inclusion dependencies are restricted to be binary. It may be noted that the implication problem is known to be decidable for functional and unary inclusion dependencies and also for inclusion dependencies without functional dependencies.

Key words. axiomatization, data base dependency, functional dependency, inclusion dependency, recursive inseparability, relational data base, undecidability

1. Introduction. Functional and inclusion dependencies are the most fundamental database integrity constraints, and they are used in essentially all data models. Their interaction has recently been investigated in several papers [CFP], [FV], [JK].

In order to utilize dependencies in the database design process one needs to be able to test for logical implication, i.e., does a set of dependencies logically imply another dependency [Be]. The implication problem is one of the prominent issues in dependency theory.

It is known that, when only functional dependencies are given or when only inclusion dependencies are given, the implication problem is decidable [BB], [CFP]. In this note we show that when functional and inclusion dependencies are considered simultaneously the problem becomes undecidable. (This result was also obtained independently by Mitchel [M2].) We also show that implication for functional and inclusion dependencies is reducible to implication for functional and binary inclusion dependencies, and we study the consequences of the undecidability result on the issue of obtaining an effective axiomatization for implication.

2. Definitions. A *relation scheme* U is a finite sequence C_1, \dots, C_n of *attributes*, which, intuitively, serve as column headings. A *tuple* t over U is a sequence $\langle c_1, \dots, c_n \rangle$ of the same length as U . A *relation* R over U is a set (not necessarily finite) of tuples over U . U is called the *scheme* of R . If C_{i_1}, \dots, C_{i_m} is a sequence of members of U , then

$$t[C_{i_1}, \dots, C_{i_k}] = \langle c_{i_1}, \dots, c_{i_k} \rangle,$$

and

$$R[C_{i_1}, \dots, C_{i_k}] = \{t[C_{i_1}, \dots, C_{i_k}]: t \in R\}.$$

A *functional dependency* [Co] (abbr. *fd*) is a statement $A_1, \dots, A_k \rightarrow B_1, \dots, B_l$, where $k, l \geq 0$ and the A 's and B 's are attributes. A relation R whose scheme includes $A_1, \dots, A_k, B_1, \dots, B_l$ satisfies this fd if, for all tuples $s, t \in R$, if $s[A_1, \dots, A_k] = t[A_1, \dots, A_k]$ then $s[B_1, \dots, B_l] = t[B_1, \dots, B_l]$.

* Received by the editors December 22, 1983.

† IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

‡ Department of Computer Science, Stanford University, Stanford, California 94305. The research of this author was supported by a Weizmann Post-Doctoral Fellowship and by the Air Force Office of Scientific Research under grant 80-0212.

An *inclusion dependency* [Fa] (abbr. *ind*) is a statement $A_1, \dots, A_k \subseteq B_1, \dots, B_k$, where the A 's and B 's are attributes. A relation R whose scheme includes $A_1, \dots, A_k, B_1, \dots, B_k$ satisfies this ind if $R[A_1, \dots, A_k] \subseteq R[B_1, \dots, B_k]$.

A relation R satisfies a set D of dependencies if it satisfies all dependencies in D . A set D is said to *imply* a dependency d , denoted $D \models d$, if d is satisfied by all relations that satisfy D . D is said to *finitely imply* d , denoted $D \models_f d$, if d is satisfied by all finite relations that satisfy D . Clearly, if $D \models d$ then also $D \models_f d$. But it is shown in [CFP] that the converse is not always the case. From a practical point of view, finite implication is the more interesting notion.

The *implication problem* for fd's and ind's is to decide, given a finite set D of fd's and ind's and an fd or an ind d , whether $D \models d$. The *finite implication problem* is to decide whether $D \models_f d$. In the following section we show that both problems are undecidable.

3. The main result. The problem that we use for our undecidability proof is the word problem for (finite) monoids. Recall that a monoid is an algebra with an associative binary operation \cdot and a unit element 1 . Let Σ be an alphabet. Σ^* is the free monoid generated by Σ . Now let $E = \{\alpha_i = \beta_i : 1 \leq i \leq n\}$ be a finite set of equalities, and let e be an additional equality $\alpha = \beta$, where $\alpha, \beta, \alpha_i, \beta_i \in \Sigma^*$. We say that E (*finitely*) *implies* e , denoted $E \models e$ ($E \models_f e$), if for every (finite) monoid M and homomorphism $h: \Sigma^* \rightarrow M$, if $h(\alpha_i) = h(\beta_i)$ for $1 \leq i \leq n$, then also $h(\alpha) = h(\beta)$. The *word problem* for (finite) monoids is to decide, given E and e , whether $E \models e$ ($E \models_f e$). The word problem for monoids was shown to be undecidable in [Po] and the word problem for finite monoids was shown to be undecidable in [Gu]. These results also follow from a recursive inseparability result of [GL]. Two sets Φ and Ψ are *recursively inseparable* if there is no recursive set Π such that $\Phi \subseteq \Pi$ but Ψ and Π are disjoint. If Φ and Ψ are recursively inseparable then clearly they are both not recursive.

THEOREM 1. [GL] *The set $\{(E, e) : E \models e\}$ is recursively inseparable from the set $\{(E, e) : E \not\models_f e\}$, where E ranges over finite sets of equalities and e ranges over equalities.*

THEOREM 2. *The set $\{(D, d) : D \models d\}$ is recursively inseparable from the set $\{(D, d) : D \not\models_f d\}$, where D ranges over finite sets of fd's and ind's and d ranges over ind's.*

Proof. We reduce the word problem for (finite) monoids to (finite) implication of ind's.

Let Σ be our alphabet. Let $E = \{\alpha_i = \beta_i : 1 \leq i \leq n\}$ be a set of equalities, and let e be another equality $\alpha = \beta$. We call every prefix of $\alpha, \beta, \alpha_i,$ or β_i , for $1 \leq i \leq n$, a *prefix* (note that the null string λ is a prefix, and so is α , etc.). We use a relation R whose relation scheme has the following attributes:

- (1) γ , for each prefix γ ;
- (2) $X, Y,$ and Z ;
- (3) Ya , for each $a \in \Sigma$;
- (4) Za , for each $a \in \Sigma$.

Intuitively, the γ 's represent the corresponding elements of the monoid, X and Y represent arbitrary elements with Z as their product, and the Ya and Za represent multiplying by a from the right.

The set D consists of the following dependencies:

- (1) $\lambda, X, Y \rightarrow Z$, and $\lambda, Y \rightarrow Ya$, for each $a \in \Sigma$.
- (2) $\lambda, \lambda \subseteq \lambda, Y$.
- (3) $\lambda, \gamma, \gamma a \subseteq \lambda, Y, Ya$, for each $a \in \Sigma$ and pair of prefixes $\gamma, \gamma a$.
- (4) $\lambda, Z, Za \subseteq \lambda, Y, Ya$, for each $a \in \Sigma$.
- (5) $\lambda, X, Ya, Za \subseteq \lambda, X, Y, Z$, for each $a \in \Sigma$.

(6) $\lambda, Y, \lambda, Y \subseteq \lambda, X, Y, Z$.

(7) $\lambda, \alpha_i \subseteq \lambda, \beta_i$.

The above dependencies ensure that the attributes behave according to the intended meaning. Note that D does not guarantee uniqueness of the elements that correspond to the prefixes. We could have added the fd $\emptyset \rightarrow \lambda$, but we did not want to use fd's with empty left-hand side, and we are willing to pay for it with some complication in the reduction.

Finally, the dependency d is $\alpha \subseteq \beta$.

CLAIM. $D \models d$ (resp. $D \models_f d$) iff $E \models e$ (resp. $E \models_f e$).

Proof of claim.

Only-if part. We show that $E \not\models e$ (resp. $E \not\models_f e$) entails $D \not\models d$ (resp. $D \not\models_f d$).

Suppose that $E \not\models e$. Then there is a homomorphism h on Σ^* such that $h(\alpha_i) = h(\beta_i)$ but $h(\alpha) \neq h(\beta)$. For a pair x, y of elements in Σ^* we define a tuple $t_{x,y}$ by:

$t_{x,y}[\gamma] = h(\gamma)$, for each prefix γ ,

$t_{x,y}[X] = h(x)$,

$t_{x,y}[Y] = h(y)$,

$t_{x,y}[Ya] = h(ya)$, for each $a \in \Sigma$,

$t_{x,y}[Z] = h(xy)$, and

$t_{x,y}[Za] = h(xya)$, for each $a \in \Sigma$.

Let now $R = \{t_{x,y} : x, y \in \Sigma^*\}$. We leave it to the reader to show that R satisfies D but not d . If $E \not\models_f e$, then in addition we can assume that $\{h(x) : x \in \Sigma^*\}$ is finite, so R is also finite.

If part. Assume that $E \models e$, and let R be a relation satisfying D . Let $s \in R$. We have to show that $s[\alpha] \in R[\beta]$. We now define a homomorphism h of Σ^* into the domain of the relation R , such that $h(\lambda) = s[\lambda]$. For an element $y \in \Sigma^*$, we define the element $h(y)$ and show that $\langle h(\lambda), h(y) \rangle \in R[\lambda, Y]$ inductively as follows:

(1) $h(\lambda)$ is $s[\lambda]$. Note that $\langle h(\lambda), h(\lambda) \rangle \in R[\lambda, Y]$ by the ind $\lambda, \lambda \subseteq \lambda, Y$.

(2) Suppose that we have defined $h(y)$ and showed that $\langle h(\lambda), h(y) \rangle \in R[\lambda, Y]$, and let $a \in \Sigma$. Define $h(ya)$ to be the unique element $t[Ya]$ for the tuple t such that $t[\lambda, Y] = \langle h(\lambda), h(y) \rangle$ (uniqueness is ensured by the fd $\lambda, Y \rightarrow Ya$). $\langle h(\lambda), h(ya) \rangle \in R[\lambda, Y]$ by the ind $\lambda, X, Ya, Za \subseteq \lambda, X, Y, Z$.

Next we show that for all $x, y \in \Sigma^*$, we have that $\langle h(\lambda), h(x), h(y), h(xy) \rangle \in R[\lambda, X, Y, Z]$. This is shown by induction on y . We have already shown that $\langle h(\lambda), h(x) \rangle \in R[\lambda, Y]$, so by the ind $\lambda, Y, \lambda, Y \subseteq \lambda, X, Y, Z$ we have $\langle h(\lambda), h(x), h(\lambda), h(x) \rangle \in R[\lambda, X, Y, Z]$. Assume now that $\langle h(\lambda), h(x), h(y), h(xy) \rangle \in R[\lambda, X, Y, Z]$. Let $t \in R$ be such that $t[\lambda, X, Y, Z] = \langle h(\lambda), h(x), h(y), h(xy) \rangle$. Then by definition $t[Ya] = h(ya)$. Also by the ind $\lambda, Z, Za \subseteq \lambda, Y, Ya$ we have that $t[Za] = h(xya)$. By the ind $\lambda, X, Ya, Za \subseteq \lambda, X, Y, Z$ we have that

$$\langle h(\lambda), h(x), h(ya), h(xya) \rangle \in R[\lambda, X, Y, Z].$$

We now define a binary operation \cdot on the set $\{h(x) : x \in \Sigma^*\}$. Let $x, y \in \Sigma^*$. The above argument shows that there is a tuple t in R such that $t[\lambda, X, Y] = \langle h(\lambda), h(x), h(y) \rangle$. We define $h(x) \cdot h(y)$ as $t[Z]$, which is unique by the fd $\lambda, X, Y \rightarrow Z$. Furthermore, we have shown that $t[Z]$ is just $h(xy)$, so it follows that h is a homomorphism on Σ^* .

Now, using the fd's $\lambda, Y \rightarrow Ya$ and the ind's $\lambda, \gamma, \gamma a \subseteq \lambda, Y, Ya$, it is easy to show by induction that R satisfies $\lambda \rightarrow \gamma$ and that $s[\gamma] = h(\gamma)$, for every prefix γ . In particular, $s[\alpha_i] = h(\alpha_i)$ and $s[\beta_i] = h(\beta_i)$. By the ind $\lambda, \alpha_i \subseteq \lambda, \beta_i$, there is a tuple $t \in R$ such that $t[\lambda, \beta_i] = \langle h(\lambda), h(\alpha_i) \rangle$. Since R satisfies $\lambda \rightarrow \beta_i$ it must be the case that $h(\alpha_i) = h(\beta_i)$.

Since $E \models e$, we have that $h(\alpha) = h(\beta)$, and since $s[\alpha] = h(\alpha)$ and $s[\beta] = h(\beta)$, it follows that $s[\alpha] \in R[\beta]$. For the finite case, if R is a finite relation, then $\{h(x) : x \in \Sigma^*\}$ is finite, and the rest of the argument is the same. \square

COROLLARY. *The sets $\{(D, d): D \models d\}$ and $\{(D, d): D \not\models_f d\}$, where D ranges over finite sets of fd's and ind's and d ranges over ind's are not recursive.*

The proof above showed that testing whether a set of fd's and ind's implies or finitely implies an ind is undecidable. How about testing (finite) implication of fd's? The next theorem shows that this is also undecidable.

THEOREM 3. *The set $\{(D, d): D \models d\}$ is recursively inseparable from the set $\{(D, d): D \not\models_f d\}$, where D ranges over finite sets of fd's and ind's and d ranges over fd's.*

Proof. We reduce the word problem for (finite) monoids to (finite) implication of fd's. Given Σ, E , and e we construct the set D of fd's and ind's as in the proof of Theorem 2 with the addition of the attributes α', β' , and Y' , and the dependencies $Y \rightarrow Y', \alpha, \alpha' \subseteq Y, Y'$, and $\beta, \beta' \subseteq Y, Y'$. The dependency d is $\alpha' \rightarrow \beta'$.

CLAIM. $D \models d$ (resp. $D \models_f d$) iff $E \models e$ (resp. $E \models_f e$).

Proof of Claim.

If part. Assume $E \models e$, and let R be a relation satisfying D . Let $s, t \in R$ such that $s[\alpha'] = t[\alpha']$. We have to show that $s[\beta'] = t[\beta']$. The If Part in the proof of Theorem 2 shows that $s[\beta] = s[\alpha]$ and $t[\alpha] = t[\beta]$. Because of the ind's $\alpha, \alpha' \subseteq Y, Y'$ and $\beta, \beta' \subseteq Y, Y'$, there are tuples $s_1, s_2 \in R$ such that $s[\alpha, \alpha'] = s_1[Y, Y']$, and $s[\beta, \beta'] = s_2[Y, Y']$. It follows that $s[\alpha'] = s[\beta']$, because of the fd $Y \rightarrow Y'$. Similarly, $t[\alpha'] = t[\beta']$. Therefore, $s[\beta'] = t[\beta']$.

Only-if part. Suppose that $E \not\models e$. Then there are a monoid M and a homomorphism $h: \Sigma^* \rightarrow M$ such that $h(\alpha_i) = h(\beta_i)$ but $h(\alpha) \neq h(\beta)$. For each element $y \in M$, let y' be a new distinct element. We construct R for h as in the proof of Theorem 2 with the additional clauses:

$$\begin{aligned} t_{x,y}[\alpha'] &= (h(\alpha))', \\ t_{x,y}[\beta'] &= (h(\beta))', \text{ and} \\ t_{x,y}[Y'] &= (h(y))'. \end{aligned}$$

Clearly, R satisfies D .

Let \tilde{M} be an isomorphic disjoint copy of M , and \tilde{h} the corresponding homomorphism from Σ^* . Again, for each element $y \in \tilde{M}$, let y' be a new distinct element, such that for all $x \in M$ and $y \in \tilde{M}$, we have that $x' = y'$ iff $x = h(\alpha)$ and $y = \tilde{h}(\alpha)$. Let \tilde{R} be constructed from \tilde{h} in a manner analogous to the construction of R from h . We leave it to the reader to verify that $R \cup \tilde{R}$ satisfies D but not d . Also, if $E \not\models_f e$, then we can assume that $R \cup \tilde{R}$ is finite. \square

COROLLARY. *The sets $\{(D, d): D \models d\}$ and $\{(D, d): D \not\models_f d\}$, where D ranges over finite sets of fd's and ind's and d ranges over fd's, are not recursive.*

4. Reduction to binary inclusion dependencies. An ind $A_1, \dots, A_k \subseteq B_1, \dots, B_k$ is said to be m -ary if $k \leq m$. The reduction in the previous section uses 4-ary ind's. It follows that the (finite) implication problem is undecidable even when restricted to 4-ary ind's. On the other hand, in [KCV] it is shown that both the implication and the finite implication problems are decidable when restricted to unary ind's. In this section we close the gap between those two results by showing that (finite) implication of fd's and ind's is reducible to (finite) implication of fd's and binary ind's.

THEOREM 4. *There is an algorithm that, given a set D of fd's and ind's and an fd (resp. ind) d , produces a finite set D' of fd's and binary ind's and an fd (resp. a unary ind) d' , such that $D \models d$ if and only if $D' \models d'$ and $D \models_f d$ if and only if $D' \models_f d'$.*

Proof. Let all the ind's in $D \cup \{d\}$ be m -ary. Furthermore, we can assume without loss of generality that the left-hand side and right-hand side of these ind's contain exactly m attributes (we can always duplicate attributes to achieve that). We denote a sequence A_1, \dots, A_m of attributes by \bar{A} . We view a sequence as a list of elements.

When we enclose the sequence in parentheses, e.g., (\bar{A}) , we refer to it as an element in the domain of sequences.

We first construct a set F of fd's and ind's. We introduce new attributes X_1, \dots, X_m, X , and put in F the fd's $\bar{X} \rightarrow X$ and $X \rightarrow \bar{X}$. For every ind $\bar{A} \subseteq \bar{B}$ in $D \cup d$, we introduce new attributes (\bar{A}) and (\bar{B}) and put in F the binary ind's $A_i, (\bar{A}) \subseteq X_i, X$ and $B_i, (\bar{B}) \subseteq \bar{X}, X$, for $1 \leq i \leq m$. Let τ be the ind $\bar{A} \subseteq \bar{B}$. We define τ' as the unary ind $(\bar{A}) \subseteq (\bar{B})$. If τ is an fd then we let $\tau' = \tau$. Now we define $D' = F \cup \{\tau' : \tau \in D\}$.

CLAIM 1. For all $\tau \in D \cup \{d\}$ we have $F \cup \{\tau'\} \models \tau$.

Proof of claim. The claim is trivially true for fd's. Let τ be the ind $\bar{A} \subseteq \bar{B}$. Let R be a relation that satisfies $F \cup \tau'$, and let $t \in R$. We have to show that there is a tuple $s \in R$ such that $s[(\bar{B})] = t[(\bar{A})]$. Since R satisfies τ' , there is a tuple $s \in R$ such that $s[(\bar{B})] = t[(\bar{A})]$. Since R satisfies F there are tuples $t_i, s_i \in R$ such that $t_i[X_i, X] = t[A_i, (\bar{A})]$ and $s_i[X_i, X] = s[B_i, (\bar{B})]$. But, since $s[(\bar{B})] = t[(\bar{A})]$, and R satisfies F , it follows that $t[A_i] = s[B_i]$. This is true for $1 \leq i \leq m$, which proves the claim.

CLAIM 2. For all $\tau \in D \cup d$ we have $F \cup \{\tau\} \models \tau'$.

Proof of claim. The claim is trivially true for fd's. Let τ be the ind $\bar{A} \subseteq \bar{B}$. Let R be a relation that satisfies $F \cup \{\tau\}$, and let $t \in R$. We have to show that there is a tuple $s \in R$ such that $s[(\bar{B})] = t[(\bar{A})]$. Since R satisfies F there are tuples $t_1, \dots, t_m \in R$ such that $t_1[A_i, (\bar{A})] = t_1[X_i, X]$. Furthermore, $t_1[X_i] = t[A_i]$, for $1 \leq i \leq m$, so $t_1[\bar{X}, X] = t[\bar{A}, (\bar{A})]$. Since R satisfies τ , there is a tuple $s \in R$ such that $s[(\bar{B})] = t[\bar{A}, (\bar{A})]$. Again, since R satisfies F , we can show that there is a tuple $s_1 \in R$ such that $s_1[\bar{X}, X] = s[(\bar{B})]$. It follows that $t_1[X] = s_1[X]$, so $t[(\bar{A})] = s[(\bar{B})]$. This proves the claim.

We can now prove the theorem.

Only-if part. Assume $D \models d$, and let R be a relation satisfying D' . By Claim 1, we have that R satisfies D , so by assumption R satisfies d . Since R satisfies F and d , it satisfies d' by Claim 2.

If part. Assume $D \not\models d$, and let R be a relation satisfying D but not d . We construct a relation R' satisfying D' but not d' . Let U be the sequence of attributes that are used in dependencies in $D \cup \{d\}$. Let U^m be a sequence consisting of all sequences of length m of attributes from U . We use “,” to denote concatenation of sequences. R_1 is a relation on U, U^m defined as:

$$\{t: t[U] \in R, t[(\bar{A})] = (t[\bar{A}]) \text{ for } \bar{A} \in U^m\}.$$

R_2 is a relation on \bar{X}, X defined as

$$\{t: t \in R_1[\bar{A}, (\bar{A})] \text{ for some } \bar{A} \in U^m\}.$$

Finally, R' is a relation on U, U^m, \bar{X}, X defined as

$$\{t: t[U, U^m] \in R_1 \text{ and } t[\bar{X}, X] \in R_2\}.$$

We leave it to the reader to show that R' satisfies F . Since $R'[U] = R[U]$, it follows that R' also satisfies D . By Claim 2 it follows that R' satisfies D' . Suppose now that R' satisfies d' . Then Claim 1 entails that R' satisfies d , and consequently that R satisfies d , contrary to the assumption. Thus R' does not satisfy d' .

Both parts of the proof work also for the finite case once we observe that the construction of R' from R preserves finiteness. \square

COROLLARY. The implication and the finite implication problems for fd's and binary ind's are undecidable.

5. Axiomatization. Parallel to the pursuit for algorithms to test implication is the pursuit for axiomatization of implication. That is, while it might be impossible to

recursively check whether $D \models d$, it might be possible to recursively check whether a given proof that $D \models d$ is correct. An immediate consequence of the existence of an axiomatization for a problem is the partial decidability of the problem; one has just to generate methodically all possible proofs and check for their correctness.

Consider now finite implication for fd's and ind's. It is not hard to see that the set $\{(D, d): D \not\models_f d\}$ is recursively enumerable; just enumerate all finite relations and check whether they satisfy D but not d . We have shown, however, that this set is not recursive. It follows that the set $\{(D, d): D \models_f d\}$ is not even recursively enumerable. Consequently, there can be no axiomatization for finite implication of fd's and ind's (an axiomatization for finite implication of fd's and unary ind's is described in [KCV]).

The situation with implication for fd's and ind's is different. Both fd's and ind's can be expressed by first-order sentences, and implication is reducible to validity of a first-order sentence. It follows that the set $\{(D, d): D \models d\}$ is recursively enumerable, and this raises the question whether implication of fd's and ind's can be axiomatized.

Casanova et al. [CFP] gave a partial negative answer to this question. They have shown that there is no k -ary axiomatization for implication of fd's and ind's, i.e., there is no axiomatization where the number of premises in the inference is bounded by some number k . (In contrast, fd's alone and ind's alone are known to have binary axiomatizations [Ar], [CFP].) In proving this result Casanova et al. assumed that axioms and inference rules do not introduce new attributes. That is, if π is a proof that $D \models d$, then only attributes that occur in dependencies in D or d occur in π . We call such an axiomatization *attribute-bounded* (it is called *universe-bounded* in [Va]). The axiomatizations for implication of fd's in [Ar] and for implication of ind's in [CFP] are both attribute-bounded. Mitchell [M1], on the other hand, has shown that by dropping the attribute-boundedness requirement we can get a ternary axiomatization for implication of fd's and ind's.

The question remained open whether there is an attribute-bounded (but non- k -ary) axiomatization for implication of fd's and ind's. (For an example of a non- k -ary axiomatization see [BV], [KCV].) Our undecidability results entail that the answer to this question is negative. Suppose to the contrary that there is such an axiomatization. Given a finite set of fd's and ind's D , and an fd or an ind d , there are only finitely many possible proofs for the implication $D \models d$ that uses only attributes that occur in D or d . By checking all these proofs, we can determine whether indeed $D \models d$ -contradiction.

Acknowledgment. We would like to thank John Mitchell for noting an error in an earlier draft.

REFERENCES

- [Ar] W. W. ARMSTRONG, *Dependency structure of database relationships*, Proc. IFIP 74, North-Holland, Amsterdam, 1974, pp. 580-583.
- [BB] C. BEERI AND P. A. BERNSTEIN, *Computational problems related to the design of normal form relational schemas*, ACM Trans. Database Systems, 4 (1979), pp. 30-59.
- [Be] P. A. BERNSTEIN, *Synthesizing third normal form relations from functional dependencies*, ACM Trans. Database Systems, 1 (1976), pp. 277-298.
- [BV] C. BEERI AND M. Y. VARDI, *Formal systems for tuple and equality generating dependencies*, this Journal, 13 (1984), pp. 76-98.
- [CFP] M. A. CASANOVA, R. FAGIN AND C. H. PAPADIMITRIOU, *Inclusion dependencies and their interaction with functional dependencies*, J. Comput. System Sci., 28 (1984), pp. 29-59.
- [Co] E. F. CODD, *Further normalization of the database relational model*, in Data Base Systems, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, 1972, pp. 33-64.

- [Fa] R. FAGIN, *A normal form for relational databases that is based on domains and keys*, ACM Trans. Database Systems, 6 (1983), pp. 387–415.
- [FV] R. FAGIN AND M. Y. VARDI, *Armstrong databases for functional and inclusion dependencies*, Inform. Proc. Letters, 16 (1983), pp. 13–20.
- [GL] Y. GUREVICH AND H. R. LEWIS, *The word problem for cancellation semigroups with zero*, J. Symbolic Logic, 49 (1984), pp. 184–191.
- [Gu] Y. GUREVICH, *The word problem for certain classes of semigroups (in Russian)*, Algebra and Logic, 5 (1966), pp. 25–35.
- [JK] D. S. JOHNSON AND A. KLUG, *Testing containment of conjunctive query under functional and inclusion dependencies*, J. Comput. System Sci., 28 (1984), pp. 167–189.
- [CV] P. C. KANELAKIS, S. S. COSMADAKIS AND M. Y. VARDI, *Unary inclusion dependencies have polynomial time inference problems*, Proc. 15th ACM Symposium on Theory of Computing, Boston, April 1983, pp. 264–277.
- [M1] J. MITCHELL, *The implication problem for functional and inclusion dependencies*, Proc. 2nd ACM Symposium on Principles of Database Systems, Atlanta, 1983, pp. 58–69.
- [M2] —, *The implication problem for functional and inclusion dependencies*, Tech. Rep. MIT/LCS/TM-235, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Feb. 1983.
- [Po] E. L. POST, *Recursive unsolvability of a problem of Thue*, J. Symbolic Logic, 12 (1947), pp. 1–11.
- [Va] M. Y. VARDI, *The implication and the finite implication problem for typed template dependencies*, J. Comput. System Sci., 28 (1984), pp. 3–28.

A LOWER BOUND FOR THE FORMULA SIZE OF RATIONAL FUNCTIONS*

K. A. KALORKOTI†

Abstract. We establish a lower bound for the formula size of quolynomials over arbitrary fields. Our basic formula operations are addition, subtraction, multiplication and division. The proof is based on Nečiporuk's [Soviet Math. Doklady, 7 (1966), pp. 999–1000] lower bound for Boolean functions and uses formal power series. This result immediately yields a lower bound for the formula size of rational functions over infinite fields. We also show how to adapt Nečiporuk's method to rational functions over finite fields. These results are then used to show that, over any field, the $n \times n$ determinant function has formula size at least $\Omega(n^3)$. We thus have an algebraic analogue to the $\Omega(n^3)$ lower bound for the Boolean determinant due to Kloss [Soviet Math. Doklady, 7 (1966), pp. 1537–1540].

Key words. quolynomial, rational function, formula size lower bound, computation tree, formal power series, determinant

1. Introduction. In this paper we provide a method for bounding from below the formula size of rational functions over arbitrary fields. The basic operations allowed are addition, subtraction, multiplication and division. The method is based on that of Nečiporuk [5] (also followed by Savage [6]) for Boolean functions. It is quite easy to adapt Nečiporuk's method if the ground field is finite. Also if attention is restricted to polynomials and division is not allowed, then there is again a fairly easy adaptation. The general case presents some difficulties in connection with division and these are overcome by the use of formal power series.

The results are used to show that the $n \times n$ determinant has formula size at least $\Omega(n^3)$ for any field. We thus have an algebraic analogue to the $\Omega(n^3)$ lower bound for the Boolean determinant due to Kloss [4].

2. Preliminaries. Throughout k will be a field and X a finite set of indeterminates over k . As usual the ring of polynomials in X with coefficients from k will be denoted by $k[X]$ and its quotient field by $k(X)$. The elements of $k(X)$ are sometimes called *quolynomials*.

Each $r \in k(X)$ defines a partial function $r: k^{X|} \rightarrow k$ as follows: put $r = p/q$ where $p, q \in k[X]$ and are coprime. If $\mathbf{a} \in k^{X|}$ and $q(\mathbf{a}) \neq 0$, then $r(\mathbf{a}) = p(\mathbf{a})/q(\mathbf{a})$, otherwise $r(\mathbf{a})$ is undefined (it will always be clear from the context whether r denotes a quolynomial or a partial function). Such functions are said to be *rational*.

DEFINITION. A *formula* over $k \cup X$ is any expression obtained by using the following rules a finite number of times:

- (a) w is a formula for all $w \in k \cup X$.
- (b) $(F_1 \circ F_2)$ is a formula whenever F_1, F_2 are formulae and $\circ \in \{+, -, \times, /\}$.

The *size* $|F|$ of a formula F is the number of times rule (b) is used in its construction. By observing brackets, F can be reduced in a unique way to obtain a numerator $n(F)$ and a denominator $d(F)$, both of these being polynomials in X . If $d(F) \neq 0$, then we say that F is a formula for the quolynomial $n(F)/d(F) \in k(X)$. Clearly each $r \in k(X)$ has a formula. The *formula size* $L(r)$ of r is given by

$$L(r) = \min \{|F| \mid F \text{ is a formula for } r\}.$$

* Received by the editors January 4, 1982, and in revised form September 8, 1984. An earlier version of the paper appeared in the conference proceedings. Automata, Languages and Programming, Ninth Colloquium, Aarhus, Denmark, July 1982, M. Nielsen and E. M. Schmidt, eds., Springer-Verlag, Berlin, 1982.

† Department of Computer Science, Edinburgh University, Edinburgh EH9 3JZ, Scotland.

When r is regarded as a rational function, its formula size $L^*(r)$ is given by

$$L^*(r) = \min \{L(r') \mid r' \text{ defines the same rational function as } r\}.$$

Thus $L^*(r) \leq L(r)$ and when the ground field is infinite equality holds (this follows from the fact that if k is infinite and $p \in k[X]$ then $p(\mathbf{a}) = 0$ for all $\mathbf{a} \in k^{|X|}$ (if and only if $p = 0$ in $k[X]$ (see Zariski and Samuel [9, Vol. I, p. 38])). When the ground field is finite, equality may not hold; for example over $GF(2)$ we have

$$L^*(x^2 + x) = L(0) = 0 < 2 = L(x^2 + x).$$

The first theorem we prove gives a lower bound for L over all fields and hence for L^* when k is infinite. The second theorem provides a lower bound for L^* when k is finite.

Remark. Our definition of a formula for $r \in k(X)$ allows the possibility of division by 0. For example $(x) + (1/(1/0))$ is a formula for x . However we are interested in minimal size formulae and here division by 0 cannot occur.

In this paper all trees will be rooted and directed with all edges directed towards the root (in diagrams the root is shown at the top and the leaves at the bottom). Let v_1, v_2 be vertices of a tree. v_1 is a *predecessor* of v_2 if (v_1, v_2) is an edge of the tree. v_1 is an *ancestor* of v_2 if there is a directed path from v_1 to v_2 . The *indegree* of a vertex is the number of its predecessors. The *subtree* of a vertex v consists of v together with all of its ancestor vertices and edges. A tree is *binary* if each nonleaf vertex has indegree 2. It is easy to see that if a binary tree has l leaves then it has $l - 1$ nonleaf vertices.

A *computation tree* over $k \cup X$ is a tree T such that the set of predecessors of each vertex is totally ordered (in diagrams the predecessors of a vertex are shown in increasing order from left to right). Each leaf of T is labelled with an element of $k \cup X$. If v is any other vertex of indegree d , then v is labelled with a rational function $R_v : k(X)^d \rightarrow k(X)$. Such vertices are called *computation vertices* and the number of them is denoted by $C(T)$. A vertex v *involves* an indeterminate x if there is a leaf labelled x which is an ancestor of v . We associate elements of $k(X)$ with some (possibly all) the vertices of T as follows:

- (a) Each leaf is associated with its label.
- (b) Let v be a computation vertex of indegree d with predecessors $v_1 < v_2 < \dots < v_d$. If each v_i has an element $r_i \in k(X)$ associated with it and $R_v(r_1, \dots, r_d)$ is defined, then we associate this element with v . Otherwise we do not associate any element with v .

If $r \in k(X)$ is associated with a vertex v , then we say that v *computes* r . If the root of T computes r , then we also say that T computes r .

Given a formula F over $k \cup X$, we can associate a computation tree T over $k \cup X$ with F in the obvious way. Each computation vertex of T is labelled with $+$, $-$, \times or $/$ and so T is binary. Moreover $|F| = C(T)$. Note that if no divisions by 0 occur in reducing F to $n(F)$ and $d(F)$, then T computes $n(F)/d(F)$.

$k[[X]]$ denotes the ring of formal power series in X over k . The elements of $k[[X]]$ are expressions of the type

$$f = \sum_{i=0}^{\infty} f_i$$

where f_i is either 0 or a form of degree i (that is a polynomial all of whose monomials have degree i). f_0 is an element of k and is called the *constant term* of f . The *order* of f , denoted by $\text{ord}(f)$, is the least i such that $f_i \neq 0$ if $f \neq 0$ and ∞ otherwise. Addition and multiplication in $k[[X]]$ are carried out in the obvious manner. Clearly $k[X]$ is embedded in $k[[X]]$.

Suppose $r \in k(X)$ and we want a lower bound for $L(r)$. It will be convenient for us to expand r as a formal power series. This, however, is not always possible. Fortunately, for our purposes, there is little loss in restricting attention to those elements of $k(X)$ which can be so expanded. We proceed to describe the set of all such elements more formally.

A power series $f \in k[[X]]$ is a *unit* if there is a $g \in k[[X]]$ such that $fg = 1$. In such a case g is unique and is denoted by f^{-1} . It is easily seen that f is a unit if and only if $\text{ord}(f) = 0$, that is to say f has a nonzero constant term. Let

$$k^u(X) = \{r \in k(X) \mid \exists p, q \in k[X] \text{ s.t. } r = p/q \text{ and } q \text{ is a unit in } k[[X]]\}.$$

Clearly $k^u(X)$ is a subring of $k(X)$. Define

$$P_X : k^u(X) \rightarrow k[[X]]$$

as follows: given $r \in k(X)$, let $r = p/q$ where $p, q \in k[X]$ and q is a unit in $k[[X]]$. Then $P_X(r) = pq^{-1}$. Easily, P_X is well defined and is in fact an 1-1 homomorphism, so it is an embedding.

We now look at the effect of restricting attention from $k(X)$ to $k^u(X)$.

DEFINITION. Let $r(x_1, \dots, x_n) \in k(X)$ and $a_1, \dots, a_n \in k$. Then we call $r(x_1 - a_1, \dots, x_n - a_n)$ the *translate* of r by a_1, \dots, a_n .

It is easily seen that if k is infinite, then each element of $k(X)$ has a translate in $k^u(X)$. The following is straightforward.

LEMMA 1. *Let $r, s \in k(X)$ with s a translate of r . Then $L(r) \cong \frac{1}{2}(L(s) - 1)$. \square*

In his method Nečiporuk partitioned the variables of a Boolean function B into disjoint subsets. For each subset he used certain transformations on a minimal formula of B and the number of subfunctions of B induced by assignments to variables outside the subset to obtain the lower bound. We now proceed to give algebraic analogues of assignments to variables and of counting the number of induced subfunctions.

From now on Y, Z will be finite sets of indeterminates over $k(X)$, thus $X \cap Y = X \cap Z = \emptyset$.

DEFINITION. A substitution of $k[[Y]]$ into $k[[Z]]$ consists of:

- (a) A function $\sigma : Y \rightarrow k[[Z]]$ such that $\text{ord}(\sigma(y)) > 0$ for all $y \in Y$.
- (b) The unique extension of σ to a homomorphism $k[[Y]] \rightarrow k[[Z]]$ given by

$$f^\sigma(y_1, \dots, y_n) = f(\sigma(y_1), \dots, \sigma(y_n)).$$

(See Zariski and Samuel [9, Vol. II, p. 135] where a wider class of substitutions is defined.)

We define σ on $k^u(Y)$ as follows: given $r \in k(Y)$, put $r = p/q$ where $p, q \in k[Y]$ and q is a unit in $k[[Y]]$. Then $r^\sigma = p^\sigma/q^\sigma$. Note that as q is a unit it has a nonzero constant term and so $q^\sigma \neq 0$. Moreover r^σ is well defined and is in $k^u(Z)$.

LEMMA 2. *Let $r \in k^u(Y)$ and σ a substitution of $k[[Y]]$ into $k[[Z]]$. Then $P_Y(r)^\sigma = P_Z(r^\sigma)$.*

Proof. Let $r = p/q$ with q a unit in $k[[Y]]$. Since σ is a homomorphism, we have $(q^{-1})^\sigma = (q^\sigma)^{-1}$ and so $p^\sigma(q^{-1})^\sigma = p^\sigma(q^\sigma)^{-1}$. \square

A substitution σ of $k[[X, Y]]$ into $k[[X, Z]]$ respects X if $\sigma(x) = x$ for all $x \in X$ and $\sigma(y) \in k[[Z]]$ for all $y \in Y$. Given $f \in k[[X, Y]]$ and $g \in k[[X, Z]]$, we say that f represents g with respect to X if there is a substitution σ which respects X such that $f^\sigma = g$.

Suppose f, g are as above and f represents g with respect to X . We shall need to have a lower bound for $|Y|$ in terms of g ; this is our algebraic analogue to the counting of subfunctions in Nečiporuk's argument. If f, g were both polynomials, they could be regarded as polynomials in X with coefficients from $k[Y]$ and $k[Z]$ respectively.

Furthermore they could be written uniquely as

$$f = \sum_{i=0}^a f_i(Y)M_i(X), \quad g = \sum_{i=0}^b g_i(Z)M_i(X),$$

where the $M_i(X)$ are distinct monomials in X and a, b are the numbers of such monomials in f and g respectively.

Recall that elements q_1, \dots, q_n of $k[Y]$ are said to be *algebraically independent over k* if the only polynomial P with coefficients in k such that $P(q_1, \dots, q_n) = 0$ is the zero polynomial. The *transcendence degree* of a subset S of $k[Y]$ is the maximum number of algebraically independent elements (over k) of S . It can be seen that this number is at most $|Y|$. (In this paper transcendence degrees will always be over k . In view of this we shall henceforth omit the phrase “over k ” and use the abbreviation *tr deg*.) The concept is analogous to dimension in vector spaces. Further material is given by Zariski and Samuel [9, Vol. I] and van der Waerden [8, Vol. I].

With f, g represented as above define

$$\text{td}_X(g) = \text{tr deg} \{g_0, \dots, g_b\}.$$

Now $f^\sigma = g$ if and only if $f_i^\sigma = g_i$ for $0 \leq i \leq b$, and $f_i^\sigma = 0$ for $b < i \leq a$. It follows that

$$\text{tr deg} \{f_0, \dots, f_b\} \geq \text{tr deg} \{g_0, \dots, g_b\}.$$

But

$$(*) \quad |Y| \geq \text{tr deg } k[Y] \geq \text{tr deg} \{f_0, \dots, f_b\}.$$

Thus $|Y| \geq \text{td}_X(g)$ as required.

Unfortunately the inequality (*) is false if $k[Y]$ is replaced by $k[[Y]]$ (indeed if $|Y| \geq 2$, then $k[[Y]]$ contains infinitely many algebraically independent elements). The rest of this section translates the above idea to formal power series in such a way as to avoid the stated difficulty.

DEFINITION. Let $f_i \in k[[Y]]$ for $i \in I$, where $f_i = \sum_{j=0}^\infty f_{ij}$. Let $m_f = \min_{i \in I} \text{ord}(f_i)$ if some $f_i \neq 0$ and $m_f = 0$ otherwise. Define

$$\text{td} \{f_i | i \in I\} = \text{tr deg} \{f_{im_f} | i \in I\}.$$

Note that this is always at most $|Y|$.

Given $r_i \in k^u(Y)$ for $i \in I$, define

$$\text{td} \{r_i | i \in I\} = \text{td} \{P_Y(r_i) | i \in I\}.$$

LEMMA 3. Let $f_i \in k[[Y]]$ and $g_i \in k[[Z]]$ for $i \in I$. Suppose there exists a substitution σ of $k[[Y]]$ into $k[[Z]]$ such that $f_i^\sigma = g_i$ for all $i \in I$. Then

$$|Y| \geq \text{td} \{g_i | i \in I\}.$$

Proof. Let $Y = \{y_1, \dots, y_s\}$ and $Z = \{z_1, \dots, z_t\}$. Truncate each f_i to a polynomial f'_i defined by

$$f'_i = \sum_{j=0}^{m_g} f_{ij}$$

where $m = m_g$. Then, for all $i \in I$,

$$f'_i(\sigma(y_1), \dots, \sigma(y_s)) = g_{im}(z_1, \dots, z_t) + h_i(z_1, \dots, z_t)$$

for some h_i with $\text{ord}(h_i) > m$.

We claim that

$$\text{tr deg } \{f'_i | i \in I\} \cong \text{tr deg } \{g_{im} | i \in I\}.$$

For suppose P is a nonzero polynomial such that

$$P(f'_{i_1}, \dots, f'_{i_r}) = 0, \quad i_1, \dots, i_r \in I.$$

Applying σ to this equation, we obtain

$$(*) \quad P(g_{i_1 m} + h_{i_1}, \dots, g_{i_r m} + h_{i_r}) = 0.$$

Let H be the homogeneous part of P of lowest degree. Then the left-hand side of $(*)$ can be written as $H(g_{i_1 m}, \dots, g_{i_r m}) + Q$ where each monomial in Q is of degree greater than each monomial in $H(g_{i_1}, \dots, g_{i_r})$, the degree being with respect to Z . Thus $H(g_{i_1}, \dots, g_{i_r}) = 0$ and the claim follows.

The result is now proved since $|Y| \cong \text{tr deg } \{f'_i | i \in I\}$. \square

Let $f \in k[[X, Y]]$ and $g \in k[[X, Z]]$ and suppose there is a substitution σ which respects X such that $f^\sigma = g$. We can regard f and g as elements of $k[[Y]][[X]]$ and $k[[Z]][[X]]$ respectively. Thus f can be written uniquely as

$$f = \sum_{i=0}^{\infty} f_i(X)$$

where $f_i(X) \in k[[Y]][X]$ and is either 0 or a form of degree i (with respect to X). Let $\{M_{ij}(X) | 0 \leq j \leq s_i\}$ be the set of all monomials in X of degree i . Then $f_i(X)$ can be written uniquely as

$$f_i(X) = \sum_{j=0}^{s_i} f_{ij}(Y) M_{ij}(X)$$

where $f_{ij}(X) \in k[[Y]]$. Similarly for g .

Note that $f^\sigma = g$ if and only if $f_{ij}^\sigma = g_{ij}$ for all $i \geq 0$ and $0 \leq j \leq s_i$.

DEFINITION. $\text{td}_X(g) = \max \{ \text{td } S | S \subseteq \{g_{ij} | i \geq 0 \text{ and } 0 \leq j \leq s_i\} \}$.

From the remarks above and Lemma 3 we have

LEMMA 4. *Let f, g be as above. If f represents g with respect to X , then $|Y| \cong \text{td}_X(g)$. \square*

3. The results.

THEOREM 1. *Suppose $f \in k(X)$ and $f \notin k(X')$ for any $X' \subset X$. Let X_1, \dots, X_r be a partition of X into disjoint subsets and suppose f has a translate $g \in k^u(X)$. Then*

$$L(f) \cong \frac{1}{24} \sum_{i=1}^r \text{td}_{X_i}(g).$$

Proof. Let T be the tree of a minimal formula for g . Let l_i be the number of leaves of T with a label from X_i and put $l = \sum_{i=1}^r l_i$. Since T is binary we have

$$(i) \quad C(T) \cong l - 1.$$

Fix i and call an indeterminate in X_i *fixed* and one which is not in X_i *free*.

We now introduce some terminology which will be used in the algorithm below. Let Z be any set of free indeterminates over k and let T' be any computation tree over $k \cup X_i \cup Z$ each of whose vertices computes some element of $k(X_i, Z)$. We say that a computation vertex is *fixed* if it involves at least one fixed indeterminate and *free* if it involves only free indeterminates. A computation vertex is *mixed* if it has indegree 2 and one of its predecessors is fixed while the other is free. Let v be a mixed

vertex and let v_0, v_1, \dots, v_s be the longest directed path containing v such that v_0 is fixed but not mixed while v_1, \dots, v_s are all mixed (clearly such a path is unique). Call this path the *mixed path* of v . Let $r \in k(X_i, Z)$ be the expression computed by v_0 . Then v_s computes an expression of form $(f_1 r + f_2)/(f_3 r + f_4)$ where $f_i \in k(Z)$ for $1 \leq i \leq 4$. Denote this expression by $R(v)$. If $f_3 = 0$, define $\tau(v) = 2$ and otherwise $\tau(v) = 3$. Let $h(y, z, a, b) = (y + a)/(z + b)$ where y, z are free indeterminates and $a, b \in k$. It is easy to see that given new free indeterminates y_i, z_i for $1 \leq i \leq \tau(v)$, then we can choose $a_i, b_i \in k$ for $1 \leq i \leq \tau(v)$, such that either $h(y_1, z_1, a_1, b_1)r + h(y_2, z_2, a_2, b_2)$ or $h(y_3, z_3, a_3, b_3) + h(y_2, z_2, a_2, b_2)/(r + h(y_1, z_1, a_1, b_1))$ represents $R(v)$ with respect to X according as $\tau(v)$ is 2 or 3. We refer to a_i, b_i for $1 \leq i \leq \tau(v)$, as the *constants* of $R(v)$.

Let M be the set of mixed vertices of T . We apply a sequence of transformation to T according to the following algorithm:

```

begin
  choose  $v \in M$ ;
   $Z \leftarrow X - X_i$ ;
   $T' \leftarrow T$ ;
  while  $M \neq \emptyset$  do
    begin
      let  $v_0, v_1, \dots, v_s$  be the mixed path of  $v$ ;
      let  $a_i, b_i$  be the constants of  $R(v)$  and  $y_i, z_i$  be new free indeterminates,  $1 \leq i \leq \tau(v)$ ;
      delete the subtree of  $v_s$  and replace it with the relevant subtree from Fig. 1;
      if  $v_s$  is a predecessor in  $T'$ , then  $u$  takes its place;
      let  $T''$  be the tree thus obtained;
      for each vertex in  $T''$  order its predecessors either as in  $T'$  or as indicated in Fig. 1. (If  $v_s$  was a predecessor, then  $u$  inherits its place in the order.);
       $M \leftarrow M - \{v_1, \dots, v_s\}$ ;
      choose  $v \in M$ ;
       $Z \leftarrow Z \cup \{y_i, z_i | 1 \leq i \leq \tau(v)\}$ ;
       $T' \leftarrow T''$ 
    end
  end

```

Let T_i be the final tree of the logarithm and Z_i the set of free indeterminates in T_i . By induction on the number of transformations used and the observations above it is easily seen that T_i computes some $g_i \in k^u(X_i, Z_i)$ which represents g with respect to X_i . Thus, by Lemma 4,

$$(ii) \quad |Z_i| \geq \text{td}_{X_i}(g).$$

Let c_0, c_1 be the number of free and fixed vertices in T_i respectively. Let c_2 be the number of vertices with both predecessors fixed. Since f , and hence g , is not in $k(X')$ for any $X' \subset X$ it follows that $l_i > 0$ and so T_i has at least one fixed vertex. Because of this and the transformations used it follows that each free vertex is attached to a mixed vertex and so $c_0 \leq c_1$. Each mixed path v_0, v_1, \dots, v_s in T_i has length at most 4 and v_0 is either a leaf or a fixed vertex with both predecessors fixed. Thus $c_1 \leq 3c_2 + 3l_i$. Now by induction on l_i we have $c_2 \leq l_i - 1$. Thus

$$\begin{aligned}
 C(T_i) &= c_0 + c_1 + c_2 \\
 &\leq 13l_i - 7.
 \end{aligned}$$

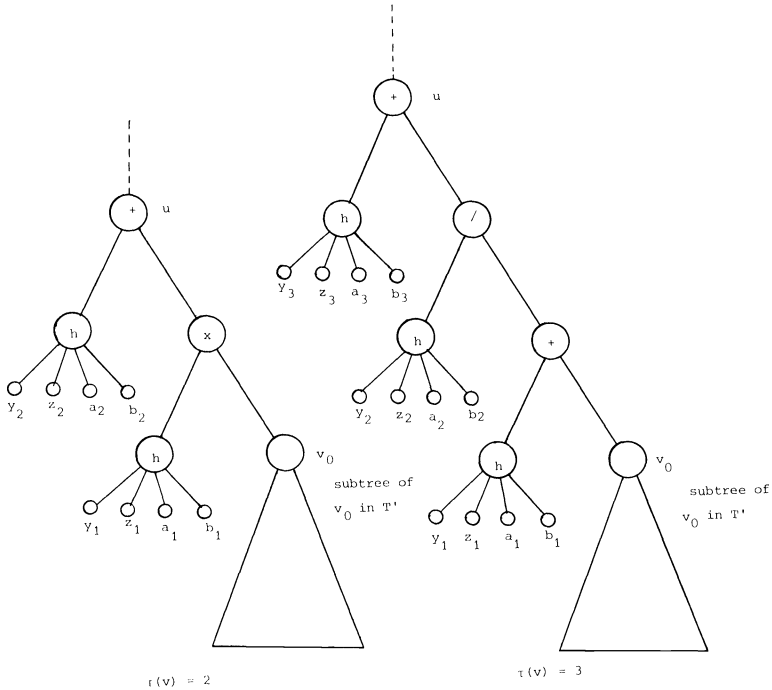


FIG. 1

The fixed vertices of T_i all have indegree 2 and the free ones all have four leaves as predecessors: two labelled with elements of Z_i and two with elements of k . Thus deleting all leaves of the second kind results in a binary tree with $l_i + |Z_i|$ leaves. It now follows that

$$C(T_i) = l_i + |Z_i| - 1.$$

This and (ii) yield

$$l_i \geq (|Z_i| + 6) > \frac{1}{12} \text{td}_{X_i}(g).$$

This and (i) now yield

$$L(g) = C(T) > \frac{1}{12} \sum_{i=1}^t \text{td}_{X_i}(g) - 1,$$

and the result follows from Lemma 1. \square

Remark. If $f \in k^u(X)$ then the multiplicative constant in Theorem 1 may be improved to $\frac{1}{12}$.

We now deal with rational functions over finite fields.

DEFINITION. Let $W \subseteq X$ and $\alpha : X \rightarrow k \cup X$ with $\alpha(x) = x$ for all $x \notin W$ and $\alpha(x) \in k$ for all $x \in W$. Given $q(x_1, \dots, x_n) \in k[X]$, define

$$q^\alpha(x_1, \dots, x_n) = q(\alpha(x_1), \dots, \alpha(x_n)).$$

Let $r = p/q \in k(X)$ where $p, q \in k[X]$. If $q^\alpha \neq 0$ in $k[X]$, then define $r^\alpha = p^\alpha/q^\alpha$ and call r^α a W -specialization of r . Two W -specializations of r are *equivalent* if they define the same rational function over k . Define

$$\text{sp}_{X-W}(r) = \# (\text{inequivalent } W\text{-specializations of } r).$$

Thus if $r, s \in k(X)$, define the same rational function then $\text{sp}_{X-w}(r) = \text{sp}_{X-w}(s)$. Note also that $\text{sp}_{X-w}(r) \leq |k|^{|\text{X-w}|}$ and if r is defined on at least one point of $k^{|\text{X}|}$, then $\text{sp}_{X-w}(r) > 0$.

THEOREM 2. Let $f \in k(X)$ be defined on at least one point of $k^{|\text{X}|}$ and suppose $f \notin k(X')$ for any $X' \subset X$. Let X_1, \dots, X_t be a partition of X into disjoint subsets. Then

$$L^*(f) \geq \frac{1}{6} \sum_{i=1}^t \log_{|k|} \text{sp}_{X_i}(f).$$

Proof. Let $L^*(f) = L(g)$. We make some simple changes to the proof of Theorem 1. Inequality (i) still holds. We replace the transformation trees of Fig. 1 by those of Fig. 2 where x_i for $1 \leq i \leq \tau(v)$, are new free indeterminates and alter the algorithm accordingly. For a given i the tree T_i obtained by applying the algorithm computes some $g_i \in k(X_i, Z_i)$ such that any $(X - X_i)$ -specialization of g , and hence of f , is equal to some Z_i -specialization of g_i (this would not hold if f were nowhere defined). Thus $|Z_i| \geq \log_{|k|} \text{sp}_{X_i}(f)$.

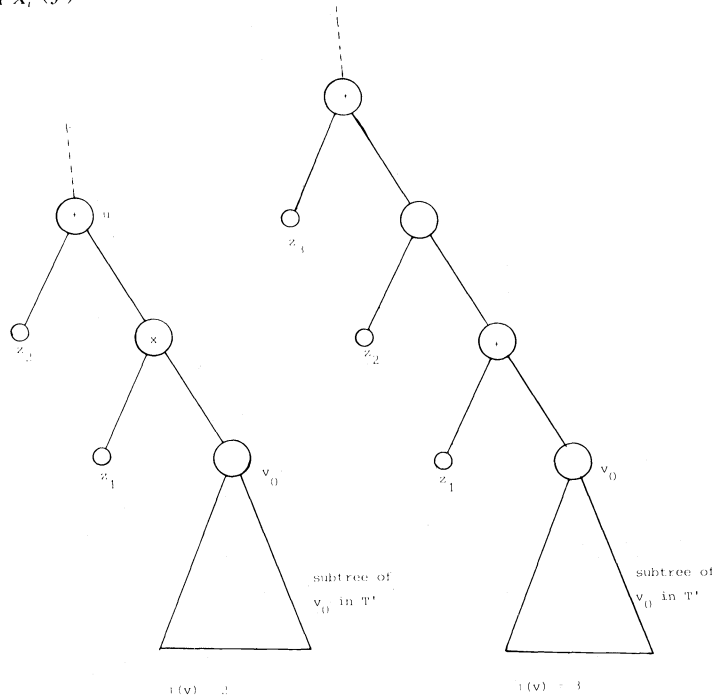


FIG. 2

The rest of the proof is the same as that of Theorem 1, taking into account that now $c_0 = 0$. \square

Remark. Neither of the bounds of Theorem 1 and Theorem 2 can grow faster than $|X|^2$. This is an inherent limitation of Nečiporuk's method. An expression which achieves this order of growth for Theorem 1 is

$$u = \sum_{i=1}^n \sum_{j=i+1}^n x_j x_i^{j-1}.$$

To see this take $X_i = \{x_i\}$ and note that

$$\text{td}_{X_i}(u) \geq \text{tr deg } \{x_{i+1}, \dots, x_n\} = n - i.$$

Thus

$$L(u) \geq \frac{1}{12} \sum_{i=1}^n (n-i) = \Omega(n^2).$$

It follows that for infinite fields we have $L^*(u) \geq \Omega(n^2)$.

4. An application. Let $M = (x_{ij})$ be an $n \times n$ matrix where the x_{ij} are indeterminates over k . Define $\det M \in k[x_{11}, x_{12}, \dots, x_{nn}]$ by

$$\det M = \sum_{\sigma \in S_n} (-1)^{s(\sigma)} \prod_{i=1}^n x_{i,\sigma i}$$

where S_n is the symmetric group of degree n and $s(\sigma)$ is 0 or 1 according as σ is even or odd.

PROPOSITION 1. $L(\det M) \geq \Omega(n^3)$, for all fields.

Proof. Let $X_i = \{x_{1,i}, x_{2,i+1}, \dots, x_{n,i+n-1}\}$ for $1 \leq i \leq n$, all indices being taken modulo n . We claim that $\text{td}_{X_i}(\det M) \geq \frac{1}{2}n(n-1)$ for all i . By symmetry it suffices to show this for $i = 1$.

Let $d_{ij} = x_{11} \cdots \hat{x}_{ii} \cdots \hat{x}_{jj} \cdots x_{nn}$ for $1 \leq i < j \leq n$, where $\hat{}$ denotes a missing term. There are $\frac{1}{2}n(n-1)$ distinct terms d_{ij} in $\det M$. Moreover the coefficient of d_{ij} is $x_{ij}x_{ji}$. The claim now follows. Thus, by Theorem 1 and the remark following it, we have

$$L(\det M) \geq \frac{1}{12} \sum_{i=1}^n \frac{1}{2}n(n-1) = \Omega(n^3). \quad \square$$

PROPOSITION 2. Let k be a finite field and $\det: k^{n^2} \rightarrow k$ the determinant function. Then

$$L^*(\det) \geq \Omega(n^3).$$

Proof. Partition the indeterminates as in Proposition 1. By a trivial adaptation of the lemma of Kloss [4] (also followed by Savage [6, p. 105]) we have $\text{sp}_{X_i}(\det) \geq |k|^{(1/2)n(n-1)}$ for each i . The result now follows from Theorem 2. \square

Putting Propositions 1 and 2 together, we obtain

THEOREM 3. Let k be any field and $\det: k^{n^2} \rightarrow k$ the determinant function. Then

$$L^*(\det) \geq \Omega(n^3).$$

We finish by remarking that the best known upper bound for $L^*(\det)$ is $0(n^{\log n})$ obtained by Csanky [2] for fields of characteristic 0. The same bound is obtained for all fields by Borodin, von zur Gathen and Hopcroft [1]. It is also possible to obtain this bound from the result of Hyafil [3] provided one uses the observation of Strassen [7] about division as in the paper of Borodin, von zur Gathen and Hopcroft [1].

Acknowledgment. I should like to thank Prof. L. G. Valiant for suggesting the above area of research to me and Dr. S. Skyum for his helpful comments on a preliminary draft of this paper.

REFERENCES

- [1] A. BORODIN, J. VON ZUR GATHEN AND J. HOPCROFT. *Fast parallel matrix and GCD computations*, Inform. and Control, 52 (1982), pp. 241-256.
- [2] L. CSANKY, *Fast parallel inversion algorithms*, this Journal, 5 (1976), pp. 618-623.
- [3] L. HYAFIL, *On the parallel evaluation of multivariate polynomials*, Proc. Tenth ACM Symposium on Theory of Computing, 1978, pp. 193-195.

- [4] B. M. KLOSS, *Estimates of the complexity of solution of systems of linear equations*, Dokl. Akad. Nauk USSR, 171 (1966), pp. 781-783; Soviet Math. Dokl., 7 (1966), pp. 1537-1540.
- [5] È. I. NEČIPORUK, *A Boolean function*, Dokl. Akad. Nauk USSR, 169 (1966), pp. 765-766; Soviet Math. Dokl., 7 (1966), pp. 999-1000.
- [6] J. E. SAVAGE, *The Complexity of Computing*, John Wiley, New York, 1976.
- [7] V. STRASSEN, *Vermeidung von Divisionen*, J. Reine Ange. Math., 264 (1973), pp. 184-202.
- [8] B. L. VAN DER WAERDEN, *Modern Algebra*, Vols I, II, Frederic Ungar, New York, 1949.
- [9] O. ZARISKI AND P. SAMUEL, *Commutative Algebra*, Vols I, II, Van Nostrand, Princeton, NJ, 1958.

ON PARALLEL SEARCHING*

MARC SNIR†

Abstract. We investigate the complexity of searching a sorted table of n elements on a synchronous, shared memory parallel computer with p processors. We show that $\Omega(\lg n - \lg p)$ steps are required if concurrent accesses to the same memory cell are not allowed, whereas $O(\lg n / \lg p)$ steps are sufficient if simultaneous reads are allowed. The lower bound is valid even if only communication steps are counted, and the computational power of each processor is not restricted. In this model, $\Theta(\sqrt{\lg n})$ steps are required for searching when the number of processors is unbounded. If the amount of information that a memory cell may store is restricted, then the time complexity for searching with an unbounded number of processors is $\Theta(\lg n / \lg \lg n)$. If the amount of information a processor may hold is also restricted, then an $\Omega(\lg n)$ lower bound holds. These lower bounds are first proven for comparison-based algorithms; it is next shown that comparison-based algorithms are as powerful as more general ones in solving problems defined in terms of the relative order of the inputs.

Key words. parallel algorithms, parallel computations

1. Introduction. With the advance in microelectronics it becomes feasible to build parallel machines with thousands of cooperating processors. Yet, practice indicates that a thousandfold increase in raw computational power does not increase performance by the same amount. There are two main reasons for that. The first one is that not every problem admits an efficient parallel solution. The second one is that not every parallel algorithm can be mapped efficiently onto a realistic parallel computer architecture. Work sharing between many processors generates significant overheads for communication of data and coordination. The study in parallel complexity is dedicated, to a large extent, to the understanding of these two phenomena.

One useful model for the study of parallel computations is that of a *paracomputer*. It consists of many identical autonomous processors, each with its own local memory and its own program. In addition, the machine has a shared memory and each processor can in one step access any cell in shared memory.

We obtain successively weaker models by varying the assumptions concerning simultaneous accesses to shared memory:

- (1) *Concurrent Read, Concurrent Write (CRCW)*. Both simultaneous reads and simultaneous writes to the same memory cell are allowed. The effect of simultaneous actions by the processors is as if the actions occurred in some serial order (for other possible definitions of CRCW, see [2]).
- (2) *Concurrent Read, Exclusive Write (CREW)*. Simultaneous reads are allowed but a processor can modify a shared memory cell only if it has exclusive access to it.
- (3) *Exclusive Read, Exclusive Write (EREW)*. Simultaneous accesses to the same shared memory cell are not allowed.

This model can be further weakened in two ways: One can restrict the number of shared memory cells. One can also restrict the set of processors that have read or write access to each memory cell. If there is a unique processor that has read access and a unique processor that has write access to each shared memory cell, then each

* Received by the editors June 15, 1982, and in final revised form February 21, 1984. This work was supported in part by the National Science Foundation under grant MCS-8203307.

† Courant Institute of Mathematical Sciences, New York University, New York, New York 10012. Present address, Department of Computer Science, The Hebrew University of Jerusalem, Jerusalem 91904, Israel.

memory cell represents a unidirectional link that connects two processors. We speak then of an *ultracomputer*.¹

In these models communication both of data and of control information is done through the shared memory. Thus, studying the relative power of these models is tantamount to studying the effect of constraints on communication and coordination on computational power.

We consider the problem of searching a key within a sorted list of n keys. The binary search algorithm solves this problem in (optimal) sequential time $O(\lg n)$. This can be generalized to a “ $(p+1)$ -ary” search algorithm that solves the problem in $O(\log_{p+1} n)$ steps with p processors. At each step p comparisons are done that split the list into $p+1$ equal length segments, and the search proceeds recursively within the unique segment that may contain the key. This algorithm is optimal, so that p processors speed up searching by a factor of $\lg(p+1)$ only: Searching does not admit an efficient parallel solution.

Consider now the problem of implementing this parallel algorithm. It turns out that the speedup can be achieved only if one item of information can be broadcast to all processors in constant time. On the other hand we show that in the EREW model, where an item of information can be accessed by one processor only at a time, searching requires at least $\Omega(\lg n - \lg p)$ steps. Note that no transmission of data is required for parallel search, but the processors need to coordinate the search at each iteration. It turns out that the time spent in coordinating the processors offsets exactly the gain obtained from simultaneous table look-ups.

The $\Omega(\lg n - \lg p)$ lower bound is valid even if each processor may in one step do any amount of local processing or transfer any amount of information. The only restrictions are that at each step a processor may read or write at a unique location in memory.

This result settles the problem of the relative power of the different shared-memory machine models. It is known that a p -processor machine which supports concurrent accesses to the same location in memory can be simulated by a p -processor machine with no concurrent accesses to the same location in memory with a $\lg p$ time penalty [5]. Our result shows that this simulation is optimal.

Under the same assumptions we prove that the time complexity of searching with an unbounded number of processors is $\Theta(\sqrt{\lg n})$.

The computational model is very strong since there are no restrictions on the amount of information that can be transmitted in one step. This is remedied by assuming that a memory cell may contain a unique input value, and that inputs are atomic entities, so that an input symbol cannot be used to encode the values of several inputs. A memory cell may also store a symbol taken from a small domain of internal values. In this model the time complexity of searching with an unbounded number of processors is $\Theta(\lg n / \lg \lg n)$. Finally, if we impose a similar restriction on the local memory of each of the processors, then an $\Omega(\lg n)$ lower bound is valid, independently of the number of processors.

The $\Omega(\lg n)$ lower bound on search implies a similar lower bound for the insertion problem on a shared memory parallel machine with no concurrent access to the same memory location. This settles an open problem posed by Borodin and Hopcroft in [2].

The lower bounds for searching with an unbounded number of processors are proven for comparison-based computations. We also prove that comparison-based

¹ The terms paracomputer and ultracomputer are taken from [10], but are used here in a slightly different meaning.

algorithms are as efficient as more general ones in solving problems that are defined in terms of the relative order of the inputs. We first show that if such a problem can be solved by comparisons for inputs taken from a subset of values where every possible permutation of the inputs occurs, then it can be solved by comparisons, using the same resources, for any input. We next prove that given a paracomputer, one can build a sufficiently large set of input values where the behavior of each processor at each step depends only on the relative order of the pairs of inputs it has access to, but not on their actual values. The last result is proven by an application of Ramsey's theorem.

The remainder of this paper is organized as follows. The implementation of the " $(p+1)$ -ary" searching algorithm is discussed in the next section. In § 3 we prove the $\Omega(\lg n - \lg p)$ lower bound for a simplified version of the searching problem. This is followed in § 4 by a description of an $O(\sqrt{\lg n})$ algorithm for searching in the EREW model with $O(n)$ processors. In § 5 we present the reduction of general paracomputer computations to computations using only comparisons. This reduction is used in § 6 to prove the $\Omega(\sqrt{\lg n})$ lower bound for searching with an unrestricted number of processors. In § 7 we examine the complexity of searching in restricted paracomputer models. Conclusions and open problems are brought in the last section.

2. Parallel searching in the CREW model. The search problem has several variants, all of which have essentially the same complexity. For sake of simplicity we consider the following version.

Range search problem (for a table of size n). Given $n+1$ distinct inputs x_1, \dots, x_n, y such that $x_1 < \dots < x_n$, find the index i such that $x_i < y < x_{i+1}$. (By definition $x_0 = -\infty$ and $x_{n+1} = \infty$.)

We first determine the complexity of searching in the CRCW and CREW models of parallelism.

THEOREM 2.1. *The range search problem for a table of size n can be solved on a CREW machine with p processors in time $O(\lg(n+1)/\lg(p+1))$.*

Proof. The algorithm used is the obvious extension of binary search to p processors, namely $(p+1)$ -ary search: p keys are chosen that divide the list of keys into $p+1$ intervals of roughly equal length. These keys are compared in parallel to the searched key. The comparisons locate the searched key within one of the subintervals, and the search proceeds recursively within this subinterval. At each iteration the length of the list is decreased by a factor of $p+1$, so that the search ends in $\lg_{p+1}(n+1)$ iterations. It remains to be shown that each iteration can be implemented in constant time, without concurrent writes.

Note that the searched key is located within the i th subinterval iff the outcomes of the comparisons made at processor $i-1$ and at processor i are different. Each processor can in constant time match the outcome of the comparison it performed against the outcome of the comparison performed by its right neighbor. The unique processor that detects the subinterval containing the searched key then updates the shared information on the search interval. All the processors next read this information, and proceed to search within the new interval. The complete algorithm is given below. The syntactic construct

for i in S pardo $P(i)$ odrap

indicates parallel execution of the statements $P(i)$ for each value of i in the set S . Variables declared within a parallel loop are private to the processor executing this instance of the loop.

```

SEARCH ( $y, \vec{x}, \text{bot}, \text{top}$ )
/*Search for key  $y$  in sorted list  $x_{\text{bot}}, \dots, x_{\text{top}}$ .
We assume that  $x_{\text{bot}} < y < x_{\text{top}}$ .
Initially  $\text{bot} = 0, \text{top} = n + 1, x_{\text{bot}} = -\infty, x_{\text{top}} = \infty$ */
cons
   $p$ ;          /*Number of processors*/
var
  newbot, newtop;
   $c_1, \dots, c_{p+1}$ ; /*Vector storing outcomes of comparisons*/
begin
   $c_{p+1} := 1$ ;
while ( $\text{top} > \text{bot} + 1$ ) do
  for  $j$  in  $[1, p]$  pardo
    var
       $i$ ;
    /*Compute index of key to be compared*/
     $i := \text{bot} + j * (\text{top} - \text{bot}) / (p + 1)$ ;
    /*Compare and store outcome*/
     $c_j := \text{if } y > x_i \text{ then } 0 \text{ else } 1$ 
    odrap;
    /*Compute next interval*/
     $\langle \text{newbot}, \text{newtop} \rangle := \langle \text{bot}, (\text{top} - \text{bot}) / (p + 1) \rangle$ ;
  for  $j$  in  $[1, p]$  pardo
    if  $c_j < c_{j+1}$  then
       $\langle \text{newbot}, \text{newtop} \rangle := \langle \text{bot} + j * (\text{top} - \text{bot}) / (p + 1),$ 
         $\text{bot} + (j + 1) * (\text{top} - \text{bot}) / (p + 1) \rangle$ 
    odrap;
     $\langle \text{bot}, \text{top} \rangle := \langle \text{newbot}, \text{newtop} \rangle$ 
  od;
return ( $\text{bot}$ )
end

```

Note that the full power of concurrent reads is not needed to implement this algorithm. It is sufficient to have a shared memory machine with broadcasting ability: One (fixed) processor is able to broadcast in constant time one item of information to all the other processors.

Parallel search in the continuous case was studied by Gal and Miranker in [6], where a parallel version of the bisection algorithm for root finding is given (the problem of processors coordination is ignored there). An adversary argument is used there to show that the policy of splitting at each stage the interval of possible values into equal length subintervals is optimal. The same argument applies to the discrete case, and implies the following result.

THEOREM 2.2. *Let P be a parallel algorithm that solves the range search problem for a table of size n , such that at each step at most p values from the table are accessed. Then the algorithm executes in the worst case at least $\lg(n+1)/\lg(p+1)$ steps.*

Thus the complexity of searching by comparisons a table of size n with p processors is $\Theta(\lg(n+1)/\lg(p+1))$, if concurrent reads are supported, both for parallel machines that support concurrent writes and for parallel machines that do not support concurrent writes.

Concurrent reads may occur at several places in the algorithm given in Theorem 2.1:

- (i) If the search interval is small, then a key from the searched table may be accessed by more than one processor.
- (ii) All the processors share the searched key.
- (iii) After each iteration, all the processors read concurrently the bounds of the new search interval.

The first type of concurrent reads may be avoided by more careful programming. The concurrent accesses to the searched key may be avoided by initially distributing it to all the processors in $O(\lg p)$ steps. If concurrent reads are not supported, then the broadcasting of the next search interval at the end of each iteration will require $\Omega(\lg p)$ steps, and the running time of the algorithm for fixed p will be $\Omega(\lg n - \lg p)$, with practically no gain obtained from parallelism. It turns out that this is indeed the best performance that can be achieved for searching on a parallel machine that does not support concurrent accesses to the same location in memory: $\Omega(\lg n - \lg p)$ steps are required, even if we do not account for the distribution of the searched key.

3. Lower bounds for discrete root finding. We shall prove the $\Omega(\lg n - \lg p)$ lower bound for the particular case of the range finding problem where the value of the searched key is fixed. This restricted problem can be reformulated as follows. We denote by \vec{w}_i^n the binary sequence consisting of i zeros followed by $n - i$ ones. The index n will be omitted when it can be inferred from the context.

Discrete root finding (for a sequence of length n). Given a monotonic binary sequence $\langle x_1, \dots, x_n \rangle$ count the number of zeros in it, i.e. find the index i such that $\langle x_1, \dots, x_n \rangle = \vec{w}_i^n$.

We first give a more formal definition of the paracomputer model. A paracomputer consists of p processors P_1, \dots, P_p , q registers R_1, \dots, R_q , an input set X , a set of processor states S , a set of register symbols $V \supset X$, a time bound T , a subset of n registers I_1, \dots, I_n that are used for input, and a subset of k registers O_1, \dots, O_k that are used for output.

With each processor P_i are associated the following (partial) functions.

$\alpha_i: S \rightarrow \{1, \dots, q\} \times \{R, RW\}$ —the access function. The first component of $\alpha_i(s)$ yields the index of the register accessed by P_i when in state s ; the second component indicates whether the access is a read (R) or a read and write (RW) operation. We assume w.l.g. that each processor accesses at each step a shared memory location.

$\omega_i: S \rightarrow V$ —the write function. $\omega_i(s)$ yields the symbol written by P_i , when in state s .

$\delta_i: S \times V \rightarrow S$ —the state transition function. If P_i in state s accesses a register containing v then the new processor state is $\delta_i(s, v)$.

The computation starts with input x_i stored in register I_i , a designated initial symbol $0 \in V$ in each of the remaining registers, and each processor P_i in a designated initial state $s_i \in S$.

Let $\vec{x} = \langle x_1, \dots, x_n \rangle$ be the tuple of inputs to the computation. We denote by $s_i^t(\vec{x})$ the state of processor P_i , and by $c_j^t(\vec{x})$ the content of register R_j , at step t of the computation with input \vec{x} . The dependency on \vec{x} will be omitted when it is obvious from the context.

We have

$$\begin{aligned}
 s_i^0 &= s_i; \\
 c_j^0 &= x_i \text{ if } R_j = I_i, \\
 &= 0 \text{ otherwise;} \\
 s_i^{t+1} &= \delta_i(s_i^t, c_j^t) \text{ if } \alpha_i(s_i^t) = \langle j, R \rangle \text{ or } \alpha_i(s_i^t) = \langle j, RW \rangle; \\
 c_j^{t+1} &= \omega_i(s_i^t) \text{ if } \alpha_i(s_i^t) = \langle j, RW \rangle \\
 &= c_j^t \text{ otherwise.}
 \end{aligned}$$

We require in the EREW model that the first components of $\alpha_i(s_i^t)$ and $\alpha_j(s_j^t)$ be distinct for any $i \neq j$. In the CREW model we require that if $\alpha_i(s_i^t) = \langle k, RW \rangle$, then the first component of $\alpha_j(s_j^t)$ is distinct from k for any $j \neq i$. In both cases the processor states and register contents are well defined.

The outputs of the computation are contained at step T in the k output registers. Thus, a paracomputer Π computes the function $F_\Pi: X^n \rightarrow X^k$ defined by

$$F_\Pi(x_1, \dots, x_n) = c_{j_1}^T(\vec{x}), \dots, c_{j_k}^T(\vec{x}),$$

where R_{j_1}, \dots, R_{j_k} are the k output registers.

Note that we do not restrict the size of the alphabet, or the number of processor states (they may both be infinite).

A function F is *finite* if it has finite range, say $\{0, \dots, k\}$. Decision problems are represented by finite functions. For example, the range search problem is associated with the function $RS(x_1, \dots, x_n, y) = \max \{i: x_i < y\}$, defined on all $(n+1)$ -tuples of distinct elements that fulfill the condition $x_1 < \dots < x_n$. The discrete root finding problem is associated with the function $DRF(x_1, \dots, x_n) = \max \{i: x_i = 0\}$, defined on all n -tuples of elements from the set $\{0, 1\}$ that fulfill the condition $x_1 \leq \dots \leq x_n$.

Let $F: X^n \rightarrow \{0, \dots, k\}$ be a finite function. A paracomputer *computes a unary encoding* of F if it has n input registers and $k+1$ output registers, and at the end of the computation with input \vec{x} the only output register that has been modified is the register with index $F(\vec{x})$. If a paracomputer with time bound T computes the finite function F , then a unary encoding of F can be computed by a paracomputer with time bound $T+1$ and the same number of processors. It will be more convenient to prove lower bounds for unary computations, since we have to consider only the indices of the registers accessed, but not the values stored.

Let Π and Π' be paracomputers with the same number of processors and registers, and the same set of input symbols. Then Π and Π' are *access equivalent* if for each input \vec{x} , each i , and each t , $\alpha_i(s_i^t(\vec{x})) = \alpha'_i(s_i^t(\vec{x}))$. If a unary encoding is used, then two access equivalent paracomputers compute the same function.

Let us introduce the following definitions. The set $[a, b] = \{a, a+1, \dots, b\}$ is called a *segment*; we denote by \bar{n} the segment $[0, n]$. Let $\mathcal{S} = \{S_1, \dots, S_k\}$ be a family of subsets of \bar{n} . We say that r is a *critical point* of \mathcal{S} if there exist a set S_i such that $r-1 \in S_i \Leftrightarrow r \notin S_i$. Let $0 < r_1 < \dots < r_s$ be the critical points of the family \mathcal{S} . The *segment partition defined by \mathcal{S}* consists of the segments $[0, r_1-1], [r_1, r_2-1], \dots, [r_s, n]$. The segment partition defined by \mathcal{S} is the coarsest partition of \bar{n} into segments that refines the partition defined by the 2^k sets $\bigcap_i^k S_i^{\epsilon_i}$, where $\epsilon_i \in \{0, 1\}$, $S^0 = S$, and $S^1 = \bar{n} - S$. In particular, if \mathcal{S} is a partition of \bar{n} , then the segment partition defined by \mathcal{S} is the coarsest partition of \bar{n} into segments that refines the partition \mathcal{S} .

THEOREM 3.1. $\Omega(\lg n - \lg p)$ steps are required to solve the discrete root finding problem for a sequence of length n on an EREW machine with p processors.

Proof. The lower bound results from the lack of a mechanism to distribute instantaneously information throughout the system. In order to prove it we have to trace at each step the "information" represented by the state of each processor and the content of each register in shared memory. We do that at each step for all the possible input values, thus obtaining a "synoptic" description of the possible computations.

The information represented by the state of a processor (the content of a register) consists of the set of input values that could produce this state (content). It is important to note that the information represented by the content of a register may change even if this register is not modified, as the fact that no processor stored a new value is informative in itself. Cook and Dwork show in [4] how such "negative" information can be used at profit.

With each processor (register) is associated at each step a partition of the input symbols, according to the distinct states (values) the processor (register) may assume at this step. The lower bound will be obtained by tracing the evolution of these partitions, and showing that the number of sets in these partitions cannot grow too fast. In fact, we shall not trace the partitions that obtain in an actual computation, but the partitions that would obtain in a computation where there is no "loss of information", i.e. a computation where a processor stores a complete account of the information it has whenever it writes in shared memory. These partitions depend only on the access pattern of the processors.

Rather than counting the number of classes in each partition, it is easier to count the number of critical points of the partition.

Let Π be an EREW paracomputer with p processors, q registers, and time bound T , that computes a unary encoding of the finite function DRF associated with the discrete root finding problem for sequences of length n . We define inductively sets $P(i, t)$, $i = 1, \dots, p$, and $R(j, t)$, $j = 1, \dots, q$, such that $P(i, t)(R(j, t))$ contains all the critical points of the partition defined by processor P_i (register R_j) at step t of the computation.

- (i) $P(i, 0) = \phi$, $i = 1, \dots, p$.
- (ii) $R(j, 0) = \{i\}$ if R_j initially contains the i th input
 $= \phi$ otherwise.
- (iii) $r \in P(i, t)$ iff
 - (a) $r \in P(i, t-1)$, or
 - (b) P_i accesses at step t of the computation on input \vec{w}_r the register R_j , and $r \in R(j, t-1)$.
- (iv) $r \in R(j, t)$ iff
 - (a) $r \in R(j, t-1)$, or
 - (b) P_i modifies at step t of the computation with input \vec{w}_r the register R_j , and $r \in P(i, t-1)$, or
 - (c) P_i modifies at step t of the computation with input \vec{w}_{r-1} the register R_j , and $r \in P(i, t-1)$.

These definitions are illustrated in Fig. 1.

CLAIM.

- (i) If $r \notin P(i, t)$, then $s_i^t(\vec{w}_r) = s_i^t(\vec{w}_{r-1})$.
- (ii) If $r \notin R(j, t)$, then $c_j^t(\vec{w}_r) = c_j^t(\vec{w}_{r-1})$.

Proof. By induction on t . The claim is obvious for $t = 0$. We suppose it holds for $t-1$ and prove it for t .

(i) Since $r \notin P(i, t-1)$, $s_i^{t-1}(\vec{w}_r) = s_i^{t-1}(\vec{w}_{r-1})$. In particular, $\alpha_i(s_i^{t-1}(\vec{w}_r)) = \alpha_i(s_i^{t-1}(\vec{w}_{r-1}))$. Let j be the index of the register accessed. If $r \notin R(j, t-1)$ then, by the

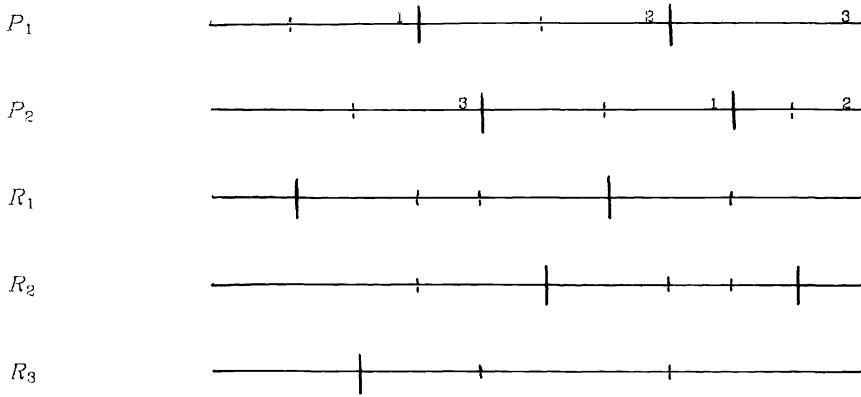


FIG. 1. Evolution of partitions over one step. | Marks critical points at step $t - 1$; | marks new critical points at step t .

inductive assertion, $c_j^{t-1}(\vec{w}_r) = c_j^{t-1}(\vec{w}_{r-1})$, so that

$$s_i^t(\vec{w}_r) = \delta_i(s_i^{t-1}(\vec{w}_r)), c_j^{t-1}(\vec{w}_r) = \delta_i(s_i^{t-1}(\vec{w}_{r-1})), c_j^{t-1}(\vec{w}_{r-1}) = s_i^t(\vec{w}_{r-1}).$$

On the other hand, if $r \in R(j, t - 1)$, then $r \in P(i, t)$.

(ii) As $r \notin R(j, t - 1)$, $c_j^{t-1}(\vec{w}_r) = c_j^{t-1}(\vec{w}_{r-1})$. Suppose that P_i modifies R_j at step t of the computation with input \vec{w}_r . If $r \notin P(i, t - 1)$, then, by the inductive assumption, $s_i^{t-1}(\vec{w}_r) = s_i^{t-1}(\vec{w}_{r-1})$, so that

$$c_j^t(\vec{w}_r) = \omega_i(s_i^{t-1}(\vec{w}_r)) = \omega_i(s_i^{t-1}(\vec{w}_{r-1})) = c_j^t(\vec{w}_{r-1}).$$

On the other hand, if $r \in P(i, t - 1)$ then $r \in R(j, t)$. A similar argument applies if R_j is modified in the computation with input \vec{w}_{r-1} . If R_j is not modified at step t neither in the computation with input \vec{w}_r , nor in the computation with input \vec{w}_{r-1} then

$$c_j^t(\vec{w}_r) = c_j^{t-1}(\vec{w}_r) = c_j^{t-1}(\vec{w}_{r-1}) = c_j^t(\vec{w}_{r-1}).$$

Let $c(t)$ be an upper bound on the number of critical points.

$$c(t) = \sum_i |P(i, t)| + \sum_j \max(0, |R(j, t)| - 1).$$

Initially

$$(3.1) \quad c(0) = 0.$$

If $R_j = O_i$ then R_j is not modified during the computations with input \vec{w}_{i-1} or \vec{w}_{i+1} , and is modified during the computation with input \vec{w}_i . It follows that $i, i + 1 \in R(j, T)$, and

$$(3.2) \quad c(T) \geq n.$$

We shall end our proof by showing that

$$(3.3) \quad c(t) \leq 4c(t - 1) + p.$$

Indeed, (3.1) and (3.3) imply that

$$c(t) \leq \frac{4^t - 1}{3} p,$$

which together with (3.2) yields the inequality

$$n \leq \frac{4^T - 1}{3} p, \quad \text{or} \quad T \geq \log_4 \left(3 \frac{n}{p} + 1 \right) \geq \frac{1}{2} (\lg n - \lg p).$$

Each occurrence of r in a set $P(i, t-1)$ contributes at most one new occurrence of r in a set $R(j, t)$ according to rule (iv.b) and one new occurrence according to rule (iv.c). Thus

$$\sum_j |R(j, t)| \leq \sum_j |R(j, t-1)| + 2 \sum_i |P(i, t-1)|.$$

Since $|R(j, t)| \geq |R(j, t-1)|$ this implies that

$$\sum_j \max(0, |R(j, t)| - |R(j, t-1)|) \leq \sum_j \max(0, |R(j, t-1)| - 1) + 2 \sum_i |P(i, t-1)|.$$

At most one processor may access at step t of a computation on input \vec{w}_r , the register R_j (this is the point where we are using the EREW property). Thus each occurrence of r in a set $R(j, t-1)$ contributes at most one new occurrence of r in a set $P(i, t)$ according to rule (iii.b). Let J_t be the set of registers accessed by some processor at step t of some computation. The number of distinct registers accessed by P_i at step t of some computation is bounded by the number of segments in the segment partition determined by the points in the set $P(i, t-1)$, i.e. by $|P(i, t-1)| + 1$. It follows that $|J_t| \leq \sum_i^p |P(i, t-1)| + p$.

We obtain the inequality

$$\begin{aligned} \sum_{i=1}^p |P(i, t)| &\leq \sum_{i=1}^p |P(i, t-1)| + \sum_{j \in J_t} |R(j, t-1)| \\ &= \sum_i |P(i, t-1)| + \sum_{j \in J_t} (|R(j, t-1)| - 1) + |J_t| \\ &\leq 2 \sum_i |P(i, t-1)| + \sum_j \max(0, |R(j, t-1)| - 1) + p. \end{aligned}$$

Combining the last two inequalities one obtains that

$$\begin{aligned} c(t) &= \sum_i |P(i, t)| + \sum_j \max(0, |C(j, t)| - 1) \\ &\leq 4 \sum_i |P(i, t-1)| + 2 \sum_j \max(0, |C(j, t-1)| - 1) + p \\ &\leq 4c(t-1) + p. \end{aligned} \quad \square$$

We did not use in the proof the fact that concurrent writes are not allowed. Indeed, the lower bound is still valid for an ERCW (exclusive read, concurrent write) parallel machine. It is also valid even if inputs are initially replicated, so that each input value can be accessed concurrently by all the processors.

The last lower bound is optimal. An EREW machine with $n+1$ processors can solve the root finding problem for n keys x_1, \dots, x_n in constant time by comparing in parallel x_i to x_{i+1} , $i=0, \dots, n$ ($x_0=0, x_{n+1}=1$, by definition). This generalizes to an algorithm that solves the problem with p processors P_1, \dots, P_p in $O(\lg(n/p))$ steps as follows. Let $i_j = \lfloor j(n+1)/p \rfloor$. Firstly, each processor P_j checks in parallel whether $x_{i_{j-1}} < x_{i_j}$. Next, the unique processor P_j that found a strong inequality continues to execute a serial bisection algorithm on the list $x_{i_{j-1}+1}, \dots, x_{i_j-1}$.

This simple algorithm can be extended to solve by comparisons the general range searching problem in $O(\lg n - \lg p)$ steps, provided that the searched key can be accessed concurrently by all the processors.

4. Searching with an unbounded number of processors. If the searched key cannot be accessed concurrently by all the processors, then $\Omega(\lg p)$ steps are required to distribute it, thus cancelling the gain obtained from parallelism in the last algorithm. This would seem to imply that $\Omega(\lg n)$ steps are required to solve the range search problem, independently of the number of processors. It turns out, however, that the range search problem can be solved much faster.

THEOREM 4.1. *The range search problem for a table of size n can be solved by an EREW paracomputer with $O(n)$ processors and $O(n)$ registers in $O(\sqrt{\lg n})$ steps.*

Proof. The idea of the algorithm is illustrated in Fig. 2. The search is carried according to a multiway search tree where the branching factor is doubled at each level (Fig. 2b). Such a tree of depth t contains $2^{t(t-1)/2} - 1$ keys, so that searching a table of that size requires t accesses to nodes of the tree. An access to a node of this tree is done in one memory access provided that an encoding of the tuple of keys at that node has been stored in one memory cell. Once the encoding of the keys at the node has been read, the decoding and the subsequent comparisons are performed locally, i.e. at no cost.

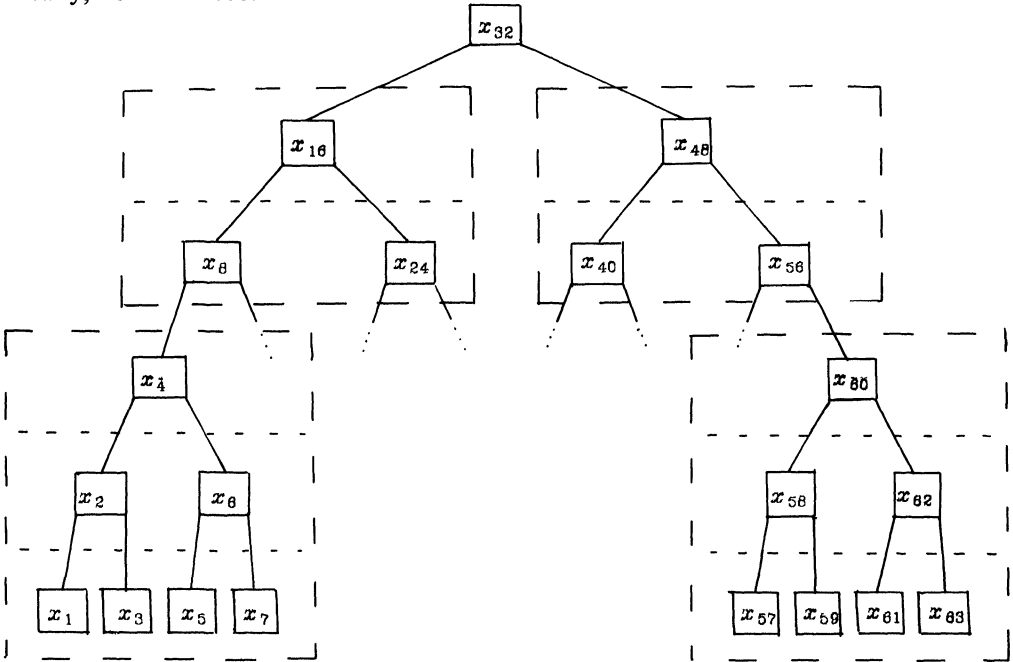


FIG. 2a. Binary search trace and its compression.

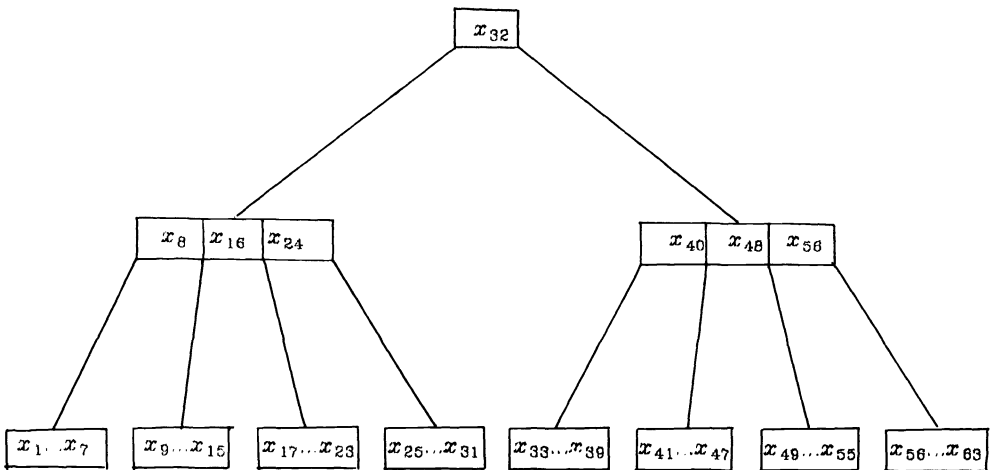


FIG. 2b. Corresponding multiway search tree.

The search will be carried by one processor. Concurrently, the remaining processors will compute and store encodings of tuples of keys. The multiway search tree is obtained by “compressing” the binary search tree: A node at level i of the multiway search tree contains the keys belonging to a complete subtree of depth i , rooted at level $\frac{1}{2}i(i-1)+1$, in the binary tree (Fig. 2a). The processors will compress the binary search tree, increasing by one at each iteration the depth of the subtrees which encodings has been computed.

We describe now this algorithm more formally. Assume w.l.g. that $n+1 = 2^{(t+1)/2}$. Let $a_k = \sum_{i=1}^k i = \frac{1}{2}k(k+1)$. The algorithm consists of t iterations. At the end of iteration i the search has proceeded through i levels of the multiway search tree, that correspond to a_i levels of the binary search tree. Also, encodings have been computed for the keys belonging to each complete subtree of the binary tree that has depth $i+1$ and has its leaves at level a_j , $j = i+1, \dots, t$ (see Fig. 2a). In particular, encodings have been computed for the keys of each subtree of depth $i+1$ rooted at level a_i+1 , i.e. for each tuple of keys belonging to a node of the multiway search tree at level $i+1$. During iteration $i+1$ the searching processor accesses one of these encodings to push the search one level down on the multiway search tree; each of the remaining processors computes an encoding of the keys belonging to a binary tree of depth $i+2$ by combining the encodings for the left and right subtrees which have been computed at the previous iteration, and the key at the root.

At each iteration, each key and each encoding is accessed at most once, and the total number of new encodings computed is less than n . It follows that each iteration can be performed in constant time using $O(n)$ processors, and that only $O(n)$ registers are needed. The total running time is $O(t) = O(\sqrt{\lg n})$. \square

5. Order invariant computations and canonical paracomputers. We prove in this section that algorithms using only comparisons are as powerful as more general algorithms in solving comparison based problems. In order to do so, it will be convenient to work with a paracomputer model where information of input values is clearly distinct from other information.

Let $X^{\leq n}$ be the set of strings over X of length at most n . A paracomputer Π with n inputs is in *canonical form* if it fulfills the following conditions.

- (i) The set of processor states is of the form $X^{\leq n} \times C$ and the set of register symbols is of the form $X^{\leq n} \times D$ (n is the number of inputs to Π). (The input symbol $x \in X$ is identified with the register symbol $\langle x, 0 \rangle$, where $0 \in D$ is a fixed constant.)
- (ii) If $\omega_i(\langle \sigma, s \rangle) = \langle \tau, v \rangle$, then every element of τ occurs in σ (a processor can write an input symbol only if it is present in its “local memory”).
- (iii) If $\delta_i(\langle \sigma, s \rangle, \langle \tau, v \rangle) = \langle \sigma', s' \rangle$ then every element of σ' occurs either in σ or in τ (a processor can store an input symbol in its “local memory” only if it is already there, or if it accessed it from shared memory).

We call C the set of *control symbols*, and D the set of *coordination symbols*.

The behaviour of a canonical paracomputer is conveniently indicated by specifying the indices of the input symbols that are moved. The functions $\bar{\omega}_i$ and $\bar{\delta}_i$ are defined by the following identities.

$$\begin{aligned} \bar{\omega}_i(\langle \sigma_1 \cdots \sigma_k, c \rangle) &= \langle j_1 \cdots j_r, d \rangle, \quad \text{where} \\ \omega_i(\langle \sigma, c \rangle) &= \langle \sigma_{j_1} \cdots \sigma_{j_r}, d \rangle. \\ \bar{\delta}_i(\langle \sigma_1 \cdots \sigma_k, c \rangle, \langle \sigma_{k+1} \cdots \sigma_r, d \rangle) &= \langle j_1 \cdots j_s, e \rangle, \quad \text{where} \\ \delta_i(\langle \sigma_1 \cdots \sigma_k, c \rangle, \langle \sigma_{k+1} \cdots \sigma_r, d \rangle) &= \langle \sigma_{j_1} \cdots \sigma_{j_s}, e \rangle. \end{aligned}$$

The definition of a canonical paracomputer is motivated by the following result.

THEOREM 5.1. *To each paracomputer Π with p processors, q registers, and time bound T , we can associate an access equivalent canonical paracomputer $\mathcal{F}(\Pi)$ with $O(2^T(p+q))$ control and coordination symbols.*

We postpone the proof of this theorem to the next section. It is important to note that the number of control and coordination symbols of $\mathcal{F}(\Pi)$ is independent of the number of input symbols.

The following definitions will make precise the notion of a comparison based computation. Two strings $\vec{x}, \vec{y} \in X^n$ are *order equivalent* ($\vec{x} \equiv \vec{y}$) if $\forall i, j, x_i < x_j \Leftrightarrow y_i < y_j$. A function F defined on X^n is *order invariant* if $\vec{x} \equiv \vec{y} \Rightarrow F(\vec{x}) = F(\vec{y})$. F is order invariant iff each set $F^{-1}(y)$ can be defined by inequalities, that is by Boolean combinations of assertions of the form $x_i < x_j$. The range search function RS is order invariant. Note that an order invariant function has finite range.

A canonical paracomputer is *order invariant* if, when $\sigma \equiv \sigma'$ and $\tau \equiv \tau'$, the following conditions are fulfilled for any c, d and i .

- (i) $\alpha_i(\langle \sigma, c \rangle) = \alpha_i(\langle \sigma', c \rangle)$.
- (ii) $\bar{\omega}_i(\langle \sigma, c \rangle) = \bar{\omega}_i(\langle \sigma', c \rangle)$.
- (iii) $\bar{\delta}_i(\langle \sigma, c \rangle, \langle \tau, d \rangle) = \bar{\delta}_i(\langle \sigma', c \rangle, \langle \tau', d \rangle)$.

Informally, a canonical paracomputer is order invariant if the behavior of each processor depends only on the value of the control and coordination symbols, and the relative order of the input values it has access to, but not on the value of the input symbols themselves. In particular, the computation will follow the same course on two sets of input values where the inputs have the same relative order.

Let $\sigma|_{\vec{y}}^{\vec{x}}$ denote the string obtained by substituting in σ each occurrence of x_i by an occurrence of y_i . We leave to the reader the straightforward proof of the following lemma, which formalizes the last claim.

LEMMA 5.2. *Let Π be an order invariant canonical paracomputer, and let \vec{x} and \vec{y} be order equivalent input vectors. Then the following holds*

- (i) *If $s'_i(\vec{x}) = \langle \sigma, c \rangle$ and $s'_i(\vec{y}) = \langle \sigma', c' \rangle$ then $c = c'$ and $\sigma' = \sigma|_{\vec{y}}^{\vec{x}}$.*
- (ii) *If $c'_i(\vec{x}) = \langle \sigma, d \rangle$ and $c'_i(\vec{y}) = \langle \sigma', d' \rangle$ then $d = d'$ and $\sigma' = \sigma|_{\vec{y}}^{\vec{x}}$.*
- (iii) *$\alpha_i(s'_i(\vec{x})) = \alpha_i(s'_i(\vec{y}))$ for all i and t .*

COROLLARY 5.3. *Let F be an order invariant function defined on $U \subset X$, and let $W \subset X^n$ be a set that contains a representative for each order equivalence class in U (i.e. $\forall \vec{x} \in U \exists \vec{y} \in W$ s.t. $\vec{x} \equiv \vec{y}$). Let Π be an order invariant canonical paracomputer that computes a unary encoding of F for inputs taken from W . Then Π computes a unary encoding of F for any input taken from U .*

Proof. Let \vec{x} be an input vector from U . Let $\vec{y} \in W$ be order equivalent to \vec{x} . Then $F(\vec{x}) = F(\vec{y})$. On the other hand, by Lemma 5.2, the computation of Π on \vec{x} is access equivalent to the computation on \vec{y} , so that a unary encoding of $F(\vec{x})$ is computed. \square

We make use of the following well-known theorem [9].

RAMSEY'S THEOREM. *For any k, m and t there exist a number $N(k, m, t)$ such that the following is true: Let S be a set of size at least $N(k, m, t)$; if we divide the k -element subsets of S into t parts, then at least one part contains all the k -element subsets of some m elements of S .*

THEOREM 5.4. *For each m, p, q and T there exist a number $N = N(m, p, q, T)$ such that the following holds: Let Π be a canonical paracomputer with p processors, q registers, time bound T , and an input set X of size $|X| \geq N$. Then there exists a subset $Y \subset X$ such that $|Y| \geq m$ and Π is order invariant when restricted to inputs from Y .*

Proof. Let $\{x_1 \cdots x_n\}$ and $\{y_1 \cdots y_n\}$ be two n -element sets of input symbols, indexed in increasing order. We say that $x_1 \cdots x_n$ is *congruent* to $y_1 \cdots y_n$ if the

following holds:

If σ and τ are strings from $X^{\cong n}$ with symbols taken from the set $\{x_1 \cdots x_n\}$, $\sigma' = \sigma|_y^{\cong}$, and $\tau' = \tau|_y^{\cong}$, then $\alpha_i(\langle \sigma, c \rangle) = \alpha_i(\langle \sigma', c \rangle)$, $\bar{\omega}_i(\langle \sigma, c \rangle) = \bar{\omega}_i(\langle \sigma', c \rangle)$, and $\bar{\delta}_i(\langle \sigma, c \rangle, \langle \tau, d \rangle) = \bar{\delta}_i(\langle \sigma', c \rangle, \langle \tau', d \rangle)$, for any c, d and i .

It is easy to see that this is indeed an equivalence relation on the n -element subsets of X . The number of distinct values the functions α_i , $\bar{\omega}_i$, and $\bar{\delta}_i$ may assume is bounded by a function of $n, q, |C|$ and $|D|$. It follows that the number G of distinct congruence classes is bounded by a function of $n, p, q, |C|$ and $|D|$. According to Ramsey's theorem, for any s there is a number $N = N(n, G, s)$ such that if $|X| \geq N$ then X contains a subset Y such that $|Y| \geq s$ and all n -element subsets of Y belong to the same congruence class. This entails that, if σ, σ', τ , and τ' are members of $Y^{\cong n}$, $\sigma \equiv \sigma'$ and $\tau \equiv \tau'$ then $\alpha_i(\langle \sigma, c \rangle) = \alpha_i(\langle \sigma', c \rangle)$, $\bar{\omega}_i(\langle \sigma, c \rangle) = \bar{\omega}_i(\langle \sigma', c \rangle)$, and $\bar{\delta}_i(\langle \sigma, c \rangle, \langle \tau, d \rangle) = \bar{\delta}_i(\langle \sigma', c \rangle, \langle \tau', d \rangle)$, for any i, c and d . \square

COROLLARY 5.5. *For each p, q and T there exist a number $N = N(p, q, T)$ such that the following holds: Let F be a finite order invariant function defined on X . Let Π be a canonical paracomputer with p processors, q registers and time bound T , that computes a unary encoding of F . Let $|X| \geq N$. Then there exists an order invariant canonical paracomputer with the same number of processors and registers, and the same time bound that computes F .*

Proof. Let n be the number of variables of F ($n \leq q$). According to Theorem 5.4, if X is large enough, then there exists a set Y such that $|Y| \geq n$ and Π is order invariant when restricted to inputs from Y . Each string from $X^{\cong n}$ is order equivalent to a string from $Y^{\cong n}$.

Let Π' be the canonical paracomputer defined as follows: Π' has the same number of processors and registers, same sets of symbols, and the same time bound as Π ; the access, write and transition functions are defined as follows.

$$\alpha'_i(\langle \sigma, c \rangle) = \alpha_i(\langle \sigma', c \rangle)$$

where $\sigma' \in Y^{\cong n}$ is order equivalent to σ ;

$$\bar{\omega}'_i(\langle \sigma, c \rangle) = \bar{\omega}_i(\langle \sigma', c \rangle)$$

where $\sigma' \in Y^{\cong n}$ is order equivalent to σ ;

$$\bar{\delta}'_i(\langle \sigma, c \rangle, \langle \tau', d \rangle) = \bar{\delta}_i(\langle \sigma', c \rangle, \langle \tau', d \rangle)$$

where $\sigma' \in Y^{\cong n}$ is order equivalent to σ and $\tau' \in Y^{\cong n}$ is order equivalent to τ .

As Π is order invariant on inputs from Y^n , Π' is well defined, and order invariant on all inputs from X^n . Also, the computations of Π' are identical to the computations of Π for inputs taken from Y^n . Thus Π' computes a unary encoding of F on Y^n , and by Corollary 5.3, computes a unary encoding of F on all X^n . \square

6. Lower bounds on searching with an unbounded number of processors. We prove in this section that $\Omega(\sqrt{\lg n})$ lower bound on searching with an unbounded number of processors. The argument consists of three parts. Firstly, we shall complete the proof of Theorem 5.1, thereby reducing the problem to canonical paracomputers. Secondly, the results of the previous section can be used to reduce the problem to canonical order invariant paracomputers. Finally, an argument similar to that used in the proof of Thm. 3.1 yields the lower bound.

Proof of Theorem 5.1. We shall build $\mathcal{F}(\Pi)$ from Π by stipulating that whenever a processor of Π writes onto shared memory, then the corresponding processor of

$\mathcal{F}(\Pi)$ writes onto shared memory a complete account of the information available to it; whenever a processor of Π reads from shared memory, the corresponding processor of $\mathcal{F}(\Pi)$ reads and stores in its local memory (its state) the content of the register accessed. Thus, each processor of $\mathcal{F}(\Pi)$ has at each step sufficient information to simulate the corresponding processor of Π .

In a processor state $\langle \sigma, c \rangle$, σ will contain the input symbols which values "are known" to the processor, and c will represent the knowledge of the processor on the memory accesses that were executed. A similar convention holds for register values.

We shall use *S-expressions* to encode information on memory accesses. $L \in \mathcal{S}(X)$, the set of *S-expressions* over the set X , iff

$$\begin{aligned} L &= \text{NIL}, \quad \text{or} \\ L &\in X, \quad (L \text{ is an atom from } C) \quad \text{or} \\ L &= (L_1 \cdot L_2), \end{aligned}$$

where L_1 and L_2 are *S-expressions* over X . The list $(L_1 \cdots L_k)$ is the *S-expression* $(\cdots (L_1 \cdot L_2) \cdots) \cdot L_k$ (this is the reverse of LISP convention).

Let Π be a paracomputer. $\mathcal{H}(P_i, \vec{x}, t)$, the *history of processor P_i at step t of the computation on input \vec{x}* , and $\mathcal{H}(R_j, \vec{x}, t)$, the *history of register R_j at step t of the computation on input \vec{x}* , are lists defined inductively as follows.

$$\mathcal{H}(P_i, \vec{x}, 0) = \hat{i},$$

where \hat{i} is an *S-expression* with no atoms, encoding the number i ;

$$\begin{aligned} \mathcal{H}(R_j, \vec{x}, 0) &= x_i \quad \text{if } R_j \text{ contains the } i\text{th input} \\ &= \text{NIL}, \quad \text{otherwise.} \end{aligned}$$

If processor P_i accesses register R_j at step t of the computation with input \vec{x} then

$$\mathcal{H}(P_i, \vec{x}, t) = (\mathcal{H}(P_i, \vec{x}, t-1) \mathcal{H}(R_j, \vec{x}, t-1)).$$

If processor P_i modifies register R_j at step t of the computation with input \vec{x} , then

$$\mathcal{H}(R_j, \vec{x}, t) = \mathcal{H}(P_i, \vec{x}, t-1);$$

otherwise

$$\mathcal{H}(R_j, \vec{x}, t) = \mathcal{H}(R_j, \vec{x}, t-1).$$

CLAIM. (i) The value of i and the state $s_i^t(\vec{x})$ of P_i at step t of the computation with input \vec{x} are uniquely determined by $\mathcal{H}(P_i, \vec{x}, t)$.

(ii) The content $c_j^t(\vec{x})$ of R_j at step t of the computation with input \vec{x} is uniquely determined by j and $\mathcal{H}(R_j, \vec{x}, t)$.

Proof. The claim is trivially true for $t=0$. Assume it is valid for $t-1$.

(i) Let $L = \mathcal{H}(P_i, \vec{x}, t)$. The value of i can be determined from the first element of the list L . From L we can extract $\mathcal{H}(P_i, \vec{x}, t-1)$ and determine $s_i^{t-1}(\vec{x})$ and, therefore, the index j of the register accessed by P_i at step t . The history $\mathcal{H}(R_j, \vec{x}, t-1)$ of register R_j at time $t-1$ occurs in L , so that the content $c_j^{t-1}(\vec{x})$ of R_j at time $t-1$ can be determined. But $s_i^{t-1}(\vec{x})$ and $c_j^{t-1}(\vec{x})$ determine $s_i^t(\vec{x})$.

(ii) Let $L = \mathcal{H}(R_j, \vec{x}, t)$. If $L = \text{NIL}$ or $L = x_j$, then no processor wrote on R_j and its content is known. Otherwise $L = \mathcal{H}(P_i, \vec{x}, t'-1)$ where t' is the last step where a processor modified R_j and P_i is the processor that modified R_j at step t' . It is possible to determine from the expression the values of i , and the state $s_i^{t'-1}$ of P_i at step $t'-1$. These determine the next value c_j^t of R_j , which is also the value of R_j at step t .

We define Π' to be a paracomputer with the same number of processors, and registers, the same time bound, and the same set of input symbols as Π . The processor

states and register values of Π' are S -expression with atoms taken from X . The functions of Π' are defined as follows.

$$\alpha'_i(L) = \alpha_i(s) \quad \text{if } s = s'_i(\vec{x}), L = \mathcal{H}(P_i, t, \vec{x});$$

$\alpha'_i(L)$ is undefined otherwise.

$$\omega'_i(L) = L.$$

$$\delta'_i(L_1, L_2) = (L_1 \cdot L_2).$$

The previous claim implies that Π' is access equivalent to Π ; in fact $s'_i(\vec{x}) = \mathcal{H}(P_i, t, \vec{x})$ and $c'_j(\vec{x}) = \mathcal{H}(R_j, t, \vec{x})$ for every i, j, t , and \vec{x} .

Each history expression of Π contains at most n distinct input symbols, and has length at most $O(2^T(p+q))$. Let L be an S -expression with atoms x_1, \dots, x_k . We represent L by the pair

$$\tilde{L} = \langle x_1 \cdots x_k, L|_{\substack{x_1 \\ \vdots \\ x_k}}^{\substack{\hat{i}_1 \\ \vdots \\ \hat{i}_k}} \rangle,$$

where \hat{i} is an atom-free S -expression (distinct from \hat{i}) that encodes the number i . The canonical paracomputer $\mathcal{F}(\Pi)$ is obtained from Π' by replacing each state symbol, and each register content symbol by its above representation. \square

We leave to the reader the proof of the following technical lemma.

LEMMA 6.1. *Let $\mathcal{P}_1, \dots, \mathcal{P}_r$ be segment partitions of \bar{n} . Let \mathcal{Q} be a family of (not necessarily distinct) segments from $\mathcal{P}_1, \dots, \mathcal{P}_r$ with the property that each element in \bar{n} is contained in at most s sets from \mathcal{Q} . Then*

$$|\mathcal{Q}| \leq \sum_{i=1}^r |\mathcal{P}_i| - r + s.$$

THEOREM 6.2. *For any p and q there is a number $N(p, q)$ such that the following holds: If an EREW paracomputer with p processors and q registers solves in time T the range search problem for n inputs taken from a totally ordered set X such that $|X| \geq N$, then $T \geq \sqrt{\lg n} + O(1)$.*

Proof. Let Π be an EREW paracomputer that computes in T steps a unary encoding of the function $RS: X^n \rightarrow \bar{n}$ that is associated with the range search problem. We can assume w.l.g., by Corollary 5.5, that Π is a canonical, order invariant paracomputer. Let us pick $2n+1$ elements from X , $b_0 < a_1 < b_1 < \dots < a_n < b_n$, and consider the behavior of the algorithm on the $n+1$ sets of inputs $\vec{z}_i = \langle a_1, \dots, a_n, b_i \rangle, i = 0, \dots, n$. Note that $RS(\vec{z}_i) = i$.

We follow now the same approach as in the proof of Theorem 3.1. Let $s'_i(\vec{z}_{r-1}) = \langle \sigma, c \rangle$ and $s'_i(\vec{z}_r) = \langle \sigma', c' \rangle$. The only inputs whose relative order in \vec{z}_{r-1} is different from their relative order in \vec{z}_r are y and x_r . Thus, the state of processor P_i distinguishes between input \vec{z}_{r-1} and input \vec{z}_r if the control symbols are distinct ($c \neq c'$), or the set of inputs accessed are distinct ($\sigma' \neq \sigma|_{\vec{z}_{r-1}}$), or neither conditions obtain, but both the values of $y (= b_{r-1} \text{ or } b_r)$ and of $x_r (= a_r)$ are known to P_i (occur in σ and σ'), in which case σ is not order equivalent to σ' . This motivates the following definitions. We define inductively the sets $P(i, t)$ and $R(j, t)$ as follows.

- (i) $P(i, 0) = R(j, 0) = \emptyset$.
- (ii) $r \in P(i, t)$ iff
 - (a) $r \in P(i, t-1)$, or
 - (b) P_i accesses R_j at step t of the computation on input \vec{z}_n and $r \in R(j, t-1)$, or
 - (c) P_i accesses R_j at step t of the computation on input \vec{z}_n , b_r occurs in $s'^{-1}_i(\vec{z}_r)$, and a_r occurs in $c'^{-1}_i(\vec{z}_r)$, or

(d) P_i accesses R_j at step t of the computation on input \vec{z}_r , a_r occurs in $s_i^{t-1}(\vec{z}_r)$, and b_r occurs in $c_j^{t-1}(\vec{z}_r)$.

(iii) $r \in R(j, t)$ iff

(a) $r \in R(j, t-1)$, or

(b) P_i modifies R_j at step t of the computation with input \vec{z}_r and $r \in P(i, t-1)$, or

(c) P_i modifies R_j at step t of the computation with input \vec{z}_{r-1} , and $r \in P(i, t-1)$.

CLAIM. (i) Let $s_i^t(\vec{z}_{r-1}) = \langle \sigma, c \rangle$ and $s_i^t(\vec{z}_r) = \langle \sigma', c' \rangle$. If $r \notin P(i, t)$, then $c = c'$, $\sigma' = \sigma|_{b_r}^{\bar{b}_r}$, and $\sigma' \equiv \sigma$.

(ii) Let $c_j^t(\vec{z}_{r-1}) = \langle \tau, d \rangle$ and $c_j^t(\vec{z}_r) = \langle \tau', d' \rangle$. If $r \notin R(j, t)$ then $d = d'$, $\tau' = \tau|_{b_r}^{\bar{b}_r}$, and $\tau \equiv \tau'$.

The proof of this claim is similar to the proof of the corresponding claim in Theorem 3.1.

Let

$$c(t) = \sum_{i=1}^p |P(i, t)| + \sum_{j=1}^q |R(j, t)|.$$

We have $c(0) = 0$ and $c(T) \geq 2n$. Let us consider now the growth of $c(t)$.

We have the following two facts.

Fact 1. Let f_i be the i th element of the Fibonacci sequence ($f_0 = f_1 = 1$). Then the number of distinct input symbols occurring in the content of a register at step t is bounded by f_{t-1} , and the number of distinct input symbols occurring in the state of a processor at step t is bounded by f_t .

Fact 2. Each input symbol may occur in the states of at most 2^{t-1} processors and the contents of at most 2^{t-1} registers at step t of the computation on a fixed vector of inputs.

Let $K = f_{T-1}$ and $H = 2^{T-1}$. Then each processor state and each register content occurring during a computation of Π contains at most K input symbols, and each input symbol occurs in the states of at most H processors and the contents of at most H registers during the computation on input \vec{z}_r .

It is easily seen that

$$(6.1) \quad \sum_j |R(j, t)| \leq \sum_j |R(j, t-1)| + 2 \sum_i |P(i, t-1)|.$$

The number of points contributed to sets $P(i, t)$ according to rules (ii.1) and (ii.b) is bounded by

$$(6.2) \quad \sum_i |P(i, t-1)| + \sum_j |R(j, t-1)|.$$

It remains to assess the contribution of rules (ii.c) and (ii.d).

Let $\mathcal{P}(i, t)$ be the segment partition associated with $P(i, t)$; let $\mathcal{P}_y(i, t)$ be the set of segments in $\mathcal{P}(i, t)$ corresponding to states of P_i where an input symbol $y = b_r$ occurs; let $\mathcal{R}(j, t)$ be the segment partition associated with $R(j, t)$. For each r there are at most H processors that contain $y = b_r$ in their state at step t of the computation on input \vec{z}_r . It follows, by Lemma 6.1, that

$$\sum_i |\mathcal{P}_y(i, t)| \leq \sum_i (|\mathcal{P}(i, t)| - 1) + H = \sum_i |P(i, t)| + H.$$

Replace each segment $S \in \mathcal{P}_y(i, t-1)$ by the segments $\{S \cap R : R \in \mathcal{R}(j, t-1)\}$, where j is the index of the register accessed by P_i when in the states corresponding to the segment S . The total number of segments thus obtained is bounded by

$$\sum_i |\mathcal{P}_y(i, t-1)| + \sum_j |R(j, t-1)| \leq \sum_i |P(i, t-1)| + H + \sum_j |R(j, t-1)|.$$

Each of these segments contributes at most K new critical points, according to rule (ii.c). Thus, the total number of critical points contributed by rule (ii.c) is bounded by

$$(6.3) \quad K \left(\sum_i |P(i, t)| + \sum_j |R(j, t)| + H \right).$$

A similar argument yields the same bound for the number of new critical points contributed by rule (ii.d).

We obtain from (6.2) and (6.3)

$$(6.4) \quad \sum_i |P(i, t)| \leq (2K + 1) \left[\sum_i |P(i, t-1)| + \sum_j |R(j, t-1)| \right] + 2HK.$$

Inequalities (6.1) and (6.4) imply that

$$\begin{aligned} c(t) &= \sum_i |P(i, t)| + \sum_j |R(j, t)| \\ &\leq (2K + 3) \sum_i |P(i, t-1)| + (2K + 2) \sum_j |R(j, t-1)| + 2HK \\ &\leq (2K + 3)c(t-1) + 2HK. \end{aligned}$$

It follows that

$$2n \leq c(T) \leq 2HK \frac{(2K + 3)^T - 1}{2K + 2} < H(2K + 3)^T,$$

so that

$$(6.5) \quad n \leq \frac{1}{2} H(2K + 3)^T.$$

Substituting back for K and H , we obtain

$$n \leq 2^{T^2 + O(T)},$$

which implies that $T \geq \sqrt{\lg n} + O(1)$. \square

The last lower bound is valid even if we allow concurrent reads from those input registers that contain the searched table. It is only the access to the searched key that has to be restricted.

7. Paracomputers with bounded bandwidth. The $O(\sqrt{\lg n})$ algorithm relies heavily on the fact that the content of one register may encode the values of an arbitrary number of inputs, so that an arbitrary amount of information can be transferred in one read or write operation, and processed in one instruction cycle. This is not a realistic assumption.

We can restrict this model by restricting the type of operations that can be performed on inputs. This is the approach usually followed in the analysis of comparison based algorithms, where it is assumed that inputs are atomic entities that can be only compared. We obtain a "structured" computational model (in the sense used by [3]), which is more amenable to analysis.

Such restriction runs against the basic approach of this paper which is that of assuming powerful computational nodes, but restricted communication ability. We shall instead impose "structure" on the type of items that can be transmitted in one access to memory. We shall assume that a memory register may contain a unique input symbol; it can also contain a communication symbol, taken from a small set. Inputs are transferred atomically, so that an input symbol cannot encode the values of a tuple of input symbols. Formally, let Π be a paracomputer with input set X and set of

register symbols V . Then Π has *memory bandwidth* d if the following two conditions hold:

- (i) $V = X \times D$, where D is a set of d communication symbols.
- (ii) A processor may write a symbol $x \in X$ only if it had read it at previous steps.

A paracomputer has *bounded memory bandwidth* if it has memory bandwidth d , for some finite d .

We shall allow the memory bandwidth d to grow as a function of the problem size n , but assume it is fixed with respect to the size of the input set.

This restriction is still not sufficient to imply an $\Omega(\lg n)$ lower bound. Indeed, it is still possible to represent the values of a tuple of keys by the state of a processor. We obtain

THEOREM 7.1. *The range searching problem for a table of size n can be solved by an EREW paracomputer with $O(n)$ processors and registers and $O(1)$ memory bandwidth in time $O(\lg n / \lg \lg n)$.*

Proof. The algorithm used is similar to that given in Theorem 4.1. Assume w.l.g. that $n = (t+1)! - 1$. The search proceeds according to a multiway search tree of depth $t = O(\lg n / \lg \lg n)$, where nodes at level i contain i keys, and have, therefore, $i+1$ children (see Fig. 3). Such a tree, of depth t , contains $(t+1)! - 1$ keys, so that a table of that size can be searched in t iterations.

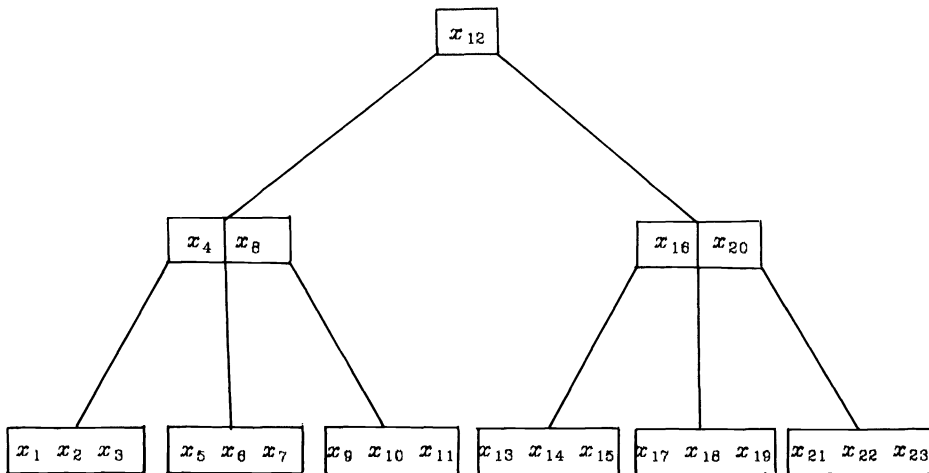


FIG. 3. Multiway search tree for algorithm in Theorem 7.1.

A processor is assigned to each node of that tree. This processor reads at each iteration one key, and stores its value in its local memory. At iteration i the processors assigned to nodes at level i have accessed all the keys at their node. Each processor is also assigned a mailbox. The searched key is initially in the mailbox of the processor assigned to the root. At iteration i the processors assigned to level i nodes access their mailbox. One processor finds the searched key in its mailbox, and compares it to the keys of its node, thereby selecting a node at level $i+1$ where the search proceeds. It then puts the searched key in the mailbox associated with the node selected.

It is easy to see that each iteration can be implemented in constant time, using $O(n)$ processors, $O(n)$ registers, and two communication symbols.

A matching lower bound can be proven, using the methods of the previous section. We leave to the reader the proof of the following analogue to Theorem 5.1.

LEMMA 7.2. To each paracomputer Π of bounded memory bandwidth, with p processors, q registers, time bound T , and set of communication symbols D we can associate an access equivalent canonical paracomputer $\mathcal{F}(\Pi)$ such that the set of processor states is of the form $X^{\cong n} \times C$ and the set of register symbols is of the form $X \times D$, with $|C| \cong O(2^T(p+q+|D|))$.

THEOREM 7.3. For any p, q and d there is a number $N = N(p, q, d)$ such that the following holds: If an EREW paracomputer with p processors, q registers, and memory bandwidth d solves in time T the range search problem for n inputs taken from a totally ordered set X such that $|X| \cong N$, then $T \cong \lg n / \lg \lg n + O(1)$.

Proof. The proof is similar to the proof of Theorem 6.2. We can assume w.l.g. that Π is a canonical paracomputer of the form given by Lemma 7.2, and order invariant. Let the $n+1$ input tuples $\tilde{z}_0, \dots, \tilde{z}_n$, the sets $P(i, t)$ and $R(j, t)$, and the sequence $c(t)$ be defined as in Theorem 6.2.

We have the following two facts.

Fact 1. At most t input symbols occur in the state of a processor at step t .

Fact 2. A fixed input symbol may occur in at most 2^{t-1} processor states and 2^{t-1} register contents at step t of a computation on a fixed input.

It follows that inequality (6.5) of Theorem 6.2 is valid with $K = T - 1$ and $H = 2^{T-1}$. We obtain that

$$n \cong \frac{1}{2} H(2K + 3)^T \cong \frac{1}{4} (4T + 2)^T$$

so that $T \cong \lg n / \lg \lg n + O(1)$. \square

We further weaken our computational model by restricting the type of information that a processor may store in its local memory (i.e. its “state”). We assume now that each processor has a fixed number of local registers. Each local register, may store an input symbol. In addition, each processor has a finite state control.

Formally, let Π be a paracomputer of bounded memory bandwidth with input set X , set of states S , and set of communication symbols D . Then Π has *processor bandwidth* (k, c) , if the following conditions hold:

(i) $S = X^k \times C$, where C is a set of c control state symbols.

(ii) Each input symbol that occurs in $\omega_i(\langle \sigma, c \rangle)$ occurs in σ .

(iii) If $\delta_i(\langle \sigma, d \rangle, u) = \langle \sigma', d' \rangle$, then each symbol of σ' occurs either in σ or in u .

The second condition states that the value of a local register at step t is either the value of a local register at step $t - 1$ or a value read from memory. The third condition states that a processor may write an input symbol only if it is stored in one of its local registers. Note that a paracomputer of bounded processor bandwidth is a canonical paracomputer (provided that $k \cong n$).

THEOREM 7.4. For any p, q, c, d, k , and T there is a number $N = N(p, q, c, d, k, T)$ such that the following holds: If an EREW paracomputer with p processors, q registers, memory bandwidth d , processor bandwidth (c, k) , time bound T solves the range search problem for n inputs taken from a totally ordered set X such that $|X| \cong N$, then $T \cong (\lg n / \lg k + O(1))$.

Proof. The same argument that was twice applied works here as well. Inequality (6.5) of Theorem 6.2 is valid with $K = k$ and $H = 2^{T-1}$. We obtain that $n \cong \frac{1}{2} H(2K + 3)^T \cong \frac{1}{4} (4k + 6)^T$, which yields the result. \square

The last result is asymptotically optimal: if processors may store in their local memory k keys, then it is possible to search a table of size n in $O(\lg_{k+1} n + \lg k)$ steps.

8. Conclusion. As mentioned in the introduction, ultracomputers can be seen as a restricted class of EREW paracomputers. Thus, each of the lower bounds is valid

for ultracomputers. Consider a network of processors, each directly connected to all the other ones, such that each processor contains one key from the searched table, and one processor contains the searched key. If each processor has a fixed size local memory, then $\Omega(\lg n)$ communication steps are required to perform a search, even if local computations are allowed for free. If local memory is not restricted in size, but only one input value may be transmitted at a time, then the problem can be solved in $O(\lg n / \lg \lg n)$ communication steps. Finally, if there are no restrictions on the type of information that can be transferred in one communication step, then the problem can be solved in $O(\sqrt{\lg n})$ steps.

There are few methods known to prove lower bounds for parallel algorithms, which are not based on fanin arguments. This paper contributes one such new method. It seems to capture two "real-life" problems encountered while writing parallel programs: it is hard to parallelize algorithms with many test and branch operations; and frequent coordination between concurrent processes may offset any gain obtained from concurrency.

This paper also provides a method to generalize lower bounds obtained for comparison based algorithms to less restricted algorithms. In that, we were inspired by the work of Yao [14]. This method can be useful in other settings as well, and in particular can be used to analyse distributed algorithms [8].

A more natural constraint on information transfer would be to restrict the number of bits that can be stored in one memory cell. We believe that our lower bounds are valid in such model too, but the proofs seem much harder to obtain.

The paracomputer models we presented may suffer a few interesting variations. As noted in § 2, the $O(\lg n / \lg p)$ searching algorithm can be implemented on any EREW shared memory parallel machine where one processor has the ability to broadcast messages to all the other processors in constant time (a BEREW machine?). If all the processors share this broadcasting ability (only one broadcast is allowed at a time), then this algorithm can be implemented even in the absence of shared memory. We have here a model of parallelism, corresponding to a bus-oriented architecture. A similar model was studied by Stout [11].

Another natural variation is to assume that conflicting memory accesses do not result in an error, but rather in a busy signal being returned to all but one of the requests; alternatively one may postulate a queuing scheme at the memory.

In a real parallel machine memory is likely to be organized into modules with exclusive access being enforced at the level of the memory module rather than at the level of the memory cell. This suggests that we consider computational models where the number of shared memory cells is restricted, and where the amount of information that can be transferred in one read or write operation is smaller than the content of a memory cell. The work of Baer, Du and Ladner [1], and of Vishkin and Wigderson [13] is a useful start in the investigation of such systems.

9. Acknowledgments. I would like to thank Clyde Kruskal who provided the initial thrust for this work, and Allan Borodin and Chee Yap for their helpful remarks.

REFERENCES

- [1] J. L. BAER, H. C. DU AND R. E. LADNER, *Binary search in a multiprocessor environment*, IEEE Trans. Comput., C-32 (1983), pp. 667-676.
- [2] A. BORODIN AND J. E. HOPCROFT, *Routing, merging, and sorting on parallel models of computations*, Proc. 14th ACM Symposium on Theory of Computing, 1982, pp. 338-344.

- [3] A. BORODIN, *Structured versus general models in computational complexity*, in Logic and Algorithmic, Monographie no. 30 de l'Enseignement Mathématique, Université de Geneve, 1982.
- [4] S. COOK AND C. DWORK, *Bounds on the time for parallel RAM's to compute simple functions*, Proc. 14th ACM Symposium on Theory of Computing, 1982, pp. 231-233.
- [5] D. M. ECKSTEIN, *Simultaneous memory access*, Res. Rep. TR-79-6, Computer Science Dept., Iowa State Univ., Ames, 1979.
- [6] S. GAL AND W. MIRANKER, *Optimal sequential and parallel search for finding a root*, J. Combin. Theory (A), 23 (1977), pp. 1-14.
- [7] C. P. KRUSKAL, *Upper and lower bounds on the performance of parallel algorithms*, Ph.D. Thesis, Computer Science Dept., New York Univ., New York, 1981.
- [8] NANCY LYNCH, *private communication*.
- [9] F. P. RAMSEY, *On a problem of formal logic*, Proc. London Math. Soc., 2nd ser, 30 (1930), pp. 264-286.
- [10] J. T. SCHWARTZ, *Ultracomputers*, ACM Trans. Programming Languages and Systems, 2 (1980), pp. 484-521.
- [11] Q. F. STOUT, *Mesh-connected computers with broadcasting*, IEEE Trans. Comput., C-32 (1983), pp. 826-830.
- [12] M. SNIR, *On parallel searching*, ACM Symposium on Principles of Distributed Computing, 1982, pp. 242-253.
- [13] U. VISHKIN AND A. WIGDERSON, *Trade-offs between depth and width in parallel computation*, Proc. 24th IEEE Symposium on Foundations of Computer Science, 1983, pp. 146-153, this Journal, 14 (1985), pp. 303-314.
- [14] A. C. YAO, *Should tables be sorted?*, J. Assoc. Comput. Math., 28 (1981), pp. 615-628.

APPROXIMATION OF THE CONSECUTIVE ONES MATRIX AUGMENTATION PROBLEM*

MARINUS VELDHORST†

Abstract. In this publication we will prove a number of negative results concerning the approximation of the NP-complete CONSECUTIVE ONES MATRIX AUGMENTATION problem. We will characterize a large class of simple algorithms that do not find a near optimum augmentation of their input matrices. We will show that there are matrices for which these algorithms find augmentations that are even far from optimal. These results are important for the analysis of a sparse matrix storage scheme.

Key words. consecutive ones property, sparse matrix storage scheme, NP-completeness, approximation algorithms, analysis of algorithms

For many years much research has been done in the design of efficient storage schemes for sparse matrices. Many storage schemes have proposed (cf. [3], [6], [9], [11]). Some of them are rather efficient for each distribution of the nonzero elements over the matrix (cf. [11]). Others are only efficient for specific distributions (cf. [9]). If one has to choose a storage scheme for sparse matrices in a practical problem, the choice will depend e.g. on the operations to be applied to the matrix. In case only matrix-vector products have to be computed the following data structure for sparse matrices may be very efficient:

Use a data structure in which a sparse matrix is stored as a sequence of rows and each row is stored in such a way that all zero elements at both ends will not be stored. We will call this the *rowmat* data structure and Fig. 1 gives an example how it can be used for a specific matrix.

$$A = (\alpha_{ij}) = \begin{pmatrix} 0.0 & 0.0 & 0.0 & 2.0 & 3.0 \\ -1.0 & 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 3.0 & -2.0 & 0.0 \\ 0.0 & 0.0 & -3.0 & 1.0 & 5.0 \\ 0.0 & 3.0 & 0.0 & -1.0 & -2.0 \end{pmatrix} \quad \text{can be stored in}$$

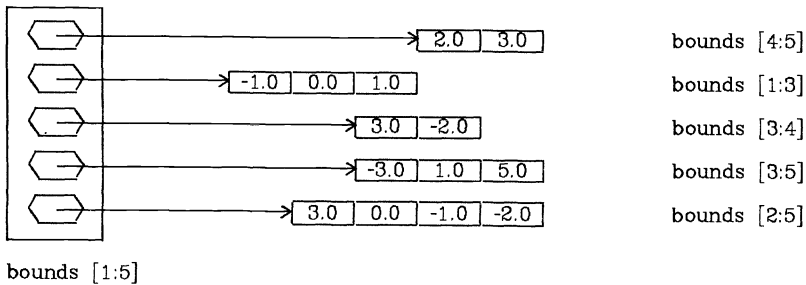


FIG. 1. Storing a matrix in a rowmat data structure.

* Received by the editors March 1, 1983. The research for this publication was done while the author was at the Department of Computer Science, University of Utrecht, the Netherlands. This publication was finished when the author was visiting the Department of Electrical Engineering and Computer Science, Princeton University, Princeton, New Jersey, on a grant from the Netherlands organization for the advancement of pure research (Z.W.O.).

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

In this paper we will discuss the storage optimality of the rowmat data structure, i.e. the amount of storage required if we store matrices in rowmat data structures. Because all nonzero elements must be stored, we only need to consider the number of stored zero elements. The rowmat data structure will not prevent zero elements from being stored in memory (see Fig. 1). However in many applications the columns of the matrix may be permuted. Thus we can look for a column permutation such that the permuted matrix can be stored in a rowmat data structure without storing any nonzero elements. Unfortunately there are matrices for which such a column permutation does not exist.

If columns may be permuted, the storage optimality of the rowmat data structure is closely related to the consecutive ones property for rows of a matrix (cf. [4]). As far as we know the relation of the consecutive ones property to a sparse matrix storage scheme has never been explored before. Although the problem to find a column permutation such that a minimum number of zero elements would be stored is NP-complete (cf. [1]), this does not justify the conclusion that the rowmat data structure should not be used in practice. One way to proceed is to design a polynomial time algorithm that finds for each matrix a column permutation such that a (hopefully) small, but not necessarily minimum, number of zero elements will be stored. These algorithms are called polynomial time approximation algorithms.

In this paper we will concentrate on the design of approximation algorithms. In the first section we will review main results about the consecutive ones property (§ 1.1) and their relation to the storage optimality of the rowmat data structure (§ 1.2). For definitions in the field of complexity theory of algorithms we refer to [5]. In §§ 2, 3 and 4 we will deal with approximation algorithms to find good column permutations. In § 2 we will characterize a large class of simple algorithms: the on-line column insertion algorithms. This section also serves as an introduction for the main theorem. In § 3 we will prove the main theorem that states that any algorithm of the class of on-line column insertion algorithms must give arbitrarily bad approximations for an infinite number of matrices. This means that no on-line column insertion algorithm can guarantee a bounded error factor with regard to the minimum number of stored elements if all column permutations of the input matrix are considered. In the last section we will extend this result to much wider classes of approximation algorithms. In analyzing the rowmat data structure we are not interested in the exact value of a nonzero element, but only in the fact that it is nonzero. Therefore, we will only consider $\{0, 1\}$ -matrices.

1. Consecutive ones property.

1.1. Consecutive ones property and submatrices.

DEFINITION 1.1. (cf. [4]). An $m \times n$ $\{0, 1\}$ -matrix A has the *consecutive ones property for rows* (COR-property) if and only if there is an $n \times n$ permutation matrix P such that the ones of $B = AP = (\beta_{ij})_{1 \leq i \leq m, 1 \leq j \leq n}$ occur consecutively in each row, i.e., for each i ($1 \leq i \leq m$) $\beta_{ij} = 1$ and $\beta_{ik} = 1$ imply $\beta_{ip} = 1$ for all p with $j \leq p \leq k$.

DEFINITION 1.2. Let $A = (\alpha_{ij})$ be an $m \times n$ matrix. A $p \times q$ matrix $B = (\beta_{ij})$ ($p \leq m, q \leq n$) is a *permuted submatrix* of A , if there are sets $I = \{i_1, \dots, i_p\}$ and $J = \{j_1, \dots, j_q\}$ such that $\beta_{hk} = \alpha_{i_h j_k}$ for all h, k ($1 \leq h \leq p, 1 \leq k \leq q$). If $I = \{i_1, i_1 + 1, \dots, i_1 + p - 1\}$, $J = \{j_1, j_1 + 1, \dots, j_1 + q - 1\}$, we say B is a *submatrix* of A and we write $B = A[i_1 : i_1 + p - 1, j_1 : j_1 + q - 1]$. We will denote row i of A by $A[i,]$ and column j of A by $A[, j]$.

THEOREM 1.1 (cf. [12]). A $\{0, 1\}$ -matrix A has the COR-property if and only if no permuted submatrix of A equals one of the matrices given in Fig. 2.

| | | | |
|--|--|--|--|
| $\begin{pmatrix} 1 & 1 & & & & \\ 0 & 1 & 1 & & & \\ 0 & 0 & 1 & 1 & & \phi \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & 0 & \cdot & \cdot & \cdot & \cdot \end{pmatrix}$ <p style="margin-left: 10%;">M_{I_p} with $p \geq 1$</p> | <p>$p+2$ rows</p> <p>$p+2$ columns</p> | $\begin{pmatrix} 1 & 1 & & & & & \\ 0 & 1 & 1 & & & & \\ \cdot & 0 & \cdot & \cdot & & & \phi \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot & \cdot & 0 & 1 & 1 & 0 \\ 1 & 1 & \cdot & \cdot & \cdot & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & \cdot & \cdot & 1 & 1 & 1 & 1 \end{pmatrix}$ <p style="margin-left: 10%;">M_{II_p} with $p \geq 1$</p> | <p>$p+3$ rows</p> <p>$p+3$ columns</p> |
| $\begin{pmatrix} 1 & 1 & & & & & \\ 0 & 1 & 1 & & & & \\ \cdot & 0 & \cdot & \cdot & & & \phi \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & \cdot & \cdot & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & \cdot & \cdot & 1 & 1 & 0 & 1 \end{pmatrix}$ <p style="margin-left: 10%;">M_{III_p} with $p \geq 1$</p> | <p>$p+2$ rows</p> <p>$p+3$ columns</p> | $M_{IV} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$ | |
| | | $M_V = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \end{pmatrix}$ | |

FIG. 2. Minimal matrices not having the COR-property.

If a matrix A contains a permuted M_i , then we shall refer to A as simply containing an M_i .

In the past, research has been done on the problem of detecting the forbidden submatrices (see Fig. 2) in an arbitrary $\{0, 1\}$ -matrix A . A number of questions can be posed:

Does A contain a forbidden permuted submatrix? By Tucker's theorem the existence of a forbidden permuted submatrix is equivalent to A not having the COR-property. Booth and Lueker (cf. [2]) gave an algorithm to test whether A has the COR-property in $O(m+n+f)$ time (f is the number of nonzero elements in A). The algorithm is on line: the rows of A are processed one by one. It starts with the set S of all column permutations of A . Processing row i means that from S all permutations are eliminated which, when applied to row i , do not place the ones in row i in consecutive order. If S gets empty before all rows of A are processed, then A does not have the COR-property. Observe that this algorithm actually finds the largest p such that $A[1:p, 1:n]$ has the COR-property.

Does A contain a forbidden permuted submatrix with at least k rows? This problem has been proved NP-complete (cf. [13]). Even the problems of detecting whether A contains an M_I, M_{II}, M_{III} , respectively, with at least k rows are NP-complete (cf. [14], [13]). The LONGEST PATH problem for graphs can be proven to be polynomially transformable to each of these four problems.

List all forbidden submatrices of A . In [13] an algorithm has been designed which enumerates all forbidden submatrices in time polynomial in the size of A and the number of submatrices listed. The core of this algorithm consists of a subroutine that finds all permuted submatrices that have only nonzero elements on their main diagonal and their first super diagonal. Such a submatrix corresponds to an induced subgraph in a bipartite graph that is a path.

1.2. COR-property and storage optimality. In this section we will investigate the consequences for the optimization of storage if a $\{0, 1\}$ -matrix does not have the COR-property. If we want to store such a matrix in a rowmat data structure, then we shall have to compromise and store embedded zero elements. In the following paragraphs we will deal briefly with two minimization problems. For related minimization problems we refer to [1], [7] and [10].

1.2.1. Augmentation.

DEFINITION 1.3. Let $A = (\alpha_{ij})$ and $B = (\beta_{ij})$ be $m \times n$ $\{0, 1\}$ -matrices. B is a k -augmentation ($k \in \mathbb{N}$) of A if $\alpha_{ij} = 1$ implies $\beta_{ij} = 1$ and moreover there are exactly k different pairs (i_p, j_p) , $1 \leq p \leq k$, such that $\alpha_{i_p, j_p} = 0$ and $\beta_{i_p, j_p} = 1$ ($1 \leq p \leq k$).

$\text{Aug}^0(A) = \{A\}$, $\text{Aug}^k(A) = \{B: B \text{ is a } k\text{-augmentation of } A\}$.

Suppose there is a $B \in \text{Aug}^k(A)$ that has the COR-property. Let P be an $n \times n$ permutation matrix such that the ones of BP occur consecutively in each row. Then, if we store AP in a rowmat data structure, at most k zero elements of A will be stored. Moreover, if k is the least integer such that $\text{Aug}^k(A)$ contains a matrix with the COR-property, then AP requires k zero elements to be stored. This leads to the following problem.

CONSECUTIVE ONES MATRIX AUGMENTATION:

Instance: a $\{0, 1\}$ -matrix A ; an integer $k \geq 0$.

Question: is there a p ($0 \leq p \leq k$) such that $\text{Aug}^p(A)$ contains a matrix with the COR-property?

THEOREM 1.2 (cf. [1]). *The CONSECUTIVE ONES MATRIX AUGMENTATION problem is NP-complete.*

This means that it will be very difficult to design a practical algorithm to find the column permutation that is optimal with regard to the number of stored zero elements. In the next sections we will prove negative results concerning the existence of simple schemes for even finding near optimum column permutations. Observe that for every fixed k one can determine in polynomial time whether a $\{0, 1\}$ -matrix has a k -augmentation with the COR-property. There is only a polynomial number (in the size of the matrix A) of k -augmentations of A , and each k -augmentation can be tested in linear time for the COR-property.

1.2.2. Storing a number of (permuted) submatrices. If an $m \times n$ $\{0, 1\}$ -matrix A does not have the COR-property, then we may try to divide A into a minimum number of submatrices $A[1:p_1, 1:n]$, $A[p_1+1:p_2, 1:n]$, \dots , such that each submatrix has the COR-property. Each submatrix has its own column permutation such that the ones in each row of the submatrix occur consecutively.

If it is not allowed to permute the rows of A , this problem can be solved in polynomial time (apply the algorithm of Booth and Lueker (cf. [2]) several times), but if the rows of A can be permuted, then this problem is NP-complete (cf. [10]). Even to find a maximum set of rows of A that has the COR-property is NP-complete (cf. [1]). Nevertheless we will show that the following problem can be solved in polynomial time (in the size of A):

Let A have rows r_1, \dots, r_m .

Partition $R = \{r_1, \dots, r_m\}$ into sets R_1, \dots, R_p such that

- (1) $|R_i| \geq |R_{i+1}|$ ($1 \leq i \leq p-1$),
- (2) each R_i ($1 \leq i \leq p$) has the COR-property,
- (3) for each i ($1 \leq i \leq p$) and for each row $r \in R_j$ with $j > i$ the set $R_i \cup \{r\}$ does not have the COR-property.

The following algorithm will solve the problem.

ALGORITHM 1.

(initialize $R_i := \{r_i\}$ ($1 \leq i \leq m$);

while there is a j and an $r \in R_j$ such that for some $i < j$ $R_i \cup \{r\}$ has the COR-property

```

do  $R_j := R_j \setminus \{r\}; R_i := R_i \cup \{r\};$ 
   sort  $(R_i)_{1 \leq i \leq m}$  according to decreasing number of rows
od ;
let  $p$  be the greatest index such that  $R_p \neq \emptyset$ 
)
    
```

If this algorithm terminates, R_1, \dots, R_p satisfy the requirements (1)–(3).

PROPOSITION 1.3. *Algorithm 1 terminates and does so within polynomial time.*

Proof. Define:

$$f(R_1, \dots, R_m) = \sum_{i=1}^m |R_i|. \quad (|R_i| \text{ is the number of rows in } R_i).$$

f can only have nonnegative integer values. With each action consisting of a deletion, an addition and an ordering, the value of f decreases. The algorithm has to terminate otherwise f would become negative. When the algorithm has processed the first line, $f(R_1, \dots, R_m) = \frac{1}{2}m(m+1)$ and the outer loop will be executed at most $\frac{1}{2}m(m+1)$ times. It requires at most polynomial time to perform the instructions in the loop-clause. Thus the algorithm will halt within polynomial time in the size of A . \square

2. Examples of approximation algorithms. In the previous section we saw that not every sparse matrix can be stored in a rowmat data structure without storing any zero elements, even if the columns are permuted. Moreover, the problem of finding a column permutation such that a minimum number of zero elements would be stored, turned out to be very hard: the problem is NP-complete. However, these results do not justify the conclusion that the rowmat data structure should not be used in practice. For such a conclusion there should be negative results in the following four directions:

a) We restrict ourselves to a special class C of matrices (which reflects the matrices used in practice) and try to design a polynomial time algorithm that finds for each matrix of C a column permutation such that a minimum number of zero elements will be stored.

b) We try to design a probabilistic (“usually efficient”) algorithm that finds for each matrix a column permutation such that a minimum number of zero elements will be stored. For most matrices such an algorithm should run in polynomial time, but for a (hopefully small) number of matrices it may need exponential time.

c) We try to design a polynomial time algorithm that finds for each matrix a column permutation such that a (hopefully) small, but not necessarily minimum, number of zero elements will be stored. These algorithms are called polynomial time approximation algorithms.

d) Some combination of a), b), c).

In the §§ 2–4 we will concentrate on approximation algorithms. The main result is given in § 3. Section 2 serves as an introduction for this main result.

DEFINITION 2.1. Let $A = (\alpha_{ij})$ be an $m \times n$ $\{0, 1\}$ -matrix.

$$ones(A) = \sum_{i=1}^m \sum_{j=1}^n \alpha_{ij} \quad (\text{the number of nonzero elements of } A),$$

$$store(A) = \sum_{i=1}^m (\max \{j: \alpha_{ij} \neq 0\} - \min \{j: \alpha_{ij} \neq 0\} + 1)$$

with $\max(\emptyset) = 0$ and $\min(\emptyset) = 1$,

$$optstore(A) = \min \{store(AP): P \text{ is an } n \times n \text{ permutation matrix}\}.$$

An $m \times n$ matrix B is a *column permutation* of A if $B = AP$ for some $n \times n$ permutation matrix P . B is an *optimum column permutation* of A if $store(B) = optstore(A)$.

DEFINITION 2.2. Let $A = (\alpha_{ij})$ be an $m \times n$ $\{0, 1\}$ -matrix and B a column permutation of A . Let column i of A be column p_i of B ($1 \leq i \leq n$). We say that α_{i_0, j_0} of A is stored in B if there are j_1 and j_2 with

$$p_{j_1} \leq p_{j_0}, \quad p_{j_2} \geq p_{j_0}, \quad \alpha_{i_0, j_1} = \alpha_{i_0, j_2} = 1.$$

As we have seen in § 1, the problem to find for each matrix A an optimum column permutation B of A , is NP-complete. Here we seek a polynomial time algorithm that finds a column permutation B of A (for each A) such that e.g.

$$\frac{\text{store}(B)}{\text{ones}(A)} \leq c$$

for some fixed constant c , because otherwise there are other more appropriate data structures for sparse matrices.

PROPOSITION 2.1. *If such an algorithm exists, then $c \geq \frac{3}{2}$.*

Proof. Consider the special case of M_{I_n} (cf. § 1.1).

$$\text{optstore}(M_{I_n}) = \text{store}(M_{I_n}) = 2(n+1) + n + 2, \quad \text{ones}(M_{I_n}) = 2n + 4.$$

Therefore, for every column permutation B_n of M_{I_n} we have

$$\frac{\text{store}(B_n)}{\text{ones}(M_{I_n})} \geq \frac{\text{optstore}(M_{I_n})}{\text{ones}(M_{I_n})} = \frac{3n+4}{2n+4}.$$

Note that for $c' < \frac{3}{2}$ there is an n such that $\text{store}(B_n)/\text{ones}(M_{I_n}) > c'$. Hence $c \geq \frac{3}{2}$. \square

However, it is more realistic to compare $\text{store}(B)$ with $\text{optstore}(A)$ than with $\text{ones}(A)$. If $\text{store}(B)$ is close to $\text{optstore}(A)$, then B is considered a "good" column permutation of A . Therefore, to analyze an algorithm X , we will use as a criterion the magnitude of the ratio

$$(4) \quad \frac{\text{store}(B)}{\text{optstore}(A)} \quad \text{with } B \text{ a column permutation of } A \text{ found by } X.$$

Moreover, we are interested in the asymptotic behavior of X .

DEFINITION 2.3. Let $f: \mathbb{N} \rightarrow \mathbb{R}$ be a mapping. Let X be an algorithm that takes $\{0, 1\}$ -matrices as input and that returns a column permutation of its input. X is an *f-approximation algorithm* (for the CONSECUTIVE ONES MATRIX AUGMENTATION problem) if there is an $N \in \mathbb{N}$ such that for all $m \times n$ $\{0, 1\}$ -matrices A ($m, n \geq N$)

$$\frac{\text{store}(X(A))}{\text{optstore}(A)} \leq f(\text{optstore}(A)).$$

As we have seen, we are only interested in c -approximation algorithms with c some constant. It should be nice if there are algorithms of that kind with a running time that is polynomial in the length of their input.

DEFINITION 2.4. A $\{0, 1\}$ -matrix A is said to be *clean* if the following five conditions are satisfied:

- (i) each column of A contains at least one nonzero element,
- (ii) each row of A contains at least two nonzero elements,
- (iii) Each row of A contains at least one zero element,
- (iv) no two rows of A are equal,
- (v) no two columns of A are equal.

DEFINITION 2.5. Let A be an $m \times n$, B an $m \times p$, C an $m \times (n+1)$ and D an $m \times (n+p)$ $\{0, 1\}$ -matrix and let u be a $\{0, 1\}$ -sequence of length m .

$$C = A \text{ concat } u \quad \text{if } C[\ , 1:n] = A \text{ and } C[\ , n+1] = u.$$

$$D = A \text{ concat } B \quad \text{if } D[\ , 1:n] = A \text{ and } D[\ , n+1:n+p] = B.$$

Now we will give two examples of straightforward approximation algorithms that are not c -approximation algorithms for any c .

ALGORITHM 2.

```

proc BESTFIT = (matrix  $A$ )matrix:
co let  $A$  be an  $m \times n$   $\{0, 1\}$ -matrix co
(initialize  $B$  as the  $m \times 0$   $\{0, 1\}$ -matrix;
  for  $k$  from 1 to  $n$ 
    do initialize  $C$  as the  $m \times k$   $\{0, 1\}$ -matrix with only nonzero elements;
      for  $j$  from 1 to  $k$ 
        do matrix  $D = B[ \ , 1:j-1 ]$  concat  $A[ \ , k ]$  concat  $B[ \ , j:k-1 ]$ ;
          if  $store(D) < store(C)$  then  $C := D$  fi
        od;  $B := C$ 
      od;
    return  $B$ 
  )
    
```

BESTFIT processes the columns of A one by one; processing column k means that it will be inserted in B such that the current columns of B do not lose their relative order; column k is inserted in B just where it causes the least number of elements of B (including column k) to be stored.

PROPOSITION 2.2. BESTFIT is not a c -approximation algorithm for any $c \geq 1$.

Proof. We have to prove that for each $N \in \mathbf{N}$ and each $c \in \mathbf{R}$ ($c \geq 1$) there is a $\{0, 1\}$ -matrix A with at least N rows and columns such that

$$\frac{store(BESTFIT(A))}{optstore(A)} > c.$$

Let $N \in \mathbf{N}$, $c \in \mathbf{R}$ ($c \geq 1$); let

$$A_{p,k} = \begin{pmatrix} 0 \dots 0 & 1 \dots 1 & 1 \dots 1 \\ 1 \dots 1 & & 0 \dots 0 \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & 1 \dots 1 & 0 \dots 0 \\ 1 \dots 1 & 0 \dots 0 & 1 \dots 1 \end{pmatrix}$$

\longleftrightarrow
 $p+1$

\longleftrightarrow
 $p+1$

\longleftrightarrow
 k

$p+2$ rows,
 $2p+k+2$ columns,
 $k \geq 2, p \geq 1$.

$optstore(A_{p,k}) = store(A_{p,k}) = (p+1)^2 + (p+1)(p+2) + 2k$. If BESTFIT is applied to $A_{p,k}$, it returns

$$BESTFIT(A_{p,k}) = \begin{pmatrix} 1 \dots 1 & 1 \dots 1 & 0 \dots 0 \\ \cdot & \cdot & 0 \dots 0 & 1 \dots 1 \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 1 \dots 1 & 0 \dots 0 & \cdot & \cdot \\ 0 \dots 0 & 1 \dots 1 & 1 \dots 1 & \cdot \end{pmatrix}$$

$\underbrace{\hspace{1.5cm}}_{p+1} \quad \underbrace{\hspace{1.5cm}}_k \quad \underbrace{\hspace{1.5cm}}_{p+1}$

and

$$\frac{store(BESTFIT(A_{p,k}))}{optstore(A_{p,k})} = \frac{2(p+1)^2 + (p+2)k}{(p+1)^2 + (p+1)(p+2) + 2k} = \frac{2\frac{p+1}{p+2} + \frac{k}{p+1}}{\frac{p+1}{p+2} + 1 + \frac{2k}{(p+1)(p+2)}} \approx \frac{p+2}{2}$$

if k large compared with p . Thus, with p and k large enough, we have $p+2 \geq N$, $2p+2+k \geq N$ and $(p+2)/2 > c$. We conclude that BESTFIT is not a c -approximation algorithm for any $c \in \mathbf{R}$. \square

Observe that the columns of $A_{p,k}$ are sorted by nonincreasing number of nonzero elements. BESTFIT has a rather bad performance because it compares the one column to be inserted with the (maybe many) columns already processed. It may decide to store the zero elements of column $A[\cdot, q]$ (A arbitrary) rather than to change not-stored zero elements of $BESTFIT(A[\cdot, 1:q-1])$ into stored zero elements in $BESTFIT(A[\cdot, q])$. This may happen even when $A[\cdot, q]$ has many zero elements and is identical to many other columns in the matrix A . Consider, for example, column $2p+3$ in the matrix $A_{p,k}$ as used in the proof of Proposition 2.2.

BESTFIT has a tendency to insert the last columns somewhere in the middle of the matrix obtained so far, which may lead to many stored zero elements that would not be stored in the optimum column permutation. Thus Algorithm 3 below in which the columns are ordered by decreasing number of zero elements, may perform better. As for identical columns we have the following lemma.

LEMMA 2.3. *Let A be an $m \times n$ $\{0, 1\}$ -matrix. Then there is an optimum column permutation B of A such that identical columns of B are consecutive in B .*

Proof. Let C be an optimum column permutation of A and suppose not all identical columns of C are consecutive. Thus there are $j_0, j_1, j_2 \in \mathbf{N}$ ($j_0 < j_1 < j_2$) such that the columns j_0 and j_2 of C are identical and the columns j_0 and j_1 of C are not identical. Without loss of generality we can assume that the number k of stored elements of column j_0 is not greater than the number of stored elements of column j_2 . If we delete column j_2 and insert it just beside column j_0 (thus obtaining a $\{0, 1\}$ -matrix C') then exactly k elements of column j_2 of C are stored in C' . Then:

$$store(C') = k + store(C[\cdot, 1:j_2-1] \text{ concat } C[\cdot, j_2+1:n]) \leq store(C).$$

Because C was assumed to be an optimum column permutation of A , we have $store(C') = store(C)$. Thus we can permute the columns of C in such a way that the

number of stored elements will not increase and identical columns will be consecutive. This gives us another optimum column permutation of A . \square

DEFINITION 2.6. Let A be an $m \times n$ matrix. We say that A has a *row partition* $K^r = (0 = k_0^r < k_1^r < \dots < k_p^r = m)$ and a *column partition* $K^c = (0 = k_0^c < k_1^c < \dots < k_q^c = n)$ if A can be partitioned into $(A_{ij})_{1 \leq i \leq p, 1 \leq j \leq q}$ such that for all i, j ($1 \leq i \leq p, 1 \leq j \leq q$) $A_{ij} = A[k_{i-1}^r + 1 : k_i^r, k_{j-1}^c + 1 : k_j^c]$. In this case $rowstrip(A, i, K^r) = A[k_{i-1}^r + 1 : k_i^r,]$ and $colstrip(A, j, K^c) = A[, k_{j-1}^c + 1 : k_j^c]$.

DEFINITION 2.7.

(i) An $m \times n$ matrix A is a *block diagonal matrix* with k blocks if it can be partitioned into $A = (A_{ij})_{1 \leq i \leq p, 1 \leq j \leq q}$ such that $\min(p, q) = k$ and $A_{ij} = 0$ for all $i \neq j$ ($1 \leq i \leq p, 1 \leq j \leq q$). *Block i* of A is the matrix A_{ii} ($1 \leq i \leq k$).

(ii) An $m \times n$ matrix $A = (\alpha_{ij})$ is a *permuted block diagonal matrix* with k blocks if there are $m \times m$ and $n \times n$ permutation matrices P and Q such that PAQ can be partitioned into a block diagonal matrix with k blocks.

Without proof we state:

LEMMA 2.4. Let A be an $m \times n$ $\{0, 1\}$ -matrix with row partition $K^r = (k_0^r, \dots, k_p^r)$. Then:

$$store(A) = \sum_{i=1}^p store(rowstrip(A, i, K^r)).$$

If A is a block diagonal matrix with row partition K^r and column partition $K^c = (0 = k_0^c, \dots, k_q^c)$, then:

(i) for each i ($1 \leq i \leq \min(p, q)$):

$$store(rowstrip(A, i, K^r)) = store(colstrip(A, i, K^c)) = store(A_{ii}),$$

$$\begin{aligned} \text{(ii)} \quad optstore(A) &= \sum_{i=1}^p optstore(rowstrip(A, i, K^r)) \\ &= \sum_{i=1}^q optstore(colstrip(A, i, K^c)) \\ &= \sum_{i=1}^{\min(p,q)} optstore(A_{ii}). \end{aligned}$$

ALGORITHM 3.

proc BESTFITDECR = (matrix A)matrix:

(Let B be a column permutation of A such that the number of zero elements per column in B is nonincreasing. Moreover, identical columns of A are consecutive in B .

Then perform BESTFIT(B) with the modification that identical columns are inserted simultaneously and are kept in consecutive order in the result of BESTFIT(B)

)

BESTFITDECR processes the matrices $A_{p,k}$ ($p \geq 2$) of the proof of Proposition 2.2 very well:

$$\frac{store(BESTFITDECR(A_{p,k}))}{optstore(A_{p,k})} = \frac{store(A_{p,k})}{optstore(A_{p,k})} = \frac{store(A_{p,k})}{store(A_{p,k})} = 1.$$

BESTFIT works rather well for a block diagonal matrix A : the columns of each columnstrip of A remain consecutive in BESTFIT(A). However, BESTFIT can have

a bad performance for permuted block diagonal matrices, which will not necessarily be improved by BESFITDECR.

PROPOSITION 2.5. BESTFITDECR is not a c -approximation algorithm for any $c \geq 1$.

Proof. We have to prove that for each $N \in \mathbf{N}$ and each $c \in \mathbf{R}$ ($c \geq 1$) there is an $m \times n$ $\{0, 1\}$ -matrix A ($m, n \geq N$) such that

$$\frac{\text{store}(\text{BESTFITDECR}(A))}{\text{optstore}(A)} > c.$$

Let $N \in \mathbf{N}$, $c \in \mathbf{R}$ ($c \geq 1$). Let A_k be a block diagonal matrix with $2k$ blocks A_{ii} with

$$A_{ii} = M_{I_i} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \quad (1 \leq i \leq 2k).$$

Let $B_k = (\beta_{ij})$ be the $6k \times 7k$ $\{0, 1\}$ -matrix with $B_k[\ , 1:6k] = A_k$ and for each i ($1 \leq i \leq k$) $\beta_{3(i-1)+1, 6k+i} = \beta_{3k+3(i-1)+1, 6k+i} = 1$ (see Fig. 3). B_k is a permuted block diagonal

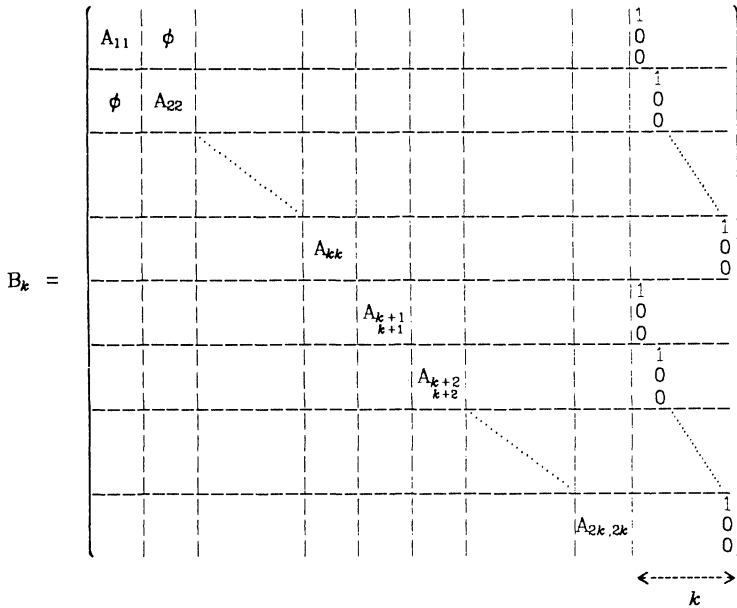


FIG. 3. Matrices used in the proof of Proposition 2.5.

matrix with k blocks. All columns of B_k are different and each column contains two nonzero elements. Let $K^c = (0, 3, 6, \dots, 6k, 7k)$. BESTFITDECR returns the columns of B_k in the ordering:

$$\begin{aligned} & \text{colstrip}(B_k, 2k, K^c), \quad B_k[\ , 7k], \quad \text{colstrip}(B_k, 2k-1, K^c), \quad B_k[\ , 7k-1], \quad \dots, \\ & B_k[\ , 6k+2], \quad \text{colstrip}(B_k, k+1, K^c), \quad B_k[\ , 6k+1], \quad \text{colstrip}(B_k, k, K^c), \\ & \text{colstrip}(B_k, k-1, K^c), \quad \dots, \quad \text{colstrip}(B_k, 1, K^c). \end{aligned}$$

Thus

$$\begin{aligned} & \text{store}(\text{BESTFITDECR}(B_k)) \\ &= \sum_{i=1}^k \text{store}(A_{ii}) + \sum_{i=k+1}^{2k} (\text{store}(A_{ii}) + 1) + \text{ones}(B_k[\quad, 6k+1:7k]) \\ & \qquad \qquad \qquad + \sum_{i=1}^k (3(k-1) + i - 1) \\ &= \frac{k}{2}(7k+27). \end{aligned}$$

On the other hand, $\text{optstore}(B_k) = 2k + \sum_{i=1}^{2k} \text{optstore}(A_{ii}) = 2k + 14k = 16k$. With k large enough we have

$$\frac{\text{store}(\text{BESTFITDECR}(B_k))}{\text{optstore}(B_k)} = \frac{7k^2 + 27k}{32k} > c.$$

Hence, BESTFITDECR is not a c -approximation algorithm for any $c \geq 1$. \square

DEFINITION 2.8. Let A be an $m \times n$ $\{0, 1\}$ -matrix and u a $\{0, 1\}$ -sequence of length m . An $m \times (n+1)$ $\{0, 1\}$ -matrix B is an *insertion matrix for u in A* if for some $y \in \mathbb{N}$ ($0 \leq y \leq n$)

$$B = A[\quad, 1:y] \text{ concat } u \text{ concat } A[\quad, y+1:n].$$

DEFINITION 2.9. Let X be an algorithm that takes $\{0, 1\}$ -matrices as input and that returns a column permutation of its input. X is an *on-line column insertion algorithm* if for each $m \times (n+1)$ $\{0, 1\}$ -matrix A , $X(A)$ is an insertion matrix for $A[\quad, n+1]$ in $X(A[\quad, 1:n])$.

PROPOSITION 2.6. BESTFITDECR is not an on-line column insertion algorithm.

Proof. Let

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}, & \text{BESTFITDECR}(A) &= \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}, \\ & & \text{BESTFITDECR}(A[\quad, 1:3]) &= \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \end{aligned}$$

and therefore, $\text{BESTFITDECR}(A)$ is not an insertion matrix for $A[\quad, 4]$ into $\text{BESTFITDECR}(A[\quad, 1:3])$. \square

We could have defined the notion of an on-line column insertion algorithm in another way.

DEFINITION 2.10. Let X be an algorithm that takes $\{0, 1\}$ -matrices as input and that returns a column permutation of its input. X has *property P* if for every $m \times n$

$\{0, 1\}$ -matrix A and every $k \in \mathbb{N}$ ($1 \leq k \leq n$), $X(A)$ and $X(A[\ , 1:k])$ satisfy (5):

- (5) let column i ($1 \leq i \leq n$) of A be column p_i of $X(A)$; let column i ($1 \leq i \leq k$) of $A[\ , 1:k]$ be column q_i of $X(A[\ , 1:k])$; for all i and j ($1 \leq i, j \leq k$) we have

$$p_i < p_j \text{ if and only if } q_i < q_j.$$

LEMMA 2.7. X is an on-line column insertion algorithm if and only if X has property P .

Proof.

\Leftarrow : Suppose X has property P . Let A be an $m \times n$ $\{0, 1\}$ -matrix and $k = n - 1$. Let $(p_i)_{1 \leq i \leq n}$ and $(q_i)_{1 \leq i \leq n-1}$ as in (5) and $y = p_n$. Then for all i with $q_i \leq y - 1$ we have $p_i = q_i$ and for all i with $q_i \geq y$, $p_i = q_i + 1$. Thus $X(A)$ is an insertion matrix for $A[\ , n]$ in $X(A[\ , 1:n-1])$. This holds for every matrix A and therefore, X is an on-line column insertion algorithm.

\Rightarrow : Suppose X is an on-line column insertion algorithm. Let A be an $m \times n$ $\{0, 1\}$ -matrix and $k \in \mathbb{N}$ ($1 \leq k \leq n$). Consider $A[\ , 1:h]$ with $k \leq h \leq n$. Let column i ($1 \leq i \leq h$) of $A[\ , 1:h]$ be column $p_{h,i}$ of $X(A[\ , 1:h])$. $X(A[\ , 1:h+1])$ is an insertion matrix for $A[\ , h+1]$ in $X(A[\ , 1:h])$, thus:

$$\text{for all } i, j \text{ (} 1 \leq i, j \leq h \text{)} \ p_{h,i} < p_{h,j} \text{ if and only if } p_{h+1,i} < p_{h+1,j}.$$

With induction it is easy to prove that

$$\text{for all } i, j \text{ (} 1 \leq i, j \leq k \text{)} \ p_{k,i} < p_{k,j} \text{ if and only if } p_{n,i} < p_{n,j}.$$

Hence, X has property P . \square

Using this lemma, one can for every $m \times n$ $\{0, 1\}$ -matrix A determine $X(A[\ , 1:k])$ from $X(A)$ for every $k \leq n$, since the relative ordering of columns of $A[\ , 1:k]$ as they appear in $X(A)$ is inherited.

COROLLARY 2.8. Let $A = (\alpha_{ij})$ be an $m \times n$ $\{0, 1\}$ -matrix and X an on-line column insertion algorithm. If for some $k \leq n$ and i_0, j_0 ($1 \leq i_0 \leq m, 1 \leq j_0 \leq k$) α_{i_0, j_0} (considered as an element) of $A[\ , 1:k]$ is stored in $X(A[\ , 1:k])$, then α_{i_0, j_0} (considered as an element) of A is stored in $X(A)$.

3. A negative result. Now the question arises whether there is some on-line column insertion algorithm that is a polynomial time c -approximation algorithm for the CONSECUTIVE ONES MATRIX AUGMENTATION problem. In Theorem 3.2 we will answer this question negatively. But Corollary 3.3 states more. To obtain a negative answer, it will be sufficient to show that for each on-line column insertion algorithm X and for each $c \geq 1$ and each $N \in \mathbb{N}$ there is an $m \times n$ matrix $A_{X,c,N}$ ($m, n \geq N$) such that

$$\frac{\text{store}(X(A_{X,c,N}))}{\text{optstore}(A_{X,c,N})} > c$$

which proves that, in fact no on-line column insertion algorithm can be a c -approximation algorithm (for some fixed c) at all. Theorem 3.2 states that $A_{X,c,N}$ can be chosen to be of arbitrary size (for fixed X and c) and Corollary 3.3 states that for fixed c and N $A_{X,c,N}$ can be chosen such that the following conditions are satisfied:

- (i) the number of rows and columns of $A_{X,c,N}$ does not depend on the on-line column insertion algorithm X ,
- (ii) let X and X' be two on-line column insertion algorithms. Then $A_{X,c,N}$ and $A_{X',c,N}$ are equal possibly except for their last columns.

In order to prove this we will use a block diagonal matrix A of $k m \times m$ blocks. The number of elements in a block is bounded from above by m^2 and in a rowstrip by km^2 . We will use blocks that all have an *optstore* value bounded by $3m$. A is chosen such that for each of its column permutations B it is possible that many elements of a rowstrip of B will be stored if one extra column is inserted in B .

THEOREM 3.1. *Let $N \in \mathbb{N}$, $c \in \mathbb{R}$ ($c \geq 1$). Then there is a clean square matrix A with $n \geq N$ rows such that for each column permutation B of A we have:*

if $store(B)/optstore(A) \leq c$ then there is a $\{0, 1\}$ -sequence u_B of length n such that for each matrix C that is an insertion matrix for u_B in B , $store(C)/optstore(A \text{ concat } u_B) > c$.

Proof. Let $N \in \mathbb{N}$ and $c \in \mathbb{R}$ ($c \geq 1$). Let $f: \mathbb{N} \rightarrow \mathbb{N}$ be any function such that

- (i) $\lim_{k \rightarrow \infty} f(k) = \infty$,
 - (ii) there are a $c' > 0$ and a $K' \in \mathbb{N}$ such that for all $k \geq K'$, $f(k)/k < \frac{1}{3} - c'$.
- For example: $f(k) = \lceil k/4 \rceil$, $f(k) = \lceil \log_2 k \rceil$, etc.

Choose $k' \in \mathbb{N}$ ($k' \geq 3$) such that for all $k \geq k'$ we have

$$(6) \quad f(k) > c \quad \text{and} \quad \frac{k - 3f(k)}{3k} (1 + 2f(k)) > c.$$

Such a k' exists. Let $k \geq k'$. Choose $m' \in \mathbb{N}$ ($m' \geq 3$) such that for all $m \geq m'$ we have

$$mk \geq N \quad \text{and} \quad \frac{k(3m - 2) + m + m^2 f(k)}{k(3m - 2) + m^2 - m + 2} > c.$$

Let $m \geq m'$. Thus:

$$(7) \quad \begin{aligned} &mk \geq N, \quad m \geq 3, \quad k \geq 3, \quad \frac{k(3m - 2) + m + m^2 f(k)}{k(3m - 2) + m^2 - m + 2} > c, \\ &\frac{1 + 2f(k)}{3} > c \quad \text{and} \quad \frac{k - 3f(k)}{3k} (1 + 2f(k)) > c. \end{aligned}$$

Observe that the fifth inequality of (7) follows from the second inequality of (6).

Let A be a block diagonal matrix with k blocks M_1, \dots, M_k and

$$(8) \quad M_i = M_{I_{m-2}} \quad (1 \leq i \leq k).$$

Let $K = (0, m, 2m, \dots, mk)$ be the corresponding row and column partition. Then we have:

$$(9) \quad optstore(M_i) = 3m - 2 \quad (1 \leq i \leq k), \quad optstore(A) = (3m - 2)k.$$

Let B be any column permutation of A with column j of A a column p_j of B ($1 \leq j \leq mk$).

Let

$$(10) \quad \begin{aligned} pmin(i) &= \min \{ p_j : (i - 1)m \leq j \leq im \}, \quad (1 \leq i \leq k) \quad \text{and} \\ pmax(i) &= \max \{ p_j : (i - 1)m \leq j \leq im \}, \quad (1 \leq i \leq k). \end{aligned}$$

Claim.

$$(11) \quad store(rowstrip(B, i, K)) \geq m + 2(pmax(i) - pmin(i)) \quad \text{for all } 1 \leq i \leq k.$$

Proof of Claim. Consider column t in $rowstrip(B, i, K)$ with $pmin(i) \leq t \leq pmax(i)$. It is either a column of M_i or it is a zero column. When it is a zero column, we have

$pmin(i) < t < pmax(i)$ and at least two of its zero elements are stored in $rowstrip(B, i, K)$. There are exactly $pmax(i) - pmin(i) - m + 1$ zero columns in $rowstrip(B, i, K)$ between the columns $pmin(i)$ and $pmax(i)$. Hence

$$store(rowstrip(B, i, K)) \geq optstore(M_i) + 2(pmax(i) - pmin(i) - m + 1).$$

With (9) this proves the claim.

We are interested in the values of $pmin(i) - pmax(j)$.

Let $q_0 = \min \{pmax(i) : 1 \leq i \leq k\}$, $q_1 = \max \{pmin(i) : 1 \leq i \leq k\}$.

Thus exactly one columnstrip of A has all its columns at the left of column $q_0 + 1$ in B and exactly one columnstrip of A has all its columns at the right of column $q_1 - 1$ in B . There are three cases.

Case 1. $q_1 \leq q_0 - mf(k)$.

Claim 1. With m and k satisfying (7) and $q_1 \leq q_0 - mf(k)$ we have

$$\frac{store(B)}{optstore(A)} > c.$$

Proof of Claim 1. $q_1 \leq q_0 - mf(k)$. Thus:

$$\text{for all } i \quad pmax(i) \geq pmin(i) + mf(k) \quad (i \leq i \leq k).$$

Using (11) this results in $store(rowstrip(B, i, K)) \geq m + 2mf(k)$ so that (with (7) and (9)):

$$\frac{store(B)}{optstore(A)} \geq \frac{mk + 2mkf(k)}{(3m - 2)k} \geq \frac{m + 2mf(k)}{3m} = \frac{1 + 2f(k)}{3} > c.$$

Hence Claim 1 is proved and Case 1 satisfies the theorem.

Case 2. $|q_1 - q_0| < mf(k)$.

Claim 2. With m and k satisfying (7) and $|q_1 - q_0| < mf(k)$ we have

$$\frac{store(B)}{optstore(A)} > c.$$

Proof of Claim 2. First assume $mf(k) < \min(q_0, q_1) < \max(q_0, q_1) < mk - mf(k)$. We divide B into five parts, as shown in Fig. 4.

$$\begin{aligned} R_1 &= B[\quad , \quad 1 \quad : \min(q_0, q_1) - mf(k) - 1], \\ R_2 &= B[\quad , \quad \min(q_0, q_1) - mf(k) \quad : \quad \min(q_0, q_1) - 1 \quad], \\ R_3 &= B[\quad , \quad \min(q_0, q_1) \quad : \quad \max(q_0, q_1) \quad], \\ R_4 &= B[\quad , \quad \max(q_0, q_1) + 1 \quad : \quad \max(q_0, q_1) + mf(k) \quad], \\ R_5 &= B[\quad , \quad \max(q_0, q_1) + mf(k) + 1 \quad : \quad mk \quad]. \end{aligned}$$

R_2, R_3 and R_4 together contain at most $3mf(k)$ columns. B contains mk columns so that R_1 and R_5 together at least $mk - 3mf(k)$ columns.

Let

$$I_{MIN} = \{i : pmin(i) < \min(q_0, q_1) - mf(k)\} \quad \text{and}$$

$$I_{MAX} = \{i : pmax(i) > \max(q_0, q_1) + mf(k)\}.$$

Then $|I_{MIN} \cup I_{MAX}| \geq k - 3f(k)$. For each $i \in I_{MIN} \cup I_{MAX} : pmax(i) - pmin(i) \geq mf(k)$.

With (11) this gives the result:

$$\text{for all } i \in I_{MIN} \cup I_{MAX} : \quad store(rowstrip(B, i, K)) \geq m + 2mf(k).$$

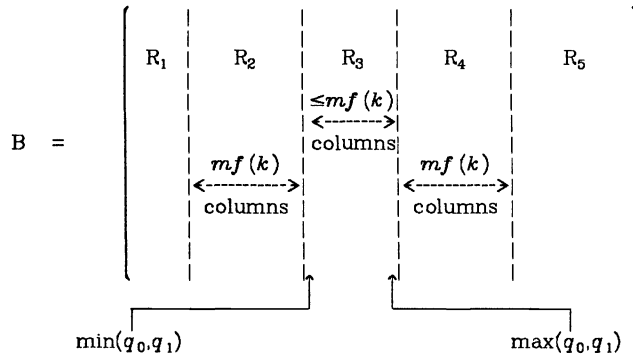


FIG. 4

Hence:

$$\begin{aligned} store(B) &\geq \sum_{i \in I_{MIN} \cup I_{MAX}} store(rowstrip(B, i, K)) \\ &\geq |I_{MIN} \cup I_{MAX}| \cdot (m + 2mf(k)) \geq (k - 3f(k)) \cdot (m + 2mf(k)). \end{aligned}$$

With (7):

$$\frac{store(B)}{optstore(A)} \geq \frac{(k - 3f(k)) \cdot (m + 2mf(k))}{(3m - 2)k} \geq \frac{(k - 3f(k)) \cdot (1 + 2f(k))}{3k} > c.$$

Now we have proved Claim 2 for the general case that $mf(k) < \min(q_0, q_1) < \max(q_0, q_1) < mk - mf(k)$.

Next assume $\min(q_0, q_1) \leq mf(k)$ or $\max(q_0, q_1) \geq mk - mf(k)$. The two special cases ($\min(q_0, q_1) \leq mf(k)$ and $\max(q_0, q_1) \geq mk - mf(k)$) are similar; thus we will only deal with $\min(q_0, q_1) \leq mf(k)$. Now we divide B in 4 parts:

$$\begin{aligned} R_2 &= B[\quad \quad \quad 1 \quad \quad \quad : \quad \min(q_0, q_1) - 1 \quad], \\ R_3 &= B[\quad \quad \quad \min(q_0, q_1) \quad \quad \quad : \quad \max(q_0, q_1) \quad], \\ R_4 &= B[\quad \quad \quad \max(q_0, q_1) + 1 \quad \quad \quad : \quad \max(q_0, q_1) + mf(k) \quad] \quad \text{and} \\ R_5 &= B[\quad \quad \quad \max(q_0, q_1) + mf(k) + 1 \quad \quad \quad : \quad \quad \quad mk \quad]. \end{aligned}$$

R_2, R_3 and R_4 together contain at most $3mf(k)$ columns, and therefore R_5 contains at least $mk - 3mf(k)$ columns and there are at least $k - 3f(k)$ columnstrips of A with a column in R_5 . Let

$$I_{MAX} = \{i: pmax(i) > \max(q_0, q_1) + mf(k)\}.$$

Then $|I_{MAX}| \geq k - 3f(k)$. For each $i \in I_{MAX}$: $pmax(i) - pmin(i) > mf(k)$. With (11):

$$store(B) \geq \sum_{i \in I_{MAX}} store(rowstrip(B, i, K)) \geq (k - 3f(k)) \cdot (m + 2mf(k))$$

and with (7): $store(B)/optstore(A) > c$.

Now Claim 2 is proven, and hence Case 2 satisfies the theorem.

Case 3. $q_1 \geq q_0 + mf(k)$.

Claim 3. With m and k satisfying (7) and $q_1 \geq q_0 + mf(k)$ there is a $\{0, 1\}$ -sequence u_B of length mk such that for each matrix C that is an insertion matrix for u_B into B , we have $store(C)/optstore(A \text{ concat } u_B) > c$.

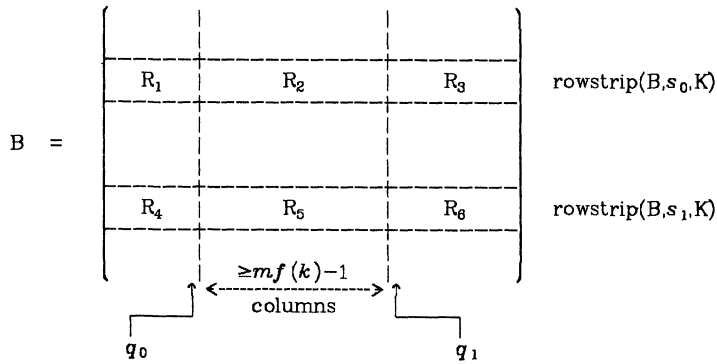


FIG. 5

Proof of Claim 3. Let $q_1 - q_0 \geq mf(k)$. With the definition of q_0 and q_1 there are s_0 and s_1 ($1 \leq s_0 \leq k, 1 \leq s_2 \leq k$) such that $q_0 = pmax(s_0)$ and $q_1 = pmin(s_1)$. Then we have: $s_0 \neq s_1$ and $pmin(s_1) - pmax(s_0) \geq mf(k)$. Let us divide B as in Fig. 5.

Because of (10) and the choice of s_0 and s_1 , the submatrices R_2, R_3, R_4 and R_5 are all zero matrices. Let u_B be a $\{0, 1\}$ -sequence of length mk defined by:

$$(12) \quad u_B[j] = \begin{cases} 1 & \text{if } (s_0 - 1)m < j \leq s_0 \cdot m, \\ 1 & \text{if } (s_1 - 1)m < j \leq s_1 \cdot m, \\ 0 & \text{otherwise.} \end{cases}$$

(13) A $\text{concat } u_B$ is a permuted block diagonal matrix with $k - 1$ blocks.

$$(14) \quad \begin{aligned} \text{optstore}(A \text{ concat } u_B) &\leq \sum_{\substack{i=1 \\ i \neq s_0, s_1}} \text{optstore}(\text{rowstrip}(A, i, K)) + 2 \left(m + 1 + \sum_{i=3}^{m+1} i \right) \\ &= (k - 2)(3m - 2) + m^2 + 5m - 2. \end{aligned}$$

Let C be an insertion matrix for u_B in B . Thus, there is a $y \in \mathbb{N}$ ($0 \leq y \leq mk$) such that

$$\begin{aligned} B[\quad , 1 : y] &= C[\quad , 1 : y], \\ u_B &= C[\quad , y + 1], \\ B[\quad , y + 1 : mk] &= C[\quad , y + 2 : mk + 1]. \end{aligned}$$

If $y \leq q_0$ then all zero elements of R_5 are stored in C though they are not stored in B . R_5 contains at least $m^2f(k) - m$ zero elements.

If $y \geq q_1 - 1$ then all zero elements of R_2 are stored in C though they are not stored in B . R_2 contains at least $m^2f(k) - m$ zero elements.

If $q_0 < y \leq q_1$, then the same number of zero elements will be stored in C , but now they are distributed over R_2 and R_5 . None of these zero elements are stored in B .

Thus:

$$\begin{aligned} \text{store}(C) &\geq \text{optstore}(B) + \text{ones}(u_B) + |\{\text{elements of } R_2 \text{ and } R_5 \text{ stored in } C\}| \\ &\geq k(3m - 2) + 2m + m^2f(k) - m = k(3m - 2) + m^2f(k) + m. \end{aligned}$$

With (14) and (7), this gives:

$$\frac{\text{store}(C)}{\text{optstore}(A \text{ concat } u_B)} > \frac{k(3m - 2) + m + m^2f(k)}{k(3m - 2) + m^2 - m + 2} > c.$$

This ends the proof of Claim 3 and the proof of Theorem 3.1. \square

As a direct consequence we have:

THEOREM 3.2. *Let X be an algorithm that takes $\{0, 1\}$ -matrices as input and that returns a column permutation of its input. Suppose there is a $c \in \mathbf{R}$ such that X is a c -approximation algorithm. Then X is not an on-line column insertion algorithm.*

However, Theorem 3.1 states more about c -approximation algorithms.

COROLLARY 3.3. *Let X satisfy the conditions of Theorem 3.2. Then for all $N \in \mathbf{N}$ and for every algorithm Y that takes $\{0, 1\}$ -matrices as input and that returns a column permutation of its input, there is an $m \times n$ matrix B ($m, n \geq N$) such that $X(B)$ is not an insertion matrix for $B[\ , n]$ in $Y(B[\ , 1:n-1])$.*

COROLLARY 3.4. *Let $N \in \mathbf{N}$, $c \in \mathbf{R}$ ($c \geq 1$). Then there are $m, n \geq N$ and an $m \times n$ matrix A such that for every two on-line column insertion algorithms X and X' there are $m \times (n+1)$ matrices B and C with*

$$\frac{\text{store}(X(B))}{\text{optstore}(B)} > c, \quad \frac{\text{store}(X'(C))}{\text{optstore}(C)} > c \quad \text{and} \quad B[\ , 1:n] = C[\ , 1:n] = A.$$

Theorem 3.2 is the special case of Corollary 3.3 with X replaced for Y . Observe that Corollary 3.3 holds even if Y finds the optimum column permutation of its input.

It actually is the insertion of the last column of its input that prevents an on-line column insertion algorithm to be a c -approximation algorithm for some $c \in \mathbf{R}$. If we design an approximation algorithm that first determines some column permutation D of all but the last column u of its input (such that D is independent of u) and then inserts u in D , then this algorithm is not a c -approximation algorithm for any $c \in \mathbf{R}$.

With Corollary 3.3 we have described a large class of approximation algorithms that are not c -approximation algorithms. This class contains, among others, the on-line column insertion algorithms.

4. Extensions to other classes of approximation algorithms. In the previous section we saw that many approximation algorithms can provide bad approximations to the optimal storage (in a rowmat data structure) of a column permutation of a matrix. In particular this bad result is obtained for some block diagonal matrices. Knowing this, one can try to design more sophisticated approximation algorithms. In this section we will extend Theorem 3.1 and, consequently, Theorem 3.2 and the Corollaries 3.3 and 3.4, and describe larger classes of approximation algorithms that still do not contain a c -approximation algorithm for any $c \in \mathbf{R}$.

4.1. Preprocessing to block diagonal form. The class of matrices used in the proof of Theorem 3.1 merely consists of block diagonal and permuted block diagonal matrices (see (8) and (13)). For each block diagonal matrix there is an optimum column permutation that is block diagonal too. Moreover, as we have seen before (Algorithm 2), there is an on-line column insertion algorithm that preserves this block diagonal structure if applied to a block diagonal matrix. Thus it seems reasonable to permute the rows and columns of a permuted block diagonal matrix to block diagonal form before applying any on-line column insertion algorithm. Unfortunately this does not give a c -approximation algorithm for any $c \in \mathbf{R}$. In the proof of Theorem 3.1 we could have chosen a slightly different A' : A' has also nonzero elements in the positions $(m+1, m)$, $(2m+1, 2m)$, \dots , $((k-1)m+1, (k-1)m)$ and $(1, km)$. With this change the blocks of A have been made loosely connected in A' . The column u_B in the proof would then make a stronger connection between two loosely connected blocks of A' that are far away from each other in the column permutation B . Preprocessing A' into block diagonal form does not change A' because it is already in this form (one block).

To prove negative results concerning more sophisticated preprocessors, we introduce the concept of a separator of a matrix.

DEFINITION 4.1. Let $p \in \mathbb{N}$ and A an $m \times n$ $\{0, 1\}$ -matrix with $p \leq n$.

(i) A set of p columns of A is a *separator* of A if the matrix A with the columns deleted, is a permuted block diagonal matrix with at least two blocks.

(ii) Let S be a smallest separator of A . A is in *connected order* if the deletion of the columns of S from A results in a block diagonal matrix.

A matrix A with a small separator is almost a permuted block diagonal matrix. If, in addition to this, A is in connected order, A can be considered almost block diagonal. The idea is that preprocessing an input matrix into connected order, would improve the worst-case behavior of on-line column insertion algorithms. However, the next theorem states that if an on-line column insertion algorithm is combined with a preprocessing algorithm that permutes matrices in connected order, this will not give rise to a c -approximation algorithm for any $c \in \mathbb{R}$.

THEOREM 4.1. Let $N \in \mathbb{N}$, $\epsilon, c \in \mathbb{R}$ ($c \geq 1, \epsilon > 0$). Then there exist an $n \in \mathbb{N}$ and a square $\{0, 1\}$ -matrix A such that for each column permutation B of A :

if $store(B)/optstore(A) \leq c$ then there is a $\{0, 1\}$ -sequence u_B of length $n^{1+\epsilon}$ such that $store(C)/optstore(A \text{ concat } u_B) > c$ for any matrix C that is an insertion matrix for u_B in B . A and $A \text{ concat } u_B$ satisfy the following conditions:

- (i) they are clean;
- (ii) they have $n^{1+\epsilon} \geq N$ rows;
- (iii) they are in connected order;
- (iv) they do not contain a separator of size $< n^{1-\epsilon}$.

Proof. Because the proof of this theorem is similar to the proof of Theorem 3.1 and uses the same kind of arguments, we will only give an outline. Let $f: \mathbb{N} \rightarrow \mathbb{N}$ be a function with $f(n) = \Omega(n)$ and there are $\delta > 0$ and $K \in \mathbb{N}$ such that for all $k \geq K$ $(3f(k) - 1)/3k - \frac{1}{3} > \delta$. Let M, U and A be as in Fig. 6. Then we have:

- (i) The smallest separator of $A_{n,k,d}$ contains $2d - 2$ columns,
- (ii) $nd \leq optstore(M_i) \leq (d - 1)n + (n - d + 1)d$,
- (iii) $kd(\frac{1}{2}(d - 1) + n) \leq optstore(A_{n,k,d})$ and $optstore(A_{n,k,d}) \leq k(3nd - \frac{1}{2}d^2 + \frac{1}{2}d - 2n) - n(d - 1) - \frac{1}{2}d(d - 1)$.

Let B be a column permutation of $A_{n,k,d}$. $pmax(i)$ and $pmin(i)$, $1 \leq i \leq k$, are defined as in (10). Without proof we state:

Claim. $store(rowstrip(B, i, K)) \geq d(pmax(i) - pmin(i) + 1)$ for all i ($1 \leq i \leq k$).

Let $q_0 = \max\{pmin(i): 2 \leq i \leq k\}$, $q_1 = \min\{pmax(i): 2 \leq i \leq k\}$. The rest of the proof is quite the same as the proof of Theorem 3.1, except that it requires more calculations. We only show where the connected order plays a role. $A_{n,k,d}$ does not have a separator of size less than $2d - 2$. Deletion of the last $d - 1$ columns of the first and last columnstrip of $A_{n,k,d}$, results in a block diagonal matrix with two blocks. Moreover, the sequence u_B that is defined will not have nonzero elements in the first n positions, because $pmin(1)$ and $pmax(1)$ do not play any role in the value of q_0 and q_1 . Therefore, deletion of the same $2d - 2$ columns of $A_{n,k,d} \text{ concat } u_B$ once again results in a block diagonal matrix with two blocks. \square

Remark. Theorem 4.1 even holds if we put the additional restriction on A (and $A \text{ concat } u_B$) that the columns of a smallest separator are consecutive in A (and $A \text{ concat } u_B$).

Thus, preprocessing a matrix into connected order does not help us to find c -approximation if we use an on-line column insertion algorithm. Of course, it is impossible to prove negative results for all combinations of preprocessing and on-line

$$M_{n,d} = \begin{pmatrix} 1 & & & & \overbrace{1 \dots 1}^{d-1} \\ \vdots & & & & \vdots \\ 1 & \dots & \phi & & 1 \\ & & & & \vdots \\ \phi & & & & \overbrace{1 \dots 1}^d \end{pmatrix}$$

an $n \times n$ $\{0,1\}$ -matrix in which each row and column has exactly d non-zero elements,

$$U_{n,d} = \begin{pmatrix} & & & & \overbrace{1 \dots 1}^{d-1} \\ & & & & \vdots \\ & & & & 1 \\ & \phi & & & \\ & & & & \vdots \end{pmatrix}$$

an $n \times n$ $\{0,1\}$ -matrix,

$$A_{n,k,d} = \begin{pmatrix} M_1 & \phi & & \phi & U_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ U_2 & M_2 & & \phi & \phi \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \phi & U_3 & \dots & & \phi \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \phi & \phi & \dots & & \phi \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \phi & \phi & & \phi & U_k & M_k \end{pmatrix}$$

with $M_i = M_{n,d}$ $(1 \leq i \leq k)$
 $U_i = U_{n,d}$ $(1 \leq i \leq k).$

FIG. 6

column insertion algorithms: one of these combinations gives for each matrix the optimum column permutation. But each preprocessing algorithm that does not change the column number of u_B if applied to A concat u_B (see (12)), does not improve the performance of any on-line column insertion algorithm.

4.2. Mixers and on-line column insertion algorithms. Another kind of more sophisticated approximation algorithms can be obtained by incorporating a "small mixer" in an on-line column insertion algorithm. On-line column insertion algorithms often can be formulated in such a way that the columns are processed one by one. The columns that are processed do not loose their relative order with the insertion of a next column. If we incorporate a "small mixer", we allow that the columns already processed are permuted slightly just before column i will be inserted.

DEFINITION 4.2. Let A be an $m \times n$ $\{0, 1\}$ -matrix, u an $\{0, 1\}$ -sequence of length m and $s: \mathbb{N} \rightarrow \mathbb{N}$ a function with $s(k) \leq k$ for all $k \in \mathbb{N}$. Let B a column permutation of A concat u and column i of A concat u is column p_i of B $(1 \leq i \leq n + 1)$.

B is an *s-mix1 insertion matrix* for u in A if there are $n - s(n)$ numbers $i_1 < i_2 < \dots < i_{n-s(n)} < n$ such that

$$(15) \quad i_j < i_k \quad \text{if and only if} \quad p_j < p_{i_k} \quad (1 \leq j \leq n - s(n), 1 \leq k \leq n - s(n)).$$

B is an *s-mix2 insertion matrix* for u in A if for all j $(1 \leq j \leq n)$ we have:

$$\text{if } p_j < p_{n+1} \text{ then } |p_j - j| \leq s(n) \text{ and if } p_j > p_{n+1} \text{ then } |p_j - 1 - j| \leq s(n).$$

DEFINITION 4.3. Let X be an algorithm that takes $\{0, 1\}$ -matrices as input and that returns a column permutation of its input. Let $s: \mathbb{N} \rightarrow \mathbb{N}$ be a function with $s(k) \leq k$

for all $k \in \mathbb{N}$. X is an *on-line column s -mix1* (resp. *s -mix2*) *insertion algorithm* if for each $m \times n$ $\{0, 1\}$ -matrix A , $X(A)$ is an s -mix1 (resp. s -mix2) insertion matrix for $A[\ , n]$ in $X(A[\ , 1:n-1])$.

In an on-line column s -mix1 insertion algorithm at most $s(n-1)$ columns may be deleted from $X(A[\ , 1:n-1])$ and inserted somewhere else in this matrix before inserting column n . In particular, an s -mix1 insertion algorithm allows that the first column of $X(A[\ , 1:n-1])$ will be the last column of $X(A)$, in case $s(n-1) \geq 1$. The way these $s(n-1)$ columns are mixed and once again inserted, will have to depend on column n , otherwise Corollary 3.3 states that we do not have a better approximation algorithm.

In an on-line column s -mix2 insertion algorithm all columns of $X(A[\ , 1:n-1])$ are allowed to be permuted, but the new column number will differ at most $s(n-1)$ from the old column number. With $s(n-1) < n-2$, this means that the first column of $X(A[\ , 1:n-1])$ will never be the last column of $X(A)$. For the same reason as above we are only interested in mix2 algorithms that depend on column n of A . Unfortunately, if $s(n)$ is not large enough, no on-line column s -mix1 or s -mix2 insertion algorithm is a c -approximation algorithm.

THEOREM 4.2. *Let $N \in \mathbb{N}$, $\epsilon, c \in \mathbb{R}$ ($c \geq 1, \epsilon > 0$). Then there are an $n \in \mathbb{N}$ and a clean square $\{0, 1\}$ -matrix A with $n^{1+\epsilon} \geq N$ rows such that for each column permutation B of A we have:*

If $store(B)/optstore(A) \leq c$ then there is a $\{0, 1\}$ -sequence u_B of length $n^{1+\epsilon}$ such that for every function $s: \mathbb{N} \rightarrow \mathbb{N}$ with $s(k) \leq k^{1-\epsilon}$ ($k \geq 2$), $store(C)/optstore(A \text{ concat } u_B) > c$ for every matrix C that is an s -mix1 insertion matrix for u_B in B .

THEOREM 4.3. *Theorem 4.2 with “mix1” replaced by “mix2”.*

We only sketch the proofs of these theorems, because they are similar to the proof of Theorem 3.1.

Proof of Theorem 4.2. Use the matrix A given in (8), with M_i an $n \times n$ matrix. Let B be a column permutation of A . Let $k = n^\epsilon$.

We only have to look at the case $store(B)/optstore(A) \leq c$ in which we have to provide u_B . Define u_B as in (12). Let B be divided as in Fig. 5.

Because $s(n^{1+\epsilon}) < n$, there is at least one column of each columnstrip of A of which the column number is one of the $i_1, \dots, i_{n^{1+\epsilon}-s(n^{1+\epsilon})}$ of (15). Let us delete all columns p_i of B of which i does not occur in $i_1, \dots, i_{n^{1+\epsilon}-s(n^{1+\epsilon})}$, resulting in a matrix B' . Then there surely is a column of $colstrip(A, s_0, K)$ in the left part of B' and one of $colstrip(A, s_1, K)$ in the right part of B' . Inserting u_B in B' causes at least $n(nf(k) - s(n^{1+\epsilon}) - 1)$ zero elements to be stored. Inserting the deleted columns can make things only worse. With suitable f and n large enough, ratio (4) is violated. \square

Proof of Theorem 4.3. Let A, B and u_B as above in the proof of Theorem 4.2. After the mixing of B has been done, column $pmax(s_0)$ of B is at most $s(n^{1+\epsilon})$ positions to the right and column $pmin(s_1)$ of B $s(n^{1+\epsilon})$ to the left. Thus inserting u_B causes at least $n(nf(k) - 2s(n^{1+\epsilon}) - 1)$ zero elements to be stored. With suitable f and n large enough, ratio (4) is violated. \square

COROLLARY 4.4. *Let X be a c -approximation algorithm. Then X is neither an on-line column s -mix1 nor an on-line column s -mix2 insertion algorithm for any $\epsilon \in \mathbb{R}$ ($\epsilon > 0$) and $s: \mathbb{N} \rightarrow \mathbb{N}$ with $s(k) \leq k^{1-\epsilon}$.*

COROLLARY 4.5. *Let $c \in \mathbb{R}$ ($c \geq 1$). Let X be a c -approximation algorithm and let $N \in \mathbb{N}$ such that $store(X(A))/optstore(A) \leq c$ for all matrices A with at least N rows and N columns. Then for every algorithm Y that takes $\{0, 1\}$ -matrices as input and*

that returns a column permutation of its input, there is an $m \times n$ matrix B ($m, n \geq N$) such that $X(B)$ is neither an s -mix1 nor an s -mix2 insertion matrix for $B[1:n, 1:n]$ in $Y(B[1:n, 1:n-1])$ for any $\varepsilon \in \mathbf{R}$ ($\varepsilon > 0$) and $s: \mathbf{N} \rightarrow \mathbf{N}$ with $s(k) \leq k^{1-\varepsilon}$.

Acknowledgments. The author is grateful to J. van Leeuwen for the many discussions and would like to thank J. van Leeuwen, A. A. Schoone and D. P. Dobkin for their careful reading of the manuscript.

REFERENCES

- [1] K. S. BOOTH, *PQ-tree algorithms*, Ph.D. thesis, Univ. California, Berkeley, 1975.
- [2] K. S. BOOTH AND G. S. LUEKER, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms*, J. Comput. System. Sci., 13 (1976), pp. 335-379.
- [3] G. VON FUCHS, J. R. ROY AND E. SCHREM, *Hypermatrix solution of large sets of symmetric positive-definite linear equations*, Comp. Meth. Appl. Mech. Eng., 1 (1972), pp. 197-216.
- [4] D. R. FULKERSON AND O. A. GROSS, *Incidence matrices and interval graphs*, Pacif. J. Math., 15 (1965), pp. 835-855.
- [5] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability, A Guide To The Theory of NP-completeness*, W. H. Freeman, San Francisco, 1979.
- [6] J. A. GEORGE AND J. W. H. LIU, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [7] S. P. GHOSH, *Data Base Organization for Data Management*, Academic Press, New York, 1977.
- [8] M. C. GOLUBIC, *Algorithmic Graph Theory And Perfect Graphs*, Academic Press, New York, 1980.
- [9] A. JENNINGS, *A compact storage scheme for the solution of simultaneously equations*, Comput. J., 9 (1966), pp. 281-285.
- [10] L. T. KOU, *Polynomial complete consecutive information retrieval problems*, this Journal, 6 (1977), pp. 67-75.
- [11] J. K. REID, *Solutions of linear systems of equations: direct methods (general)*, in Sparse Matrix Techniques, V. A. Barker, ed., Lecture Notes in Mathematics 572, Springer-Verlag, Berlin, 1977, pp. 102-129.
- [12] A. TUCKER, *A structure theorem for the consecutive ones property*, J. Combin. Th., 12(B) (1972), pp. 153-162.
- [13] M. VELDHORST, *An analysis of sparse matrix storage schemes*, Mathematical Centre Tract 150, Mathematical Centre, Amsterdam, the Netherlands, 1982.
- [14] M. YANNAKAKIS, unpublished results.

OPTIMAL ALLOCATION OF AREA FOR SINGLE-CHIP COMPUTATIONS*

ZVI M. KEDEM†

Abstract. This paper presents initial results on the problem of allocation of the available VLSI chip's area among various functional components such as I/O pads, memory cells, and internal wiring. First, a general lower bound for any chip computing a transitive function is derived; this bound is tight for certain functions. The arguments used in the various derivations are later used to specify which of the components are critical depending on the relative sizes of the chip and the number of variables of the function to be computed. The general lower bound is powerful enough that many of the previously proved lower bounds (which could account only for some of the functional requirements) are obtained as explicit special cases of the new result.

Key words. VLSI complexity, area/time tradeoffs, optimal allocation of resources

1. Introduction. The purpose of this paper is to examine the following question: Given chip area A , how to optimally allocate it among I/O, memory and wiring (internal communication) so that some function of interest can be computed in the provably minimum time? Perhaps surprisingly, the geometric nature of the model makes it sufficiently "structured" so that meaningful study of the consequences of such allocation for the purpose of designing provably optimal (up to constant factors) chips is feasible. (It is interesting to contrast this situation with the one in unstructured models, where similar attempts at formal treatment of optimal allocation do not seem to be as fruitful; see also [Ar80, p. 210].) Our results permit us to completely characterize both the amount of resources required and their optimal allocation for certain computations, and to obtain some of the previously proved lower bounds (which generally accounted for some of the resources, or restricted their utilization in some way) as special cases, by invoking explicit constraints.

Our physical and geometrical assumptions on the chip structure are the same as those commonly employed in theoretical research (see e.g., [Th79] and [BrKu81] to which the reader is referred) and therefore we will only sketch them briefly, more for the purpose of fixing the terminology. We consider chips having a constant number of layers ν ; for advantages of using three-dimensional chips see the work of A. Rosenberg [Ro81]. We will assume that the chip is designed on a square grid; the "squares" of the grid will be referred to as *tiles*. We also find it convenient to assume that the chip is convex (or we might deal with its convex hull), as we can then use our simple separator theorem. Hence, without loss of generality [Le80] we assume that the chip is a square whose sides lie in the layout grid. (Somewhat less stringent assumptions could suffice, see, e.g., [Th79], [LeMe81] and [Sa81].) A major decision to be made concerns the appropriate wire delay function, namely what is the time required to move a bit along a wire as a function of the distance. We will consider two delay functions studied previously: unit (constant) and linear. They seem to be sufficiently representative to give insight into other possible monotonic delay functions. Ignoring nonessential constants, we state some "reasonable" assumptions concerning the technological capabilities of the chip. (As we shall see in § 4, it is worthwhile to treat input and output separately.)

* Received by the editors February 3, 1983. This research was partially supported by the National Science Foundation under grants MCS 80-25376, MCS 81-04882 and MCS 81-10097. Part of the research was conducted when the author was visiting the Department of Computer Science, Columbia University.

† Department of Computer Science, State University of New York, Stony Brook, New York. Current address: Courant Institute of Mathematical Sciences, New York University, New York, New York 10012.

- *Input.* In one time unit an I/O pad of unit area can read in one input bit.
- *Output.* In time unit an I/O pad of unit area can write out one output bit.
- *Memory.* One unit of area can store a number of bits equal to the numbers of layers.
- *Wires.* Each wire is of width one, and can “move” only one bit at any time unit. The wire delay function may be chosen to be either constant or linear.
- *Computation.* In one time unit a computational element of one unit of area can compute an arbitrary boolean function of the bits stored in it.

We will now describe the chip’s functional behavior. A chip computing some function has defined an I/O *schedule*, that is a set of triples of the form (i, j, k) , meaning that I/O pad number i at time instance j reads the input variable (or respectively writes the output variable) number k ; as the input and the output variables’ indices will be distinct there will be no confusion using this definition. Specifically, we will say that a chip with I/O pads P_1, P_2, \dots, P_p computes some boolean function $(z_1, z_2, \dots, z_n) = f(z_{n+1}, z_{n+2}, \dots, z_m)$ in time T , if there exists a subset μ (the I/O schedule) of $\{1, \dots, P\} \times \{0, \dots, T\} \times \{1, \dots, m\}$ with the following properties:

1. If $(i_1, j_1, k_1) \neq (i_2, j_2, k_2)$ are elements of μ , then $(i_1, j_1) \neq (i_2, j_2)$. (A single I/O pad can deal with only one variable in a single time instance.)
2. $\forall k \leq n \exists i \geq 1 \exists j [(i, j, k) \in \mu]$. (Every output variable is produced.)
3. If whenever $(i, j, k) \in \mu$ for $k \geq n + 1$ the input z_k is supplied at I/O pad P_1 at time j , then whenever $(i, j, k) \in \mu$ for $k \leq n$ the output z_k is produced by the chip at I/O pad P_i at time j , and the values of the variables are related by $(z_1, \dots, z_n) = f(z_{n+1}, \dots, z_m)$. (If the chip receives the inputs according to μ , it will produce the outputs according to μ also, and the values of the variables satisfy the function f for all possible values of the input variables.)

We explicitly point out to the reader that there is no assumption that an input variable can be read at most once during the computation. This point is discussed towards the end of this section.

What we have actually defined above was a *where-* and *when-determinate* schedule (which is therefore identical for all possible input variables). Informally, a schedule is *where-indeterminate* if the first elements of the triples in μ are not “fixed,” but can be determined (perhaps by an oracle) on the basis of the values of the input variables; it is *when-indeterminate* if the second elements of the triples can be determined on the basis of the values of the input variables. As we do not assume that the chip has any control over its input, the where- and when-indeterminate schedules do not model actual chip behavior well, but may be of theoretical interest.

We will specifically consider chips computing *transitive* groups (or functions) as done first by J. Vuillemin [Vu83], who also attempted to relate their complexity to that of other functions of interest. (For a discussion of the relations between the complexity of transitive functions and more “useful” functions such as binary multiplication see [KeZo81].) A group G of permutations of $\{1, 2, \dots, n\}$ is called transitive if $\forall i, j \in \{1, 2, \dots, n\} \exists g \in G [g(i) = j]$. A boolean function f as above is called a *transitive* function of n variables (or computes a transitive group) if it is of the form $(y_1, \dots, y_n) = f(x_1, \dots, x_n, s_1, \dots, s_p)$, and for some transitive group G ,

$$\forall g \in G \exists s_1, \dots, s_p \in \{0, 1\} \forall x_1, \dots, x_n \in \{0, 1\} \\ [f(x_1, \dots, x_n, s_1, \dots, s_p) = (x_{g(1)}, \dots, x_{g(n)})].$$

The inputs were partitioned into two classes, variables to be permuted x_1, \dots, x_n (onto y_1, \dots, y_n) and arbitrary parameters s_1, \dots, s_p dependent only on the permutation to

be computed. Note that the definition above allows one, in effect, to input a different program (the values of s_1, \dots, s_p) for each permutation. This is of no concern in proving lower bounds, but we will return to this point briefly when upper bounds are considered.

VLSI complexity theory has been attracting growing interest following the pioneering work of C. Thompson [Th79]. Nevertheless, we believe that previous work does not address a very important concern of how to utilize the available area of the chip. Specifically, it was assumed in previous work, explicitly or implicitly, that one or two of the requirements dominate and the other can be ignored. It was also believed that this would suffice for predicting adequately the total area needs of the computation.

As the most important example of that previous approach we will focus now on the $AT^2 = \Omega(n^2)$ type results considered first by [Th79] and later further developed by among others [BrKu81] and [Vu83]. A careful reader of the various proofs of these lower bounds will observe that they account only for the wiring (internal communication) requirements of the computation, and thus they can shed little insight on the other resources required during computations. It may indeed seem at first glance that wiring is the critical resource, as for some functions $AT^2 = O(n^2)$. Furthermore, sometimes a result of the form $A = \Omega(n)$ is proved, essentially implying that storing the complete problem's description can/must always be done [BrKu81]. However, these are both really only consequences of the model chosen, which frequently assumes that each input is read only once, an assumption which is not made in this paper. Thus, it is not possible to study within that framework the consequences of limited memory which sometimes is the bottleneck of the computation. In other words this lower bound cannot predict the complexity unless the chip is large.

As a motivating analogy for our more general approach, consider the problem of sorting sequences of items on a computer with limited memory. If the sequence to be sorted is small compared to the memory size, then one normally tries to minimize the amount of internal computation. If the sequence is large (analogous to the violation of $A = \Omega(n)$), a different set of concerns arises. The bottleneck is then the limited memory size, which causes for instance, considerable movement of information between internal and external storage (see, e.g. [Kn73]), and theory should account for this. It is generally the case that during sorting intermediate results are read in and out of CPU/RAM, a more sophisticated strategy than just reading the input several times, as we allow here. However, even the assumptions we make allow us to prove interesting results, which seem to indicate the relative importance of input, memory and wiring *depending on the relative sizes of n and A .*

It should be mentioned that a lower bound of the form $A^2T = \Omega(n^2)$ was considered by [Va81] and [Sa82], further developing [Gr76]. This lower bound accounts for input and memory requirements of VLSI chips, even though it was originally developed for boolean circuits. The model used in the derivation of that result also assumes that inputs may be read more than once. However, this lower bound too characterizes only some of the resources needed, as it cannot predict the complexity unless the chip is small, and the wiring needs are not critical.

As stated in the first paragraph of this section, we wish to study the optimal allocation of the area of a single chip among the various resources required. To this end we first prove a general lower bound of the form $F(A, T) = \Omega(G(n))$, where F and G are suitable functions, and n is the number of variables being permuted. This is indeed done in § 2, although we find it more convenient to first state T as a function of A and n . In § 3 we very briefly sketch a construction showing that our lower bound is optimal for some transitive functions. In § 4 we compare and contrast our work with

some of the previously published results. We conclude that section by actually stating which resources are critical, depending on the relative sizes of A and n .

2. The lower bound.

LEMMA 1. Consider a set of tiles of the layout to which some positive weights are assigned. Assume that each weight is at most λ , and the sum of weights is at least ξ . Then for every $\eta \leq \xi$ there exist two slices, vertical S_1 and horizontal S_2 defined by the sequences of "points": $(x_1, -\infty), (x_1, y_1), (x_2 = x_1 + 1, y_1), (x_1 + \infty)$ for S_1 , and $(-\infty, y_2), (x_3, y_2), (x_3, y_3 = y_2 + 1), (+\infty, y_3)$ for S_2 , for integer $x_1, x_2, x_3, y_1, y_2, y_3$, and such that both the weight of the tiles to the left of S_1 and the weight of the tiles below S_2 lie in the interval $(\eta - \lambda, \eta]$.

Proof. We sketch the proof for S_1 . Let $x_2 = \min \{\text{integer } x \mid \text{the weight of the tiles to the left of the line } x = x_2 \text{ is at least } \eta\}$, and let $x_1 = x_2 - 1$. As the weight of each tile is at most λ , the existence of an appropriate y_1 follows immediately. \square

We now prove a convenient version of the planar separator theorem. (A similar result was also proved by J. Savage [Sa82].)

LEMMA 2. Let $q > \frac{1}{2}$ and let $m \geq 128q$ be integer. Let nonnegative integer weights be assigned to the tiles of the chip such that the sum of the weights of the tiles is m , and the weight of each tile is at most $\lfloor m/128q \rfloor$. (This, of course, implies that $A \geq 128q$.) Then the chip can be separated by a separator lying in the layout grid and of length $5\sqrt{Aq}$ at most into $\beta \geq 2q$ not necessarily connected "patches" such that the weight of the tiles in each patch is at least $m/10q$.

Proof. Before starting the formal proof we present an intuitive explanation. We wish to divide the chip into pieces (patches) of approximately equal weight (actually of certain minimal weight which is close to the average) by means of cuts of small total length. First, consider the very simple example (see Fig. 1), where each tile of

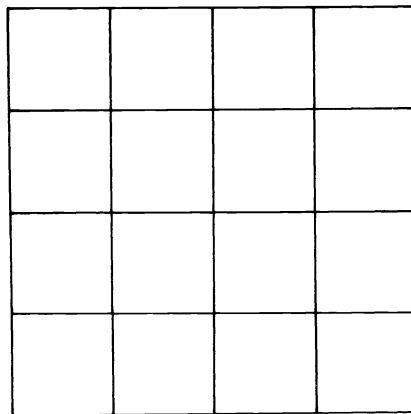


FIG. 1

the chip is of weight m/A . Then, assuming that \sqrt{A} is divisible by $\sqrt{2q}$, we can slice the chip by means of $\sqrt{2q} - 1$ vertical and $\sqrt{2q} - 1$ horizontal segments of length \sqrt{A} each so that each of the $2q$ small squares (patches) obtained is of weight $m/2q$ and the total length of the segments is $2(\sqrt{2q} - 1)\sqrt{A}$. \square

The lemma assumes a more general situation, as there is no requirement that all the tiles are of equal weight. Therefore the patches are constructed in two stages (see Fig. 2). First, approximately $\sqrt{2q}$ horizontal "strips" (parcels), which are not necessarily

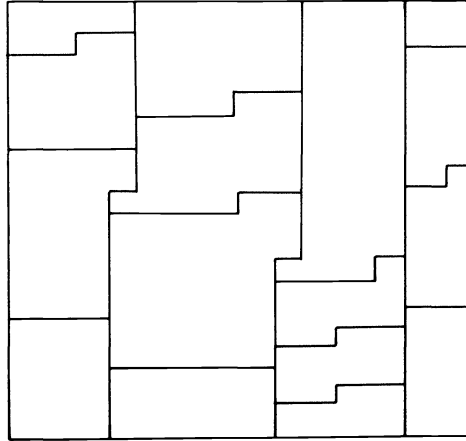


FIG. 2

connected, are obtained by means of successive slicing off strips of weight $m/\sqrt{2q}$ approximately. (Technically, at least $m/3\sqrt{q}$.)

Each such cut is of length \sqrt{A} or $\sqrt{A} + 1$ as it may contain one “knee” (see Lemma 1). Then, each such strip is sliced in turn into approximately $\sqrt{2q}$ patches each of weight $m/(\sqrt{2q}\sqrt{2q})$ approximately. (Technically, at least $m/10q$.) Although a single strip is not of constant width and furthermore about $\sqrt{2q} - 1$ additional knees are possible in each strip, the total length of the cuts is still $O(\sqrt{Aq})$.

We now proceed with the formal proof. By repeated application of Lemma 1 it can be shown that there exist $\lceil \sqrt{2q} - 1 \rceil$ vertical slices dividing the chip into $\lceil \sqrt{2q} \rceil$ regions, which we will refer to as *parcels*, each containing tiles of total weight at least

$$\left\lceil \frac{m}{\sqrt{2q} + 1} \right\rceil - \frac{m}{128q} \geq \frac{m}{2\sqrt{2q}} - \frac{m}{64\sqrt{q}} \geq \frac{m}{3\sqrt{q}}.$$

The total length of the vertical cuts (each cut is obtained by restricting a slice to the chip) is bounded from above by $(\sqrt{A} + 1)\sqrt{2q}$.

By repeated application of Lemma 1 to each parcel in turn, it can be shown that there exist $\lceil \sqrt{2q} - 1 \rceil$ horizontal slices for each parcel dividing the parcel into $\lceil \sqrt{2q} \rceil$ regions, referred to as *patches* in the statement of Lemma 2, each containing tiles of total weight at least

$$\left\lceil \frac{m/3\sqrt{q}}{\sqrt{2q} + 1} \right\rceil - \frac{m}{128q} \geq \frac{m}{3\sqrt{q}2\sqrt{2q}} \geq \frac{m}{10q}.$$

The total length of the horizontal cuts (each cut is obtained by restricting the slice to its parcel) is bounded from above by $(\sqrt{A} + 3(\sqrt{2q} + 1))\sqrt{2q}$. As $A \geq 128q > 64$, it follows that $1 \leq \sqrt{2q} \leq \sqrt{A}/8$ and therefore the total length of the cuts is bounded from above by $5\sqrt{Aq}$. \square

During the proof of our results a certain argument needs to be used a number of times. In order to avoid repetition, we will prove a basic lemma which states the argument in sufficient generality that it can be used as needed later. Let the chip compute some transitive G , and let g be some generic element of G . We may consider the computation performing this particular g . Pick two time instances t_1 and t_2 during the computation, such that $0 \leq t_1 < t_2 \leq T$. Assume that the chip was separated by means

of some separator of length σ into at least $\beta \geq 1$ disjoint patches (which are not necessarily connected). Let β' be the actual number of patches. P^π , $\pi = 1, \dots, \beta'$, will stand for the π th patch. Of course, if $\beta' = 1$ then $\sigma = 0$. We do not assume here that weights are assigned to the tiles of the chip, so we will not use Lemmas 1 and 2. During the time interval $(t_1, t_2]$ each patch outputs some variables, at least γ in number. Let the actual number of variables output by P^π be γ^π . To do that each patch needed to acquire by time instance $t_2 - 1$ enough information to compute the appropriate outputs. This information can come out of three sources: the information stored in the patch at time instance t_1 , the information read into the patch during the interval $[t_1, t_2)$, and the information that entered the patch from other patches during the interval $[t_1, t_2)$ by crossing the separator. We will consider all these sources of information.

Let α^π be the upper bound on the information that can be stored in P^π at t_1 in addition to the information present there at the beginning of the computation. (We wish to consider only the information that is dependent on the values read by the chip during the computation.) Let $\alpha_g = \sum_{\pi=1}^{\beta'} \alpha^\pi$ and $\alpha = \max_{g \in G} (\alpha_g)$. Note that a necessary condition for $\alpha > 0$ is $t_1 > 0$. Furthermore, $\alpha \leq \nu A$, where ν denotes the number of layers.

Let ρ^π be the upper bound on the information that has to arrive in P^π from the other patches by crossing the separator during the time interval $[t_1, t_2)$. Let $\rho_g = \sum_{\pi=1}^{\beta'} \rho_g^\pi$ and $\rho = \max_{g \in G} (\rho_g)$.

Let δ_i be the number of patches into which at least one copy of x_i was read during the interval $[t_1, t_2)$, and let $\delta = (\sum_{i=0}^{n-1} \delta_i) / n$.

LEMMA 3. *Using the above notation, $\rho \geq \gamma(\beta - \delta) - \alpha$. Furthermore, if $\beta > 1$ (and thus $\sigma > 0$), then $t_2 - t_1 \geq \rho / \sigma \nu \geq (\gamma(\beta - \delta) - \alpha) / \sigma \nu$.*

Proof. Let g vary over the elements of the transitive group G . Let $G_{ij} = \{g \in G | g(i) = j\}$. For every i and j choose g_{ij} to be some element of G_{ij} . As noted by J. Vuillemin [Vu83], for transitive G all G_{ij} 's have the same cardinality, which we will denote by h . Furthermore, $|G| = hn$.

Define:

$$D_i^\pi = \begin{cases} 1 & \text{if during } [t_1, t_2) \text{ no copy of } x_i \text{ was read into } P^\pi, \\ 0 & \text{otherwise.} \end{cases}$$

Note that $\sum_\pi D_i^\pi \geq \beta' - \delta_i$. Let $P_{\text{out}}^\pi = \{j | y_j \text{ was written by } P^\pi \text{ during } (t_1, t_2]\}$. Note that all P_{out}^π 's are disjoint and $\bigcup_{\pi=1}^{\beta'} P_{\text{out}}^\pi$ may be a proper subset of $\{1, \dots, n\}$. Define

$$E_i^g = \begin{cases} 1 & \text{if during } [t_1, t_2) \text{ no copy of } x_i \text{ was read into } P^\pi \text{ for which } g(i) \in P_{\text{out}}^\pi, \\ 0 & \text{otherwise.} \end{cases}$$

$$\sum_{g \in G} E_i^g = \sum_{j=0}^{n-1} \sum_{g \in G_{ij}} E_i^g = \sum_{j=0}^{n-1} h E_{ij}^g = h \sum_{j=0}^{n-1} E_{ij}^g \quad \text{as } \forall g \in G_{ij} [E_i^g = E_{ij}^g].$$

Note now that for each π , $\pi = 1, \dots, \beta'$, $\sum_{j \in P_{\text{out}}^\pi} E_{ij}^g = \gamma^\pi D_i^\pi \geq \gamma D_i^\pi$ and therefore $\sum_{j=0}^{n-1} E_{ij}^g \geq \sum_{j=0}^{\beta'} \gamma D_i^\pi$. Thus, $\sum_{g \in G} E_i^g \geq h \sum_{\pi=1}^{\beta'} \gamma D_i^\pi \geq h \gamma \sum_{\pi=1}^{\beta'} D_i^\pi \geq h \gamma (\beta' - \delta_i)$. Summing over i we get, $\sum_{i=0}^{n-1} \sum_{g \in G} E_i^g \geq nh \gamma (\beta' - \delta)$ and as $\sum_{g \in G} \sum_{i=0}^{n-1} E_i^g = \sum_{i=0}^{n-1} \sum_{g \in G} E_i^g$, we have $\sum_{g \in G} \sum_{i=0}^{n-1} E_i^g \geq nh \gamma (\beta' - \delta) = |G| \gamma (\beta' - \delta)$. We conclude that for some g , $\sum_{i=0}^{n-1} E_i^g \geq \gamma (\beta' - \delta)$.

Informally, $\sum_{i=0}^{n-1} E_i^g$ is a lower bound on the number of bits that are needed in various locations of the chip for output, but which have not been read into the "right" patches. Thus they were either stored in the right patches, or arrived there from other patches. More formally, we have $\alpha_g + \rho_g \geq \gamma (\beta' - \delta)$, and thus, $\alpha + \rho \geq \gamma (\beta' - \delta)$, proving the first claim of the lemma.

We now prove the second claim of the lemma. During $t_2 - t_1$ time units at least $\rho \geq \gamma(\beta' - \delta) - \alpha \geq \gamma(\beta - \delta) - \alpha$ bits have to move through the separator of length σ . As there are ν layers on the chip, at most $\sigma\nu$ bits can move across the separator during one time unit. This immediately proves the second claim. \square

COROLLARY 4. Consider t_1, t_2 such that $0 \leq t_1 \leq t_2 \leq T$. Let q_i be the number of times x_i was read into the chip during some interval $[t_1, t_2]$, and let $q = (\sum_{i=0}^{n-1} q_i)/n$. Let m be the number of outputs produced during the interval $(t_1, t_2]$. Then, $q \geq 1 - (\min\{t_1 A, \nu A\}) \div m$.

Proof. The first claim of Lemma 3 states that $\rho \geq \gamma(\beta - \delta) - \alpha$. Clearly, the most that the chip can “remember” at time t_1 is $\min\{\text{the number of inputs the chip has read so far, the storage available}\}$. Thus we have $\alpha = \min\{t_1 A, \nu A\}$. Applying Lemma 3 to our case and noting that $\beta = 1$ we set $\rho = 0$, $\gamma = m$, and $\delta \leq q$. Thus, $0 \geq m(1 - q) - \alpha$, and $q \geq 1 - \alpha/m = 1 - (\min\{t_1 A, \nu A\})/m$. \square

Not surprisingly, Lemma 3 will be used later (in conjunction with Lemma 2) to show that q is a fraction of 1 if α is a fraction of m , and thus during certain time intervals “many” inputs had to be read.

THEOREM 5. For any chip computing a transitive function of n variables,

$$T = \Omega\left(\frac{n}{A} \min_{q \geq 1} \left(\left(1 + \frac{n}{A}\right)q + \sqrt{\frac{A}{q}} \right) + \tau(n)\right),$$

where

$$\tau(n) = \begin{cases} \log n & \text{if the wire delay is constant,} \\ n^{1/3} & \text{if the wire delay is linear.} \end{cases}$$

Proof. The statement to be proved is equivalent (by trivial manipulations) to the conjunction of the following three statements:

1. $T = \Omega\left(\frac{n}{A} \min_{q \geq 1} \left(\frac{nq}{A} + \sqrt{\frac{A}{q}}\right)\right) = \Omega\left(\frac{n}{A} \min_{q \geq 1} \max\left\{\frac{nq}{A}, \sqrt{\frac{A}{q}}\right\}\right)$ for $A < \frac{n}{20\nu}$.
2. $T = \Omega\left(\frac{n}{A} \min_{q \geq 1} \left(q + \sqrt{\frac{A}{q}}\right)\right) = \Omega\left(\frac{n}{A} \min_{q \geq 1} \max\left\{\frac{nq}{A}, \frac{n}{\sqrt{qA}}\right\}\right)$.
3. $T = \Omega(\tau(n))$.

The reader will find it more convenient to follow the proof if each of these cases is considered separately.

LEMMA 6.

$$T = \Omega\left(\frac{n}{A} \min_{q \geq 1} \max\left\{\frac{nq}{A}, \sqrt{\frac{A}{q}}\right\}\right) \text{ for } A < \frac{n}{20\nu}.$$

Proof. By the structure of our argument we will actually prove that

$$T = \Omega\left(\frac{n}{A} \min_{q > 1/2} \max\left\{\frac{nq}{A}, \sqrt{\frac{A}{q}}\right\}\right),$$

which is, of course, equivalent to the claim of the lemma.

Choose a shortest time interval during which some $m \in [20\nu A, (20\nu + 1)A]$ outputs are written. (Such an interval exists, as at most A bits can be written during a single time unit.) Assume this is the interval $(t_1, t_2]$ for some $0 \leq t_1 < t_2 \leq T$. Let $t = t_2 - t_1$. By our choice of t_1 and t_2 it follows that $T \geq nt/(20\nu + 1)A$. As $n/(20\nu + 1)A = \Theta(n/A)$, it will suffice to show that $t = \Omega(\min_{q > 1/2} \max\{nq/A, \sqrt{A/q}\})$.

Let q_i be the number of times x_i was read into the chip during the interval $[t_1, t_2)$, and let $q = (\sum_{i=0}^{n-1} q_i)/n$. By Corollary 4, $q \geq 1 - (\min\{t_1, A, \nu A\})/m \geq 1 - \nu A/m \geq 1 - \nu A/20\nu A > \frac{1}{2}$. We will now show that, whatever the value of q (subject of course to $q > \frac{1}{2}$), $t = \Omega(nq/A)$ and $t = \Omega(\sqrt{A/q})$, thus proving the lemma.

First observe that, as there were at least nq input bits read during t time units, we immediately see that $t \geq nq/A$.

Before showing that $t = \Omega(\sqrt{A/q})$ we give an overview of the basic idea. The most interesting case occurs when no pad writes "many" bits during the interval. We will partition the chip into a certain number of patches each writing out approximately the same number of outputs during the interval, which will be possible as no pad has many outputs. We then use the fact (proven in Lemma 3) that for some "unlucky" permutation "many" bits that need to be output were not read into the "right" patches, but as the chip was "small," it could not "remember" a "large" number of them. Thus we conclude that "many" bits needed to travel during the interval across the separator that defines the patches. The separator was chosen using Lemma 2 in a way that makes it quite "short" (with respect to the already small chip). From here it follows that the time interval t had to be long enough in order to accommodate the required movement of information. We now proceed with the formal proof for this case.

Assume first that no output pad writes more than $m/128q$ bits during the interval $(t_1, t_2]$. Assign to each cell of the chip *weight* equal to the number of bits it writes during the interval. Apply Lemma 2 to the chip, obtaining a partition of the chip into patches each containing output pads writing $m/10q$ bits at least by means of a separator of length $5\sqrt{Aq}$ at most. We now apply Lemma 3. In this case we have, $\beta \geq 2q > 1$ (by Lemma 2 and $q > \frac{1}{2}$), $\sigma \leq 5\sqrt{Aq}$, $\gamma \geq m/10q$, $\alpha \leq \nu A$, $\delta_i \leq q_i$, and thus $\delta \leq q$. The conclusion of Lemma 3 gives us that

$$t \geq \frac{\gamma(\beta - \delta) - \alpha}{\sigma\nu} \geq \frac{(m/10q)(2q - q) - \nu A}{5\nu\sqrt{Aq}} \geq \frac{20\nu Aq/10q - \nu A}{5\nu\sqrt{Aq}} = \frac{A}{5\sqrt{Aq}} = \frac{1}{5} \sqrt{\frac{A}{q}}$$

We will now deal with the case when some output pad writes out more than $m/128q$ bits during the time interval. We will show that $t \geq (20\nu/128)\sqrt{A/q}$. As t has to be at least as long as it takes the output pad to write out its outputs we have $t \geq m/128q \geq 20\nu A/128q \geq (20\nu/128)A/q$. If $A/q \geq 1$, then as $A/q \geq \sqrt{A/q}$ the claim follows. Otherwise, note that as there are at most A pads writing m outputs we have, $t \geq m/A \geq 20\nu A/A = 20\nu > (20\nu/128)1 > (20\nu/128)\sqrt{A/q}$.

We have thus shown that $t = \Omega(\min_{q>1/2} \max\{nq/A, \sqrt{A/q}\})$, from which the lemma follows. \square

LEMMA 7.

$$T = \Omega\left(\frac{n}{A} \min_{q \geq 1} \max\left\{\frac{nq}{A}, \frac{n}{\sqrt{qA}}\right\}\right).$$

Proof. Similar to the proof of Lemma 6, but accounting for memory requirements is not necessary. \square

Proof of Theorem 5. It remains to show that $T = \Omega(\tau(n))$. We deal here in detail only with the case when the wire delay is linear, using a modification of an argument due to B. Chazelle and L. Monier [ChMo81]. Consider the output pad writing y_0 . During the T time units of the computation only the inputs read within the distance of at most T from the pad can "influence" the value of y_0 . Thus some region of the chip of area $O(T^2)$ must read at least one copy of each out of the n inputs x_0, \dots, x_{n-1} . As each unit of area could read at most T inputs during the computation, it follows

that $T^2 = \Omega(n/T)$, or $T = \Omega(n^{1/3})$. If the wire delay is constant, the application of a similar argument for fan-in, immediately gives $T = \Omega(\log n)$. \square

COROLLARY 8. *For a chip computing a transitive group of n variables:*

If the wire delay is constant then:

- *If $A \leq n^{2/3}$, then $A^2 T = \Omega(n^2)$.*
- *If $n^{2/3} \leq A \leq n$, then $A^3 T^3 = \Omega(n^4)$.*
- *If $n \leq A \leq (n/\log n)^{3/2}$, then $A^2 T^3 = \Omega(n^3)$.*
- *If $(n/\log n)^{3/2} \leq A$, then $T = \Omega(\log n)$.*

If the wire delay is linear then:

- *If $A \leq n^{2/3}$, then $A^2 T = \Omega(n^2)$.*
- *If $n^{2/3} \leq A \leq n$, then $A^3 T^3 = \Omega(n^4)$.*
- *If $n \leq A$, then $T = \Omega(n^{1/3})$.*

Proof. The proof of the corollary is entirely trivial; for various values of n and A ,

$$\frac{n}{A} \min_{q \geq 1} \left(\left(1 + \frac{n}{A} \right) q + \sqrt{\frac{A}{q}} \right) + \tau(n)$$

(see statement of Theorem 5) is minimized by selecting an appropriate value of q subject to $q \geq 1$. The actual derivation, nevertheless, is most instructive and therefore we will prove the first two claims of the corollary here. We consider the case where $A \leq n$. It is easy to see that $\tau(n)$ can be ignored and therefore as $n/A \geq 1$ we minimize $nq/A + \sqrt{A/q}$ subject to $q \geq 1$. We have two cases:

1. $A \leq n^{2/3}$. Here, as $q \geq 1$, we have $nq/A \geq n/A \geq n^{1/3} \geq \sqrt{A} \geq \sqrt{A/q}$. Thus, $q = 1$ and $T = \Omega((n/A)(n/A)) = \Omega(n^2/A^2)$,

2. $A > n^{2/3}$. In this case we “balance” $nq/A = \sqrt{A/q}$ obtaining $q = A/n^{2/3}$ and $T = \Omega((n/A)(n/A)(A/n^{2/3})) = \Omega(n^{4/3}/A)$. \square

As communicated to the author by J. Savage, part 1 of the corollary was conjectured by L. Valiant for at least some transitive functions [Va81].

Let us examine the proof of the first case. We have, in effect, deduced that the term $\sqrt{A/q}$ was not of significance in the range $A \leq n^{2/3}$, and thus *if using some different assumptions on the chip’s capabilities we could derive only the weaker formula*

$$T = \Omega\left(\frac{n}{A} \min_{q \geq 1} \left(\left(1 + \frac{n}{A} \right) q \right) + \tau(n)\right),$$

it would be as strong as the original one in the range of interest. Comparing with the proof of Lemma 6, we see that the term $\sqrt{A/q}$ was obtained on the basis of accommodating the internal movement of information (using internal wiring), and thus even if we assume that “wires are free,” the same lower bound would hold. After showing that the lower bounds are tight (§ 3) and making the argument more formal, it is possible to conclude that if $A \leq n^{2/3}$ then wires are not a critical resource. The fact that wires are not critical if the chip is small fits nicely with the fact that if the RAM of the computer is small then internal computations are not critical for sorting large sequences (an analogy introduced in § 1). A complete summary stating which resources are critical in various ranges is given at the end of § 4.

3. The upper bound.

THEOREM 9. *For each A greater than some constant, there exists a chip computing all circular shifts of an n -bit vector (a transitive function) such that*

$$T = O\left(\frac{n}{A} \min_{q \geq 1} \left(\left(1 + \frac{n}{A} \right) q + \sqrt{\frac{A}{q}} \right) + \tau(n)\right).$$

Proof. We do not present here the construction for the case of linear wire delay. The construction shown is a uniform construction for constant wire delay, and builds on the ideas of the construction presented in [KeZo81] for the special case $A = n^{3/2}$ and $T = \log n$. The interested reader is referred to that paper for a more detailed description of that special case to help him understand the general construction, as we only sketch here its more salient points.

Let u and w be such that $1 \leq w \leq u \leq n$. For simplicity we assume that w divides u and, u and w^2 divide n . By selecting only the $\log w$ smallest stages from the standard circular barrel shifter of u bits it is possible to construct a layout of size $O(u)$ by $O(w)$, which implements all circular shifts of a vector of u bits by $0, 1, \dots, w-1$ positions. Attach to each of the u in-terminals an H -tree of w leaves. The leaves of the trees will serve as input pads and memory cells capable of storing one bit each. This can be done so that the new layout is still of size $O(u)$ by $O(w)$. (Figure 3 shows an informal drawing for the case where $u = 8$ and $w = 4$.)

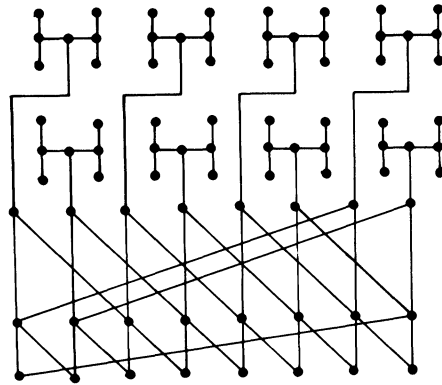


FIG. 3

We only describe the computation informally. Let the value of the shift be written as $s = \phi u + \psi$, for nonnegative integers ϕ and ψ , and $\psi \leq u - 1$. The computation will consist of $\max \{n/uw, 1\}$ phases. During each phase the leaves of each set of trees connected to the in-terminals $\{p, p+1, \dots, p+w-1\}$ for p that is divisible by w , read in all n inputs using a total of n/w^2 time units. Each tree stores in its leaves a total of $q = \min \{w, n/u\}$ of the inputs. The bits selected will, in effect form a circular shift of the input by ϕu positions. Then the stored inputs are pipelined from the leaves to the in-terminals and then into the shifter. The shifter will then perform a shift by ψ positions. This is done in $\max \{\log w, q\}$ time units.

Thus, as $\max \{a, b\} = a + b$ for nonnegative a and b , we get that $T = O((n/uw + 1)(n/w^2 + (\log w) + \min \{w, n/u\}))$. For each value of area size A some choice of u and w will give the claimed value of T . Specifically:

- If $A \leq n^{2/3}$, choose $u = w = \sqrt{A}$.
- If $n^{2/3} \leq A \leq n$, choose $u = A/n^{1/3}$, $w = n^{1/3}$.
- If $n \leq A \leq (n/\log n)^{3/2}$, choose $u = A^{2/3}$, $w = A^{1/3}$.
- If $(n/\log n)^{3/2} \leq A$, use only $(n/\log n)^{3/2}$ area and choose $u = n/\log n$, $w = (n/\log n)^{1/2}$. \square

Notice that our model allowed the chip to read in during the computation arbitrary parameters that were only dependent on the permutation and not on the vector to be permuted. One may object and say that this in effect allows us to input a different

program for each permutation, which is indeed the case. It is, however, possible to modify the above construction so that the area is just large enough to store the index of the permutation using area of $O(\log(n))$ and the parameters read are just sufficient to specify the permutation.

4. Conclusions and comparison with previous work. We have shown that for an arbitrary transitive group the lower bound

$$T = \Omega\left(\frac{n}{A} \min_{q \cong 1} \left(\left(1 + \frac{n}{A}\right)q + \sqrt{\frac{A}{q}} \right) + \tau(n)\right)$$

holds. Furthermore, we have exhibited transitive groups for which

$$T = \Theta\left(\frac{n}{A} \min_{q \cong 1} \left(\left(1 + \frac{n}{A}\right)q + \sqrt{\frac{A}{q}} \right) + \tau(n)\right)$$

for any feasible A , showing that for general transitive functions no additional lower bounds or tradeoff functions on area/time seem to be of interest (assuming our model). It is gratifying that such a tight result is obtained.

We have allowed the chip to read multiple copies of the inputs, in contrast with what is customarily assumed in theoretical VLSI research. We feel that a more liberal assumption is unavoidable if one is interested in computations of "large" functions, namely functions for which the number of variables n is larger than the on-chip memory. This case is of importance, as the size of a chip is bounded by a constant, and it should not be assumed that it can be made as large as one wishes. Just as it is of great interest and importance to study sorting of large sequences on a computer whose RAM is relatively small, we feel it is of interest to study computations of large functions on small chips. This paper deals with single-chip computations, ignoring system complexity, which is not addressed here.

We will now consider how our work relates to previously obtained results on lower bounds. We can consider only the time and the area (and thus we do not deal here with other measures of interest, such as the period or the power). We also do not discuss related work of A. Yao [Ya81], who studied the consequences of allowing arbitrary coding of the inputs.

It will be convenient to ignore the "boundary condition" function $\tau(n)$, so we will only consider the "partial" lower bound function

$$(*) \quad T = \Omega\left(\frac{n}{A} \min_{q \cong 1} \left(\left(1 + \frac{n}{A}\right)q + \sqrt{\frac{A}{q}} \right)\right).$$

We proceed to consider the implications of modifying the assumptions stated in our description of the model in order to obtain previous results. There will be several cases of interest (special cases of (*)), depending on additional restrictions on the permitted allocation of resources. We will also note the weaker assumptions on "what needs to be paid for," sufficient to prove those lower bounds. The fact that some known results can be proved using weaker, sometimes less realistic assumptions, may indicate to us that they do not completely characterize the complexity. Our discussion will be rather oversimplified, but the interested reader can easily derive the results more formally.

For each of the cases, we list only the *changes to our original model*, as described in § 1 (and which we encourage the reader to review). The various input assumptions deal here only with the variables being permuted. We again abuse the formalism by ignoring constant factors as much as possible.

Case 1. The modified assumptions.

- *Input.* In one time unit an input pad of unit area can read in an unbounded number of bits. However, each input bit can be read into only one input pad, possibly more than once.

- *Memory.* One unit of area can store an unbounded number of bits.

- *When-indeterminate.* The I/O schedule is not “completely fixed,” it is when-indeterminate (but still where-determinate).

(It was precisely in order to be able to discuss this and similar cases that we treated the input and the output requirements separately in our model.) Under these assumptions, it can be shown that the following weaker version of (*) holds:

$$T = \Omega\left(\frac{n}{A} \min_{q \geq 1} \left(q + \sqrt{\frac{A}{q}}\right)\right)$$

with the additional restriction that $q = 1$. Thus, $T = \Omega((n/A)(1 + \sqrt{A})) = \Omega((n/A)\sqrt{A}) = \Omega(n/\sqrt{A})$, or

$$AT^2 = \Omega(n^2).$$

This is the lower bound, of among others, Thompson [Th79], Brent and Kung [BrKu81] (but see case 2 below), Vuillemin [Vu83], Chazelle and Monier [ChMo81] and Savage [Sa81] (who used a model similar to the modification above): We stress that such a strong lower bound can be proved without any accounting for memory.

Case 2. The modified assumptions.

- *Input.* In one time unit an input pad of unit area can read in an unbounded number of bits. However, each input bit can be read only once.

We sketch the argument somewhat imprecisely. We are dealing here with a restricting modification of Case 1, forcing $(1 + n/A)q = 1$ (actually, of course, $O(1)$), which is strictly stronger than Case 1. As $q \geq 1$, it follows that $q = 1$ and $A \geq n$. Again, $T = \Omega((n/A)(1 + \sqrt{A})) = \Omega(n/\sqrt{A})$, or $AT^2 = \Omega(n^2)$, in addition to $A = \Omega(n)$. Thus the argument of Brent and Kung [BrKu81] holds, showing that $AT^{2\alpha} = (AT^2)^\alpha A^{1-\alpha} = \Omega(n^{2\alpha} n^{1-\alpha}) = \Omega(n^{1+\alpha})$, for any $\alpha = [0, 1]$.

It is important to note that as it is not reasonable to assume that chip area can be made arbitrarily large, the inequality $A \geq n$ seems to imply that large functions *cannot* be computed on a single chip even given an unlimited number of additional chips used as auxiliary storage. As discussed in § 1, this would be analogous to allowing sorting of sequences only on computers whose RAM is at least as large as the sequences.

Case 3. The modified assumption.

- *Memory.* One unit of area can store an unbounded number of bits.

Under this assumption, the weaker version of (*):

$$T = \Omega\left(\frac{n}{A} \min_{q \geq 1} \left(q + \sqrt{\frac{A}{q}}\right)\right)$$

(without the additional restriction that $q = 1$, used in Case 1) applies. Then, $T = \Omega((n/A)A^{1/3})$, giving $A^2 T^3 = \Omega(n^3)$.

This is the optimal lower bound if $n \leq A \leq (n/\log n)^{3/2}$ (see Corollary 8).

Note also that

$$T = \Omega\left(\frac{n}{A} \min_{q \geq 1} \left(q + \sqrt{\frac{A}{q}}\right)\right) = \Omega\left(\frac{n}{A} \min_{q \geq 1} \left(q + \frac{\sqrt{A}}{q}\right)\right) = \Omega\left(\frac{n}{A} A^{1/4}\right)$$

giving $A^3 T^4 = \Omega(n^4)$. This lower bound which is not tight is due to Kedem and Zorat [KeZo81]. (For arguments on similar issues, see also [LiSe81].)

Case 4. The modified assumptions.

- *Wires.* Wires are not necessary, because of the following assumption:
- *Computation.* In one time unit every computational element of unit area can compute an arbitrary boolean function of all the information stored in the whole chip.

Under these assumptions, it can be shown (as indicated in the proof of Corollary 8) that the following weaker version of (*):

$$T = \Omega\left(\frac{n}{A} \min_{q \geq 1} \left(\frac{n}{A} q\right)\right),$$

applies. From here $T = \Omega((n/A)(n/A))$, giving $A^2 T = \Omega(n^2)$.

This is the optimal lower bound if $A \leq n^{2/3}$ (see Corollary 8).

Grigoryev [Gr76] proved a general result on boolean circuit complexity, which was later interpreted in the VLSI context by Valiant giving the same bound. It was also explored by Savage [Sa82]. It is interesting to note that such a “nongeometric” model also gives an optimal lower bound if $A \leq n^{2/3}$ (see Corollary 8). (The intuitive explanation for this is that the chip is so small, or alternatively the function so “large,” that the internal communication requirements are of lower order of magnitude than those of input and memory, as “everything is close to each other.”)

We can now re-examine the proof of Theorem 5 in order to see which resources needed to be accounted for in the proof. This will determine which computational resources are “critical” for a least some transitive functions.

If the wire delay is constant then:

- If $A \leq n^{2/3}$, the complexity is dominated by input and memory requirements.
- If $n^{2/3} \leq A \leq n$, the complexity is dominated by input, memory and wiring requirements.
- If $n \leq A \leq (n/\log n)^{3/2}$, the complexity is dominated by input and wiring requirements.
- Area greater than $(n/\log n)^{3/2}$ cannot be used to decrease T because of fan-in limitations.

If the wire delay is linear then:

- If $A \leq n^{2/3}$, the complexity is dominated by input and memory requirements.
- If $n^{2/3} \leq A \leq n$, the complexity is dominated by input, memory and wiring requirements.
- Area greater than n cannot be used to decrease T because of delay limitations.

This classification of critical resources helped also in the design of optimal chips in the proof of Theorem 9.

As stated above, for $A \leq n^{2/3}$ both input and memory are the critical resources, and thus it may be useful to balance the area devoted to I/O pads with the area devoted to memory. One can object and say that whereas the chip designer can allocate the chip area between (internal) wires and memory, he cannot assume that the system designer will supply any requested number of (external) wires to the chip. We do not consider system level complexity here, but it may be worthwhile to mention that if the area devoted to the pads is constant, then the lower bound $AT = \Omega(n^2)$ holds.

Acknowledgments. The comments of anonymous referees in addition to stimulating conversations with Mike Atallah, John Savage, Alan Siegel, Clark Thompson, Jeff Ullman and Alessandro Zorat are gratefully acknowledged.

REFERENCES

- [Ar80] B. W. ARDEN, ed., *What Can Be Automated? The Computer Science and Engineering Research Study (COSERS)*, MIT Press, Cambridge, MA, 1980.
- [BrKu81] R. P. BRENT AND H. T. KUNG, *The area-time complexity of binary multiplication*, J. Assoc. Comput. Mach., 28 (1981), pp. 521-534.
- [CHMo81] B. CHAZELLE AND L. MONIER, *A model of computation for VLSI with related complexity results*, Proc. 13th Annual ACM Symposium on Theory of Computing, May 1981, pp. 318-325.
- [Gr76] D. YU. GRIGORYEV, *An application of separability and independence notions for proving lower bounds on circuit complexity*, Notes of Scientific Seminars, Steklov Math. Inst., 60 (1976), pp. 35-48. (In Russian.)
- [KeZo81] Z. M. KEDEM AND A. ZORAT, *On relations between input and communication-computation in VLSI*, Proc. 22nd Annual Symposium on Foundations of Computer Science, October 1981, pp. 37-44.
- [Kn73] D. E. KNUTH, *The Art of Computer Programming* Vol. III: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
- [LeMe81] T. LENGAUER AND K. MEHLHORN, *On the complexity of VLSI computations*, in VLSI Systems and Computations, H. T. Kung, B. Sproull, and G. Steele, eds., Computer Science Press, Potomac, MD, 1981, pp. 89-99.
- [Le80] C. E. LEISERSON, *Area-efficient graph layouts (for VLSI)*, Proc. 21st Annual Symposium on Foundations of Computer Science, October 1980, pp. 270-281.
- [LiSe81] R. J. LIPTON AND R. SEDGEWICK, *Lower bounds for VLSI*, Proc. 13th Annual ACM Symposium on Theory of Computing, May 1981, pp. 300-307.
- [Ro81] A. L. ROSENBERG, *Three-dimensional integrated circuitry*, in VLSI Systems and Computations, H. T. Kung, B. Sproull, and G. Steele, eds., Computer Science Press, 1981, pp. 69-80.
- [Sa81] J. E. SAVAGE, *Planar circuit complexity and the performance of VLSI algorithms*, in VLSI Systems and Computations, H. T. Kung, B. Sproull, and G. Steele, eds. Computer Science Press, Potomac, MD, 1981, pp. 61-68.
- [Sa82] ———, *Bounds on the performance of multilevel VLSI algorithms*, Rep. No. CS-82-10, Dept. Computer Science, Brown Univ., Providence, RI, March 1982, pp. 1-28.
- [Th79] C. D. THOMPSON, *Area-time complexity for VLSI*, Proc. 11th Annual ACM Symposium on Theory of Computing, April 1979, pp. 81-88.
- [Va81] L. VALIANT, Unpublished, brought to the author's attention by J. Savage.
- [Vu83] J. VUILLEMIN, *A combinatorial limit to the computing power of VLSI circuits*, IEEE Trans. Comput., C-32 (1983), pp. 294-300.
- [Ya81] A. YAO, *The entropic limitations on VLSI computations*, Proc. 13th Annual ACM Symposium on Theory of Computing, May 1981, pp. 308-311.

SCHEDULING FILE TRANSFERS*

E. G. COFFMAN, JR†, M. R. GAREY†, D. S. JOHNSON† AND A. S. LAPAUGH‡

Abstract. We consider a problem of scheduling file transfers in a network so as to minimize overall finishing time. Although the general problem is NP-complete, we identify polynomial time solvable special cases and derive good performance bounds for several natural approximation algorithms, assuming the existence of a central controller. We also show how these bounds can be maintained in a distributed regime.

Key words. Edge coloring, protocols, NP-completeness, approximation algorithms

1. Introduction. In this paper we study a fundamental problem of distributed processing, that of transferring large files between various nodes of a network. In particular, we are interested in how collections of such transfers can be scheduled so as to minimize the total time for the overall transfer process.

In our model, an instance of the problem consists of a labeled, undirected multi-graph $G = (V, E)$, which we shall call the *file transfer graph*. Both the vertices and edges of this graph are labeled with integers. Vertices correspond to computers/communication centers, each of which is assumed to have the ability to communicate directly with every other center. The label $p(v)$ of a vertex v is its *port constraint*, and denotes the maximum number of simultaneous file transfers that the given vertex can engage in. Edges correspond to the files to be transferred, with the label $L(e)$ of an edge e representing the amount of time needed to transfer that file. This is assumed to be independent of the time at which the file is transferred or the ports involved. Forwarding is not allowed; each file is transferred directly between the centers that are its endpoints. We also assume that once the transfer of a file begins, it continues without interruption until the transfer is complete.

This model is relevant to a variety of situations. For instance, consider a network of home computers, where each computer has an automatic dialer and interconnection is accomplished via a standard telephone network. This would correspond to our model with $p(v) = 1$ for each vertex. On a larger scale, one could consider the computer centers of a large company, where each has many automatic dialers.

In this paper, we concentrate on the problem of minimizing the *makespan* of the schedule (the time interval between the beginning of the first transfer and the completion of the last transfer). In the dial-up applications, this might be motivated by the need to make all transfers during periods of low usage (or low telephone rates), and hence to find schedules which are "short" enough to fit into such intervals.

In §2 and §3 of this paper we shall consider the construction of schedules by a single *central controller* that, given the file transfer graph, constructs an overall schedule in advance. In §2 we present complexity results and algorithms for the problem of finding an optimal schedule under various restrictions on the problem instance. In §3 we analyze approximation algorithms that guarantee near-optimal solutions for those cases where finding an optimal solution is too difficult.

Results for a central controller give us an idea of the best schedules one could hope to obtain. However, in many applications no single processor knows all the

*Received by the editors April 14, 1984, and in revised form October 2, 1984. This paper was typeset at AT&T Bell Laboratories, Murray Hill, New Jersey, using the *troff* program running under the UNIX® operating system. Final copy was produced on May 14, 1985.

†AT&T Bell Laboratories, Murray Hill, New Jersey 07974

‡Princeton University, Princeton, New Jersey 08544

details of the files to be transferred. In §4 we consider what is possible in the perhaps more common case of distributed control, where the schedule is constructed on the fly in a distributed fashion, with each vertex knowing only its local part of the file transfer graph (and perhaps not all of that in advance), and with the possibility that vertices (and communication lines) may not be reliable. We conclude in §5 by listing some directions for future research.

2. Complexity results and solvable subcases. Given a file transfer graph $G = (V, E)$, a *schedule* can be viewed formally as a function $s : E \rightarrow [0, \infty)$ that assigns to each edge e a start time $s(e)$, such that, for each vertex v and time $t \geq 0$,

$$|\{e : v \text{ is an endpoint of } e \text{ and } s(e) \leq t < s(e) + L(e)\}| \leq p(v).$$

The *length* or *makespan* of a schedule s is the largest finishing time, i.e., the maximum, over all edges e , of $s(e) + L(e)$. Figure 1 illustrates file transfer graphs and the timing diagrams to be used in representing schedules. Our goal is to find, given a file transfer graph G , a schedule s with the minimum possible makespan.

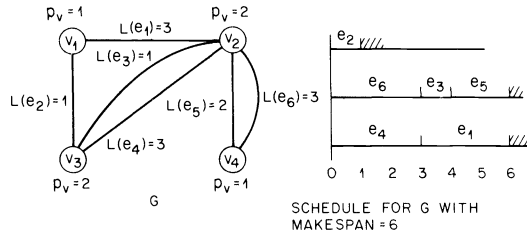


FIG 1. A file transfer graph and a schedule.

An elementary lower bound is obtained as follows. Let E_u denote the set of files that are to be sent or received by vertex u . The degree of node u is given by $d_u = |E_u|$. Let $E_{u,v}$ denote $E_u \cap E_v$, the set of files to be communicated between u and v . We define $\Sigma_u = \sum_{e \in E_u} L(e)$ and $\Sigma_{u,v} = \sum_{e \in E_{u,v}} L(e)$. For consistency with this notation, we shall also use p_u to denote the port constraint $p(u)$ for each vertex u .

The time to transmit all of the files sent or received by vertex u is at least $\lceil \Sigma_u / p_u \rceil$. Thus we have the following:

LEMMA 1. The optimal schedule length $OPT(G)$ for any graph G must satisfy

$$OPT(G) \geq \max_u \left\lceil \frac{\Sigma_u}{p_u} \right\rceil.$$

Note that this lower bound is achieved by the schedule in Fig. 1, where $T = \lceil \Sigma_{v_2} / p_{v_2} \rceil = 12/2 = 6$. Figure 2 illustrates the fact that $OPT(G)$ can be substantially larger than $\max_{u \in V} \lceil \Sigma_u / p_u \rceil$. Here the value of the bound is 2, but the following simple analysis shows that a makespan of 2 is not achievable.

In order to finish the three files of any one triangle in 2 time units, a two-port vertex of that triangle has to transmit two of the triangle's three files simultaneously, during one or the other time unit. Since there are only 2 two-port vertices, two of the triangles must be transmitting two files in the same time unit. This can be done only

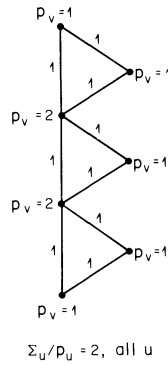


FIG 2. An example where $OPT(G) > \max_{u \in V} \{\Sigma_u / P_u\}$.

by the first and third triangles, each transmitting the files incident with its two-port vertex. But during this time unit no file of the middle triangle can be in transmission. Since this triangle requires 2 time units, we thus have $OPT(G) > 2$. In fact, it is easy to see that $OPT(G) = 3 = (3/2) \max_{u \in V} \lceil \Sigma_u / P_u \rceil$.

It is not always “easy to see” what $OPT(G)$ is, however. The general decision problem “Given G and a bound B , is there a schedule s for G with makespan B or less?” is NP-complete and hence unlikely to be solvable efficiently, i.e., by a polynomial time algorithm. (For a discussion of NP-completeness and its implications, see [8].) Let us refer to this decision problem as FILE TRANSFER SCHEDULING. In the remainder of this section we will consider the complexity of this problem and some of its more interesting special cases.

Before we begin our search for polynomial time solvable subproblems of FILE TRANSFER SCHEDULING, it is important to note that there are two distinct reasons why the general problem is NP-complete, one having to do with the structural complexity of the file transfer graph and the other with the complexity involved in balancing the transmission of files of different lengths. We shall illustrate this by proving that two very restricted versions of the problem are NP-complete.

THEOREM 1. FILE TRANSFER SCHEDULING is NP-complete even when restricted to file transfer graphs in which the port capacity $p(v)$ of each vertex is 1, there is at most one edge between each pair of vertices, and all edges have the same length.

Proof. The problem is in NP because the general problem is: a nondeterministic algorithm need only guess a schedule and check (in polynomial time) that it is a valid schedule and meets the specified bound on makespan. Thus all we must show in this and future NP-completeness proofs is that some known NP-complete problem is polynomially transformable into the current one.

In this case the known NP-complete problem is CHROMATIC INDEX [11]: “Given a graph G and a positive integer k , can the edges of G be colored with k or fewer colors so that no two edges with the same color share a common endpoint?” This can be directly translated into the current special case, for if we view G as a file transfer graph with edges of length 1 and port capacities of 1, then an edge coloring corresponds to a schedule, with all the edges of the same color being scheduled during the same time unit and with the makespan equalling the total number of colors used.

■

THEOREM 2. FILE TRANSFER SCHEDULING is NP-complete even when restricted to file transfer graphs with just two vertices, u and v .

Proof. In this case the transformation is from MULTIPROCESSOR SCHEDULING [8]: "Given a set T of tasks, each task t having an integer length $L(t)$, a number m of processors, and a bound B , can the tasks be partitioned among the m processors so that the total length of the tasks assigned to any one processor never exceeds B ?"

In this case, we let $p(u) = p(v) = m$ and create $|T|$ edges between u and v , one for each task, letting the length of an edge equal the length of the corresponding task. It is then easy to see that a schedule with makespan B or less will exist if and only if the desired task partition exists. Note that, since MULTIPROCESSOR SCHEDULING is NP-complete for any fixed value of m greater than 1 [8], FILE TRANSFER SCHEDULING is thus NP-complete for any fixed bound of 2 or more on the port capacities. Like MULTIPROCESSOR SCHEDULING for an arbitrary number of processors, it is NP-complete "in the strong sense" [8] if port capacities are arbitrary. ■

Theorems 1 and 2 considered restrictions on FILE TRANSFER SCHEDULING involving edge lengths, the possibility of multiple edges between two vertices, bounds on port capacities, and the structure of the file transfer graph. We have conducted a more thorough investigation of the complexity of the problem in terms of these parameters, and our results are summarized in Tables 1 and 2. Table 1 lists results for the case where all edge lengths are equal and hence structural considerations dominate; Table 2 considers the case where arbitrary edge lengths are allowed. The other parameters are as follows:

1. File transfer graph structure; special cases include bipartite graphs, trees, cycles, and paths. Such restrictions might well arise in physical networks with restricted interconnection patterns, where nodes only communicate with their neighbors.
2. Port constraints; a single port per vertex will be the special case, corresponding for example to the personal computer application mentioned above.
3. Edge multiplicity; as a special case we will consider graphs having at most one edge between each pair of vertices. This restriction is often imposed in practice, e.g., when only a single telephone call between any pair of vertices is feasible. In this case an edge of the file transfer graph represents *all* the files to be transferred between its two endpoints.

Entries in the tables are either "NPC" for NP-complete, "?" for "Open," or an upper bound on running time (in those cases where polynomial time algorithms have been found). Theorem(s) that imply the result are also indicated. The theorems themselves (other than the already-seen Theorems 1 and 2) are presented in the following two subsections, one devoted to each table.

2.1. Complexity results when all edge lengths are equal. Throughout this section we assume that all edge lengths are equal. The only NP-completeness results known under this assumption are for special cases where general graphs are allowed, and are implied by Theorem 1. Thus this section will be devoted to presenting polynomial time algorithms for cases where the graph structure is restricted. The polynomial time algorithms for the case of bipartite graphs all come from the following theorem, which is based on the same analogy with edge coloring used in Theorem 1.

THEOREM 3. *If G is bipartite and all edge lengths are equal, a minimum makespan schedule can be found in polynomial time, even when multi-edges and arbitrary port capacities are allowed.*

Proof. When only one port per vertex is allowed, the result follows from known

TABLE 1

Complexity classification for FILE TRANSFER SCHEDULING when all edge lengths are equal

| EQUAL LENGTH EDGES | One Port | | Arbitrary Ports | |
|-----------------------|-----------------------------------|-----------------------------------|---------------------------|---------------------------|
| | Single Edges | Multi-Edges | Single Edges | Multi-Edges |
| General Graphs | NPC [Theorem 1] | NPC [Theorem 1] | NPC [Theorem 1] | NPC [Theorem 1] |
| Bipartite Graphs | $O(E \cdot V)$ [Theorem 3] | $O(E \cdot V)$ [Theorem 3] | $O(E ^2)$ [Theorem 3] | $O(E ^2)$ [Theorem 3] |
| Trees | $O(E)$ [Theorem 4] | $O(E)$ [Theorem 4] | $O(E)$ [Theorem 4] | $O(E)$ [Theorem 4] |
| Paths and Even Cycles | $O(E)$ [Theorem 5] | $O(E)$ [Theorem 5] | $O(E)$ [Theorem 5] | $O(E)$ [Theorem 5] |
| Odd Cycles | $O(E)$ [Exercise] | $O(E)$ [Theorem 6] | $O(E)$ [Exercise] | ? |

TABLE 2

Complexity classification for FILE TRANSFER SCHEDULING when arbitrary edge lengths are allowed

| ARBITRARY EDGE LENGTHS | One Port | | Arbitrary Ports | |
|------------------------|--------------------------|-------------------------|--------------------------|-----------------------|
| | Single Edges | Multi-Edges | Single Edges | Multi-Edges |
| General Graphs | NPC [Theorem 1] | NPC [Theorem 1] | NPC [Theorem 1] | NPC [Theorems 1,2] |
| Bipartite Graphs | NPC [Theorem 7] | NPC [Theorem 7] | NPC [Theorem 7] | NPC [Theorems 2,7] |
| Trees | NPC [Theorem 7] | NPC [Theorem 7] | NPC [Theorem 7] | NPC [Theorems 2,7] |
| Paths | $O(E)$ [Theorem 8] | $O(E)$ [Theorem 8] | $O(E)$ [Theorem 8] | NPC [Theorem 2] |
| Even Cycles | $O(E)$ [Theorem 8] | $O(E)$ [Theorem 8] | $O(E)$ [Theorem 8] | NPC [Theorem 2] |
| Odd Cycles | $O(E)$ [Theorem 10] | NPC [Theorem 9] | $O(E)$ [Theorem 10] | NPC [Theorem 2] |

algorithms for edge coloring of bipartite graphs and multigraphs, e.g., see [1],[3],[7]. We present an extension of these algorithms to handle the case of arbitrary ports. We may assume without loss of generality that the common edge length is 1, in which case the lower bound on makespan from Lemma 1 becomes $B = \max_{v \in V} [d_v/p_v]$. We shall prove that this lower bound is in fact achievable. Given the analogy to edge coloring explained in the proof of Theorem 1, all we need to do is show how to color any bipartite file transfer graph using B colors, where we allow p_v edges incident with any vertex v to share the same color. We prove this by showing how to modify any partial coloring of the edges of G to obtain a new partial coloring that colors one additional edge.

Let $\{u_0, v_0\}$ be any currently uncolored edge. Since this edge is incident to both

u_0 and v_0 , the choice of B implies that each of these two vertices has at least one color that has not been used a maximum number of times on the edges incident with that vertex. If some color is available at both u_0 and v_0 , we can color $\{u_0, v_0\}$ with that color and hence obtain our desired extended coloring. Otherwise, let "red" be an available color at u_0 , and let "blue" be an available color at v_0 . We will construct an alternating path $v_0, u_1, v_1, u_2, v_2, \dots, x_k$, where x_k is either u_k or v_k , such that each edge $\{v_i, u_{i+1}\}$ is currently colored red and each edge $\{u_i, v_i\}$ is currently colored blue. Moreover, x_k will have color red available to it in the current coloring if $x_k = v_k$, and will have blue available to it if $x_k = u_k$. Note that, since G is bipartite, this means that x_k is neither v_0 nor u_0 .

If we have such a path, we can obtain a new legal partial coloring by interchanging the colors red and blue along the path. In this new coloring, the color red becomes available at v_0 and remains available at u_0 , and so we can extend the coloring by coloring $\{u_0, v_0\}$ red, thus obtaining our desired extension of the original coloring. All that remains to be shown is how to construct the path.

Since the color red is not available at v_0 , we can start the path by choosing some red edge incident with v_0 , calling its other endpoint u_1 . In general, suppose we have constructed the partial augmenting path $v_0, u_1, v_1, \dots, u_j, v_j$ (the case in which the last vertex is u_j is symmetric and will not be detailed here). If v_j has the color red available, we can halt with $x_j = v_j$. If v_j does not have the color red available, then it must have its full capacity of red edges in the current coloring and there must be some edge incident with v_j that is colored red and that does not belong to the path constructed so far: if $v_j = v_0$ this follows because v_0 by assumption has at least one fewer blue edge than capacity allows, and so far the path contains an equal number of blue and red edges incident with v_0 ; otherwise it follows because the path currently contains one more blue edge than red edges incident with v_j . Choose such an unused red edge to add to the path, and label its other endpoint u_{j+1} . Combining this with the symmetric argument for extending a path ending at u_j , we see that we can always extend the path whenever the desired termination condition fails to hold. Since there is a bound of $|E|$ on the length of the path, at some point in our procedure the desired termination condition must hold.

Thus, the desired augmenting path always exists, and we can certainly find it in time proportional to the minimum of $|E|$ and $|V| \cdot \max_{v \in V} p(v)$ following the above method. We thus have an overall running time bound of $O(\min\{|E|^2, |E| \cdot |V| \cdot \max_{v \in V} p(v)\})$. Standard methods for constructing augmenting paths more efficiently [3],[7] seem likely to extend to the above, and so substantial improvements to this bound are likely to exist [6]. ■

This result immediately implies that all the remaining special cases in Table 1, except those involving odd cycles, can be solved in polynomial time. However, those special cases can in fact be solved with even faster algorithms.

THEOREM 4. *If G contains no simple cycles other than the trivial 2-vertex cycles induced by multiple edges (i.e., if G becomes a forest when multiple edges are coalesced), and if all edge lengths are equal, then a minimum makespan schedule can be found in time $O(|E|)$.*

Proof. If G is not connected, the schedules for its connected components are independent and hence can be constructed separately. Thus all we need to show is how to do the scheduling when G is connected (and hence a tree). Without loss of generality assume that all edges have unit length. Recall that $\max_{u \in G} \lceil \sum_u / p_u \rceil$ is a lower bound on schedule length when G has arbitrary port constraints. Here, with unit-length edges, the bound specializes to $B = \max_{u \in G} \lceil d_u / p_u \rceil$. The algorithm

below achieves B .

For each vertex assume that the edges (files) are ordered so that multiple edges to the same neighboring vertex are consecutive in the ordering. We label the edges of the tree as follows.

We begin with an arbitrary vertex, v , and label the edges in their given order with the first B nonnegative integers, used cyclically in the sequence $0, 1, \dots, B-1, 0, 1, \dots, B-1, 0, 1, \dots$. Figure 3 shows an example.

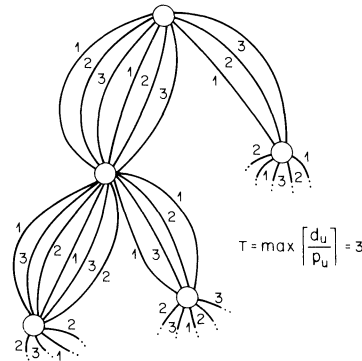


FIG 3. The edge labelling used in Theorem 4.

Now suppose v is any vertex whose edges have all been labeled and w is a vertex adjacent to v , some of whose edges are not yet labeled. (Since G is a tree, we can proceed in such a way that the only edges involving w that are labeled are those with v as the other endpoint, i.e., those in $E_{v,w}$.) If $j, 0 \leq j < B$, is the label of the last edge in $E_{v,w}$ to be labeled, then the labeling of w 's remaining edges begins with $j+1$ (or 0 if $j = B-1$) and proceeds cyclically through the sequence $0, 1, \dots, B-1$ as before.

In this way we repeatedly choose a vertex w adjacent to a completely labeled vertex v and then complete the labeling for edges incident on w . It is easy to see that, no matter what order we choose for the vertices and edges consistent with the above description, the edges of each vertex u will be labeled in some sequence i_1, i_2, \dots, i_k , where $0 \leq i_1 < B$ and $i_{j+1} = i_j + 1 \pmod{B}$. The number of times an integer appears in this sequence of d_u integers is at most $\lceil d_u/B \rceil$. By definition of B we have

$$\lceil d_u/B \rceil \leq \left\lceil \frac{d_u}{\max_{v \in V} \lceil d_v/p_v \rceil} \right\rceil \leq \left\lceil \frac{d_u}{\lceil d_u/p_u \rceil} \right\rceil \leq \left\lceil \frac{d_u}{d_u/p_u} \right\rceil = p_u.$$

Thus, if we begin transmitting every file (edge) labeled i at time $i, 0 \leq i < B$, we will have a valid schedule for G that is finished at time B and hence has B for its makespan. Since our algorithm obviously has a linear running time, the theorem is proved. ■

If the file transfer graph is a path, then the algorithm of Theorem 4 applies because a path is a tree. Thus the only results from Table 1 that remain unproved are those for cycles. For even cycles with both arbitrary ports and multi-edges allowed, we rely on a hybrid of the algorithms in Theorems 3 and 4 to obtain a linear time algorithm.

THEOREM 5. *If G is an even cycle with multi-edges, all of equal length, and arbitrary port capacities, then a minimum makespan schedule can be found in linear time.*

Proof. Pick a pair (u, v) of adjacent vertices in the cycle which has the minimum value for $|E_{u,v}|$. Note that this value is at most $|E|/|V|$. Deleting the edges of $E_{u,v}$, we obtain a path and can apply Theorem 4 to obtain a minimum makespan schedule in linear time. This can then be viewed as a partial edge coloring for G in the sense of Theorem 3, which we wish to extend to the edges of $E_{u,v}$, thus obtaining a schedule meeting the Lemma 1 lower bound on makespan. (We know this bound is attainable because even cycles are bipartite.) By the algorithm of Theorem 3, this will involve the finding of $|E_{u,v}|$ augmenting paths. Since each of these can be found in time $O(|V|)$ when G is an even cycle with multi-edges (an exercise we leave to the reader), the overall time for constructing all the paths (and hence completing our minimum makespan schedule) is $O(|V| \cdot (|E|/|V|)) = O(|E|)$, as claimed. ■

Surprisingly, the situation becomes considerably more complicated when even cycles are replaced by odd ones. Although the equal-length, single edge case remains fairly straightforward (and we leave the derivation of linear time algorithms as an exercise), the equal-length, arbitrary port capacity, multi-edge problem remains open and the restriction of this to single ports, although solvable in linear time, requires a new approach:

THEOREM 6. *If G is an odd cycle with multi-edges, with all edge lengths equal, and with all port capacities equal to 1, then a minimum makespan schedule can be found in linear time.*

Proof. Let us once again view our problem as one of edge coloring, as in Theorem 3. By a result in [1, p. 255], the number of colors needed for the edges of an odd cycle with n vertices and $|E|$ multi-edges is

$$c(G) = \max \left\{ \max_v d_v, \left\lceil \frac{2|E|}{n-1} \right\rceil \right\}.$$

The following algorithm, based on the proof of the above result, constructs a coloring with precisely this number of colors.

1. Set $G_0 = G$, $i = 0$.
2. While G_i contains a copy of each edge of the cycle, do
 - Find a maximum matching M from G_i , with the one vertex not covered by any edges of M being a vertex of minimum degree in G_i . Schedule all edges in M to start at time i . Construct G_{i+1} by deleting all edges of M from G_i .
3. Now G_i must be a collection of 0 or more disjoint paths, and we can find a schedule for it starting at time i and using exactly $\max_{v \in V} d_v$ time units in linear time by the algorithm of Theorem 4. This completes our construction of the schedule.

That this algorithm runs in linear time follows from the fact that, so long as G_i contains a copy of each edge of the cycle, the desired maximum matching M contains $(n-1)/2$ edges and can easily be found in time proportional to n . That the constructed schedule has makespan $c(G)$ follows from the fact that $c(G_{i+1}) = c(G_i) - 1$ for all $i > 0$, which we will now demonstrate. Note that every time we delete a maximum matching, the quantity $2|E|/(n-1)$ decreases by 1, since the size of a maximum matching is always $(n-1)/2$ in Step 2. Thus the only way $c(G_i)$ could fail to decrease is if $\max_{v \in V} d_v \geq 2|E|/(n-1)$ and all vertex degrees are equal to this maximum degree. However, this is impossible: If all vertices have the same degree d , then the total number of edges is $dn/2$, and so

$$\frac{2|E|}{(n-1)} = \frac{dn}{n-1} > d$$

Thus our claim about $c(G_{i+1})$ holds and, by induction, the algorithm produces a schedule with makespan $c(G)$, the optimal value. ■

This completes our consideration of special cases where all edge lengths are equal.

2.2. Complexity results for arbitrary edge lengths. Let us now turn to Table 2 and those results for arbitrary edge lengths that are not already implied by Theorems 1 and 2. The following theorem shows that, when arbitrary edge lengths are allowed, NP-completeness cannot be banished even if we restrict our attention to trees.

THEOREM 7. FILE TRANSFER SCHEDULING is NP-complete even when G is a tree of single port vertices without multi-edges.

Proof. Our reduction uses the 3-PARTITION problem: Given $A = \{a_1, \dots, a_{3k}\}$, with $B/4 < a_i < B/2$, $1 \leq i \leq 3k$, where $B = (1/k) \sum_{i=1}^{3k} a_i$, does there exist a partition $A = A_1 \cup A_2 \cup \dots \cup A_k$ such that $\sum_{a \in A_i} a = B$, $1 \leq i \leq k$? (Note that this is again a special case of MULTIPROCESSOR SCHEDULING.) Assume without loss of generality that k is odd and $k = 2n + 1$, and let $C_i = iB + i - 1$, $1 \leq i \leq n$. Given an instance of the above form, we construct a file transfer graph as illustrated in Fig. 4.

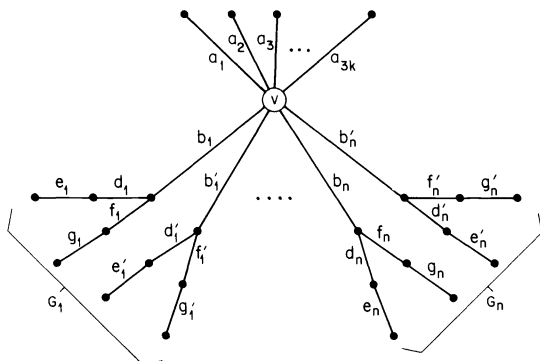


FIG 4. Tree used in proof of Theorem 5.

The tree is constructed around a hub vertex v , on which are incident (i) $3k$ edges with lengths a_1 through a_{3k} and (ii) n subtrees G_1 through G_n , each G_i having two unit-length edges, b_i and b'_i , incident with v . Let $M = \Sigma_v = \sum_{i=1}^{3k} a_i + 2n$. (Note that M is a lower bound on the optimum makespan, since v has a single port.) The lengths of the remaining edges in G_i are as follows (for simplicity we shall use the same symbol to denote both the edge and its length):

$$\begin{aligned} g_i &= g'_i = M - C_i, & 1 \leq i \leq n, \\ f_i &= f'_i = C_i, & 1 \leq i \leq n, \\ e_i &= e'_i = C_i + 1, & 1 \leq i \leq n, \\ d_i &= d'_i = M - C_i - 1, & 1 \leq i \leq n. \end{aligned}$$

We shall show that this tree of single-port vertices can be scheduled with makespan equal to the lower bound M if and only if the desired 3-partition exists. NP-completeness will then follow in the usual way.

Consider the scheduling of G_i , for any i , under the assumption that G can be scheduled in time M . Clearly, since $d_i + e_i = M$, transmission of d_i must either begin at $t = 0$ or end at $t = M$. By similar arguments, this must also hold for f_i , d'_i , and f'_i . Since d_i and f_i cannot both start at time 0 because they share an endpoint, one must start at time 0 and the other end at time M . This means that b_i must be scheduled in the interval between them, either at time C_i or at time $M - C_i - 1$, depending on whether f_i or d_i goes first. A similar argument holds for b'_i , so between the two tasks, each time slot $[C_i, C_i + 1]$ and $[M - C_i - 1, M - C_i]$ must be occupied by one of the edges b_i and b'_i , as in Fig. 5a.

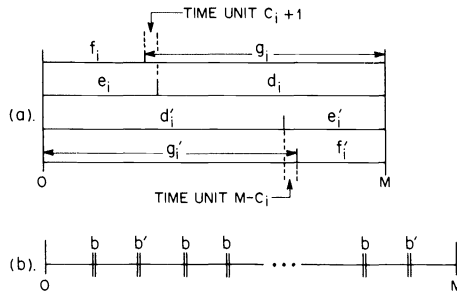


FIG 5. Scheduling the tree of Figure 4.

Applying this argument for all i , $1 \leq i \leq n$, we deduce that the schedule at vertex v must look as in Fig. 5b, when restricted to tasks of types b and b' . Noting that $C_{i+1} - (C_i + 1) = B$, $1 \leq i < n$, and that $[M - (nB + n)] - (nB + n) = M - 2nB - 2n = kB + 2n - 2nB - 2n = B$, we conclude that there are $2n + 1 = \kappa$ gaps left in the schedule, each of length exactly B , and into which the tasks of type a_i must be scheduled. It thus follows that G can be scheduled with makespan M if and only if there is a 3-PARTITION of A . ■

As Theorem 7 implies all the complexity results in Table 2 for cases where the graph is at least as complicated as a tree, the remaining results in this section concern only cycles and paths. As evidenced by the table, the distinction between odd and even cycles is here quite pronounced. Although the single-port, multi-edge case is solvable in linear time for even cycles, the same problem for odd cycles is NP-complete! The case of even cycles can be combined with that of paths, and is covered in the next theorem. (Note that we must forbid either multiple edges or arbitrary port capacities, since if both are allowed the problem becomes NP-complete even for a path of length 1, and hence also for cycles of any length, by Theorem 2.)

THEOREM 8. *If G is a path or even-length cycle with arbitrary edge lengths, and either all port capacities equal 1 or no multi-edges are present, then a minimum makespan schedule can be found in linear time.*

Proof. We first consider the case of port capacity 1. Observe that with single-port vertices the Lemma 1 lower bound on makespan simplifies to

$$OPT(G) \geq \max_u \Sigma_u . \tag{*}$$

This bound can be achieved rather simply in the present case.

Let $\{v_i, 0 \leq i \leq n-1\}$ be the vertices of the path or cycle, where v_i and v_{i+1} , $0 \leq i \leq n-2$, are adjacent. Nodes v_0 and v_{n-1} are also adjacent in the case of a cycle. Let C_i denote the set of edges between v_i and $v_{i+1 \pmod n}$, with C_{n-1} being

empty in the case of a path. Starting at $t = 0$ we schedule without interruption all of the edges in the sets C_i , i odd. The order within each set is unimportant. Each remaining edge set C_i , i even, is scheduled to start and proceed without interruption at the earliest possible time thereafter; i.e. as soon as all edges in $C_{i-1(\text{mod } n)}$ and $C_{i+1(\text{mod } n)}$ have been completed. Figure 6 shows examples.

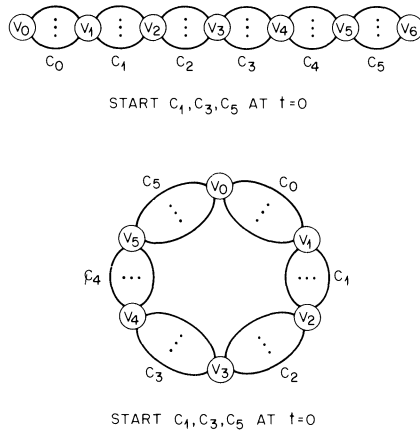


FIG 6. Illustrations for Theorem 8.

Since G is either a path or an even cycle, any schedule of the above type is clearly valid. To see that it is optimal, let C_k be an edge set with the latest finishing time. If C_k started at time 0, the makespan for the schedule is no more than Σ_{v_k} and hence must equal the above lower bound (*) and be optimal. Otherwise k must be even and the makespan must be

$$\max \left\{ \sum_{1e \in C_{k-1(\text{mod } n)}} L(e) + \sum_{1e \in C_k} L(e), \sum_{1e \in C_{k+1(\text{mod } n)}} L(e) + \sum_{1e \in C_k} L(e) \right\}.$$

But this is simply the maximum of Σ_{v_k} and $\Sigma_{v_{k+1(\text{mod } n)}}$ and hence must again equal the lower bound of (*) and be optimal.

Now suppose that at least one vertex v of G has port capacity $p(v)$ greater than 1, but that there are no multi-edges. Given that the file transfer graph is either a path or an even cycle, the lack of multi-edges means that no vertex is involved in more than two edges. We thus can proceed as follows: For each vertex v_k with port capacity 2 or greater, replace v_k by two copies, one adjacent to $v_{k-1(\text{mod } n)}$ and one adjacent to $v_{k+1(\text{mod } n)}$, giving each copy a port capacity of 1. This breaks G into a collection of paths, each of which has all port capacities equal to 1. By the above we can find minimum makespan schedules for each in linear time. These determine a schedule for the original graph G , which must be valid since all the duplicated vertices had port capacity at least 2. It is easy to see that it also must have the minimum possible makespan. ■

Turning now to odd cycles, we have the following results, depending on whether one forbids arbitrary port capacities (Theorem 9) or multiple edges (Theorem 10).

THEOREM 9. FILE TRANSFER SCHEDULING is NP-complete even when restricted to file transfer graphs that are 5-cycles of single-port vertices.

Proof. We shall make use of the NP-complete problem PARTITION: Given a

sequence $A = (a_1, a_2, \dots, a_n)$ of positive integers, does there exist a subset $A' \subseteq A$ such that $\sum_{a \in A'} a = (1/2) \sum_{i=1}^n a_i$. (Note that this corresponds to the special case of MULTIPROCESSOR SCHEDULING where there are just $m = 2$ processors.)

Given an instance A of PARTITION, the corresponding file transfer graph is shown in Fig. 7, where we use B to denote $(1/2) \sum_{i=1}^n a_i$, and our desired makespan is $5B$. This graph can clearly be constructed in polynomial time, so we can complete our proof by showing that a schedule with the makespan $5B$ exists if and only if the desired partition exists.

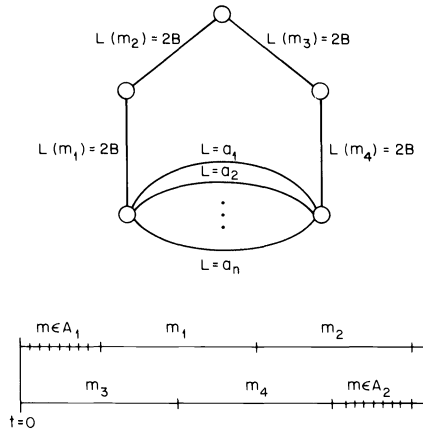


FIG 7. Illustration for Theorem 9.

First, suppose there is a partition $A_1 \cup A_2 = A$ such that $\sum_{a_i \in A_1} a_i = \sum_{a_i \in A_2} a_i = B$. The files are scheduled as shown in Fig. 7. It is easily verified that no edges with common endpoints are scheduled at the same time, and hence the schedule is a valid one of length $5B$.

Next, suppose there exists a schedule with a finishing time at most $5B$. Then m_1 and m_4 must overlap for exactly B units of time, because:

(1) If the overlap is less than B , then there are more than $3B$ time units during which either m_1 or m_4 is being sent and the files in A cannot be sent in parallel with either of these. This would imply that the schedule length is greater than $5B$.

(2) If the overlap is more than B , we observe that m_2 and m_3 cannot overlap at all with the interval when m_1 and m_4 are both being sent, and m_2 and m_3 cannot overlap each other. Thus, we again have that the schedule length is greater than $B + 2B + 2B = 5B$.

Without loss of generality, let us now assume that m_1 starts before m_4 . If m_3 were to start after m_4 (it cannot overlap with m_4), the schedule would already occupy $5B$ time units, with the endpoints of m_2 being simultaneously idle only in the interval from time $2B$ to time $3B$, an interval too short to schedule m_2 . Thus, m_3 starts (and ends) before m_4 . Similarly, m_2 starts after m_1 .

It follows that, in order to finish by time $5B$, the schedule must start m_3 at time 0 , m_1 at time B , m_4 at time $2B$, and m_2 at time $3B$, so it has the form of Fig. 7. The files in A must be sent during either the first B or last B time units, since they cannot overlap with either m_1 or m_4 . 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 have the desired partition. ■

It is not difficult to extend the above theorem to cycles with *any* fixed odd length $2k + 1$, $k \geq 2$. (For cycles of 3 single port vertices, it is easy to see that no two edges may be scheduled simultaneously, so all schedules with no idle time have optimal makespan.) However, if we restrict ourselves to single edges, the odd cycle problem once more becomes solvable in linear time, even if we allow multiple ports.

THEOREM 10. *If G is an odd cycle without multi-edges, then a minimum makespan schedule can be found in linear time.*

Proof. As in Theorem 6, the presence of a vertex with port capacity exceeding 1 reduces the problem to that of a path, for which we already have algorithms. Thus we may assume that G is a cycle of single port vertices. Let us label them v_0, v_1, \dots, v_{T-1} , where v_i is joined to $v_{i+1(\text{mod } T)}$ by the edge e_i , $0 \leq i \leq T-1$. We shall show that a lower bound on makespan is the maximum of the following two quantities:

$$LB1(G) \equiv \max \left\{ L(e_i) + L(e_{i+1(\text{mod } T)}): 0 \leq i \leq T-1 \right\},$$

$$LB2(G) \equiv \min \left\{ L(e_i) + L(e_{i+1(\text{mod } T)}) + L(e_{i+2(\text{mod } T)}): 0 \leq i \leq T-1 \right\},$$

and that a schedule with makespan equalling the larger of these two quantities can be found in linear time.

Note that $LB1(G)$ is just our old lower bound from Lemma 1, specialized to the current case. To see that $LB2(G)$ is also a lower bound, let s be a minimum makespan schedule. For each edge e_i , call e_i a *trailing* edge if $s(e_i) > s(e_{i+1(\text{mod } T)})$ (and hence $s(e_i) \geq s(e_{i+1(\text{mod } T)}) + L(e_{i+1(\text{mod } T)})$). Otherwise we call e_i a *leading* edge, and $s(e_{i+1(\text{mod } T)}) \geq s(e_i) + L(e_i)$. Since G is an odd cycle, there must be either two consecutive trailing edges or two consecutive leading edges. In either case, if e_j and $e_{j+1(\text{mod } T)}$ are these two edges, the makespan of s must be at least $L(e_j) + L(e_{j+1(\text{mod } T)}) + L(e_{j+2(\text{mod } T)})$, which is at least as large as $LB2(G)$.

To see that $\max \{LB1(G), LB2(G)\}$ is attainable, let j be chosen to minimize the sum $L(e_j) + L(e_{j+1(\text{mod } T)}) + L(e_{j+2(\text{mod } T)})$. Schedule e_j at time 0, along with the $(T-3)/2$ edges $e_{j-2(\text{mod } T)}, e_{j-4(\text{mod } T)}, e_{j-6(\text{mod } T)}, \dots, e_{j+5(\text{mod } T)}, e_{j+3(\text{mod } T)}$. Then schedule $e_{j+1(\text{mod } T)}$ to start at time $L(e_j)$, $e_{j+2(\text{mod } T)}$ to start at the maximum of $L(e_{j+3(\text{mod } T)})$ and $L(e_j) + L(e_{j+1(\text{mod } T)})$, and all remaining tasks to start as soon as their endpoints become available. It is straightforward to verify that the makespan for this schedule is $\max \{LB1(G), LB2(G)\}$. ■

We note in concluding this section that the proofs of Theorems 1 and 7 actually prove NP-completeness "in the strong sense" with all that implies [8], although those of Theorems 2 and 9 do not. This leaves open the possibility of "pseudo-polynomial time" algorithms [8] in the latter two cases. In fact it is easy to see that one exists in the case of two vertices (Theorem 2), although it is not clear how far such an approach can be generalized.

3. Approximation algorithms. In the previous section we identified a number of special cases where minimum makespan schedules can be found efficiently. In general, however, NP-completeness blocks our way to finding optimal schedules efficiently. A standard approach to use when confronted with such an obstacle is to search for good "approximation algorithms," algorithms that efficiently generate schedules, but that provide no guarantee of optimality (although a guarantee of "near-optimality" may be possible). In this section we shall consider such algorithms,

showing how classical results from the study of multiprocessor scheduling can be adapted to the more general problem of file transfer scheduling, and can provide both efficient algorithms and reasonable guarantees.

The main algorithm we shall study is *List Scheduling (LS)*. This algorithm assumes that the edges of the file transfer graph are ordered in a "priority" list as e_1, e_2, \dots, e_m . At time $t = 0$, and thereafter each time t an edge is finished, the list is scanned for "ready" edges, i.e. edges that have not yet been started and for which there is an available port at both endpoints. Whenever such an edge is encountered on the list, it is assigned starting time t , the two ports it uses become unavailable, and the scanning of the list is continued from that point. The list is always scanned in the given order, and ready edges are never delayed. The obvious implementation for List Scheduling is $O(|E|^2)$, since the list might have to be scanned once for each edge scheduled. However, a cleverer implementation can reduce this bound to $O(|V||E| + |E|\log|E|)$ (note that, since multiple edges are allowed, $|E|$ can be larger than $|V|^2$). The key to this bound is choosing the right data structures and doing the right "bookkeeping" in assigning work to the starts and finishes of edges.

Initially, we simply scan down the priority list to determine in the obvious way the edges to start at time 0, for a cost of $O(|E|)$. We also create at this time a list for each vertex v of the as-yet-unstarted edges involving v , sorted in priority order (and labelled by their position in the priority list), with the exception that for each neighbor u of v we include in v 's list only the highest priority unstarted edge $\{u, v\}$. Any lower priority copies of $\{u, v\}$ are kept in an auxiliary list, also sorted by priority, which is pointed to by the highest priority copy. This additional pre-processing requires time $O(|E|)$.

To determine the next time at which an edge may start, we maintain a heap containing the finishing times of all the currently executing edges. The minimum value in the heap is the desired time, and we delete this value once we begin scheduling at that time. Since there are $|E|$ values inserted into the heap overall, standard heap implementations will yield a total time of $O(|E|\log|E|)$ for the heap operations.

To schedule edges at a time t when one or more edges have just finished, we proceed as follows: Let V' denote the set of endpoints of the just-completed edges. For each $v \in V'$, scan down the list of unstarted edges involving v to find the highest priority ready edge on that list. Assign the highest priority such edge among all $v \in V'$ to start at t , add the finishing time for that edge to the heap, delete the edge from both edge lists it belongs to (inserting the next higher priority copy of the edge, if any, in those lists), and remove from V' those of its endpoints which no longer have free ports. This list updating requires time $O(|V|)$ and is charged to the newly started edge. We then continue scanning down the unstarted-edge lists for all $v \in V'$, starting from where we left off, once again finding on each such list the highest priority ready edge (possibly the same edge as before suffices, but it may no longer be ready since one of its endpoints may no longer have a free port). Again we assign that one of these with the overall highest priority to start at t , add its completion time to the heap, update V' and the lists for its endpoints accordingly, and repeat the process described above, continuing until we have scanned to the end of each unstarted-edge list without finding a ready edge, at which point no further edges can possibly be scheduled to start at t . We then use the heap to find the next time at which edges can be scheduled, repeating the above process until no unscheduled edge remains.

The list scanning process requires time $O(|V|)$ for each vertex in V' , plus at most two additional individual item scans for each newly started edge (to account for the lower priority copies of that edge which may have been inserted in the lists). The

former is charged to the edges that just completed, for a cost of $O(|V|)$ per edge completion (at least $|V|/2$ edges must have been completed). The latter is charged to the newly started edge, which, with the $O(|V|)$ charge for list updating assigned to that edge, gives a total charge of $O(|V|)$ per newly started edge. Because each edge is started and completed exactly once, this gives a total time of $O(|V| \cdot |E|)$ plus the $O(|E| \log |E|)$ for the heap operations, which combine to give the claimed time bound.

As illustrated in Fig. 8 schedule lengths resulting from different lists can differ substantially. Note that the example is easily generalized to one for which the degrees are $d_0 = k, d_i = k - 1, 1 \leq i \leq k$, and the schedule lengths corresponding to lists $(e_1, \dots, e_k, e_{k+1}, \dots, e_{k^2})$ and $(e_{k+1}, \dots, e_{k^2}, e_1, \dots, e_k)$ are k and $2k - 1$, respectively. This file transfer graph will be given the special designation H_k ; it will come up again later in this section.

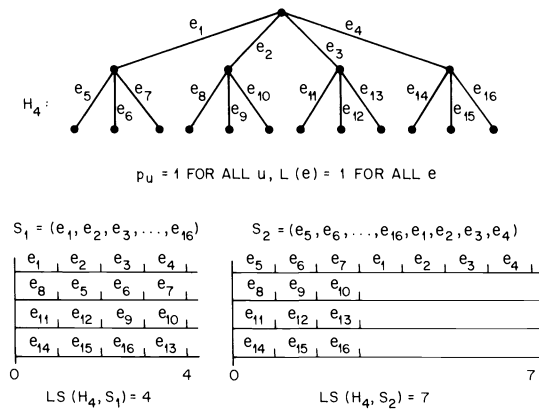


FIG 8. The file transfer graphs H_4 .

Given a file transfer graph G and a sequence S of the edges, denote by $LS(G, S)$ the makespan of the schedule produced when LS is applied to G with list S . It is easily verified that the complexity results of the last section also apply to the problem of finding an ordering of the edges of a file transfer graph that minimizes $LS(G, S)$. However, as the next result shows, even the best list schedule need not always be optimal. Define $OPT_{LS}(G) = \min_S LS(G, S)$, and let $OPT(G)$ be the optimal makespan for G .

THEOREM 11. For any $\delta > 0$ there exists a file transfer graph G such that

$$OPT_{LS}(G) > \left\lfloor \frac{4}{3} - \delta \right\rfloor OPT(G).$$

Proof. Consider the graph in Fig. 9, where edges a_1, a_2 , and a_3 have length $1 + \epsilon, \epsilon \ll 1$, and all other edges have length 1. Each of the three subtrees incident on vertex u can be scheduled with makespan $3 + 3\epsilon$ by scheduling in parallel b_{i1} with e_{i1}, b_{i2} with c_{i2} , and a_i with c_{i1} and e_{i2} in the intervals $[0, 1 + \epsilon], [1 + \epsilon, 2 + 2\epsilon]$, and $[2 + 2\epsilon, 3 + 3\epsilon]$, taken in any order. Adopting an order for the i th subtree that schedules a_i, c_{i1} and e_{i2} in interval $[(i - 1) + (i - 1)\epsilon, i + i\epsilon], i = 1, 2, 3$, therefore assures that the subtrees incident on u can be scheduled in parallel, i.e. a port conflict at u can be avoided. Thus, the graph can be scheduled in time $3 + 3\epsilon$. Note, however, that the schedules above are not list schedules; delays (of at most ϵ) have been introduced at times when there are ready edges.

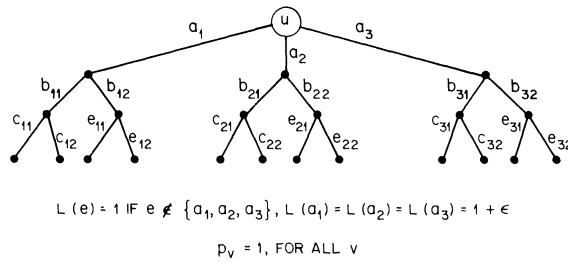


FIG 9. Example for Theorem 11.

Let us now suppose that there is an ordering S for the edges of G such that $LS(G,S) < 4$. Since all edges other than the a_i 's have unit lengths, the earliest of the a_i 's to start, say a_1 , must start at an integer time. If a_1 started at a time $t \geq 1$, then since the a_i 's must be scheduled in disjoint intervals, the length of the schedule would be at least $4 + 3\epsilon$. Thus, a_1 must start at $t = 0$ along with one of c_{11} and c_{12} and one of e_{11} and e_{12} , say c_{11} and e_{11} . At $t = 1$, a_1 is not finished so b_{11} and b_{12} cannot start. Thus, c_{12} and e_{12} must start, and this prevents b_{11} and b_{12} from starting prior to $t = 2$. Since b_{11} and b_{12} must be scheduled in disjoint intervals we have immediately $LS(G,S) \geq 4$. Thus,

$$\frac{OPT_{LS}(G)}{OPT(G)} \geq \frac{4}{3+3\epsilon}$$

and the theorem follows by choosing ϵ sufficiently small. ■

Fortunately, although the best list schedule may not be all that close to optimal, the worst one cannot be all that far away.

THEOREM 12. For any file transfer graph G and any list S of its edges, we have

$$LS(G,S) \leq 2OPT(G) + \max\left\{0, L\left[1 - \frac{2}{p}\right]\right\}$$

where L is the maximum edge length and p is the maximum port capacity. Moreover, the bound is tight, even for trees. To be specific, for $p = 1$, there exist trees G with orderings S such that $LS(G,S)/OPT(G)$ is arbitrarily close to 2 and for $p \geq 2$ there exist trees G with orderings S such that $LS(G,S) = 2OPT(G) + L(1 - (2/p))$, even in the case where L is as large as $OPT(G)$.

Proof. In any given list schedule of G , let $e = \{u, v\}$ denote an edge with latest finishing time, and consider the vicinity of e in G as shown in Fig. 10. At any time prior to the start of e , it must be the case that either all ports at u or all ports at v are busy. But all ports at u can be busy, with edges other than e , for at most $\lfloor (\Sigma_u - L(e))/p_u \rfloor$ units of time, and the symmetric bound holds for v (recall that all edge lengths are integers). Thus, since $LS(G,S)$ equals the starting time for e plus its length $L(e)$, we have

$$\begin{aligned}
 LS(G,S) &\leq \left\lfloor \frac{\Sigma_u - L(e)}{p_u} \right\rfloor + \left\lfloor \frac{\Sigma_v - L(e)}{p_v} \right\rfloor + L(e) & (**) \\
 &\leq \frac{\Sigma_u}{p_u} + \frac{\Sigma_v}{p_v} + L(e) \left[1 - \frac{1}{p_u} - \frac{1}{p_v} \right]
 \end{aligned}$$

The sum of the first two terms above is no more than $2OPT(G)$, since $OPT(G) \geq \Sigma_u/p_u$ and $OPT(G) \geq \Sigma_v/p_v$. The contribution of the last term is bounded as follows. If $(1/p_u) + (1/p_v) \geq 1$, then the last term is non-positive and we have $LS(G,S) \leq 2OPT(G)$. Otherwise, the last term is non-negative, but no more than $L(1 - (2/p))$, since $L(e) \leq L$ and since neither p_v nor p_u can exceed p . Thus the upper bound of the theorem is proved.

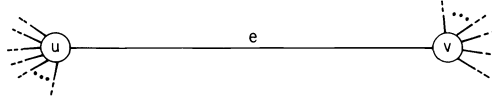


FIG 10. Illustration for Theorem 12.

To see that the upper bound is tight for $p = 1$, consider the trees H_k illustrated in Figure 8 for $k = 4$. As described in the text relating to that figure, $OPT(H_k) = k$ but there are orderings S such that $LS(H_k,S) = 2k - 1 = (2 - 1/k)OPT(H_k)$. Thus as k goes to ∞ , the ratio $LS(H_k,S)/OPT(H_k)$ becomes arbitrarily close to 2, the claimed upper bound.

Let us now turn to the case where $p \geq 2$. For any such p , we can construct a tree and an ordering S of its edges such that $LS(G,S)$ attains the quoted bound. See Figure 11. This graph is built around a single edge $e = \{u,v\}$ of length L , where L will turn out to equal $OPT(G)$. To the endpoint u of e , which has port constraint p , are attached $p(p-1)$ unit-length edges $a(i)$, while to the other endpoint v , also with port constraint p , are attached $p-1$ trees $G(i)$, each $G(i)$ being a copy of the tree H_p , all of whose internal vertices have port constraint 1 and all of whose edges have length 1. It is easy to see that we can order the edges so that a schedule of the following form is produced by LS .

In each of the first $p-1$ time units we schedule one edge from each of the sets $c_{jk}(i)$ ($1 \leq j \leq p, 1 \leq k \leq p-1$) in each of the $G(i)$. Thus a total of $p(p-1)$ of these edges are scheduled in each time unit of $[0, p-1]$. Also starting at $t = 0$, all of the edges $a_i, 1 \leq i \leq p(p-1)$, are scheduled. Since the port constraint at vertex u is p , the a_i 's are scheduled p at a time and thus occupy the first $p-1$ time units. Thus, edge e cannot be scheduled during $[0, p-1]$, and at $t = p-1$ we are left with the tree consisting of e and the $p(p-1)$ edges $b_j(i)$ ($1 \leq i \leq p-1, 1 \leq j \leq p$).

Starting at $t = p-1$ we schedule all of the $b_j(i)$'s; since the port constraint at vertex u is p they are scheduled p at a time in the interval $[p-1, 2(p-1)]$. The start of edge e is clearly delayed until $t = 2(p-1)$. Since $L(e) = p$, we thus have $LS(G,S) = 3p-2$.

An optimal schedule for the graph is easily found. Each of the trees $G(i), 1 \leq i \leq p-1$, is scheduled optimally as shown in Figure 8 for H_k . Since the port constraint at vertex u is p , edge e and each of the $p-1$ $G(i)$'s can be done in parallel in the first p time units. During $[0, p]$ the $p-1$ ports left available by e at vertex v are used to schedule the $p(p-1)$ a_i 's, $p-1$ at a time. This schedule is obviously a best possible schedule, and hence $OPT(G) = p = L$. We thus have

$$LS(G,S) = \left\lfloor \frac{3p-2}{p} \right\rfloor OPT(G) = 2OPT(G) + L \left[1 - \frac{2}{p} \right]$$

with $L = OPT(G)$, as desired. ■

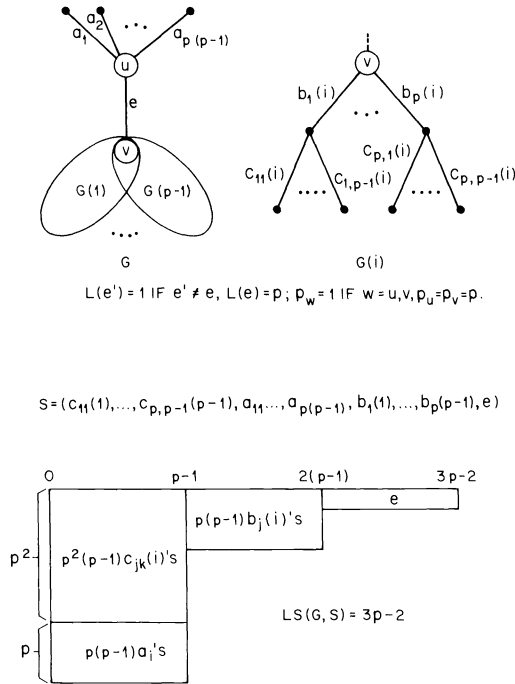


FIG 11. A worst-case example for Theorem 12.

The significance of Theorem 12 is best seen in terms of the following corollaries.

COROLLARY 12.1. For any file transfer graph G and all orderings S of the edges of G ,

$$\frac{LS(G, S)}{OPT(G)} < 3,$$

and this bound is tight in the sense that there are trees G and orderings S such that $LS(G, S)/OPT(G)$ is arbitrarily close to 3.

Proof. Since $OPT(G)$ is always at least as large as the maximum edge length L of G , the upper bound follows from the upper bound of Theorem 12. The lower bound examples are those of Theorem 12 for arbitrarily large values of p . ■

COROLLARY 12.2. If the maximum port capacity p for G is 2 or less, then

$$\frac{LS(G, S)}{OPT(G)} \leq 2$$

and this bound is tight in the sense that there are trees G obeying these port capacity bounds and orderings for which the ratio is arbitrarily close to 2.

Proof. If $p \leq 2$, then $1 - 2/p \leq 0$, so the upper bound once again follows from Theorem 12. The lower bound examples are the trees H_k , as k goes to ∞ . ■

THEOREM 12.3. If all edges in G have the same length, then

$$\frac{LS(G, S)}{OPT(G)} < 2$$

and this bound is tight in the sense that there are trees of equal length edges and orderings for which the ratio is arbitrarily close to 2.

Proof. We may assume without loss of generality that all edge lengths are 1, in which case $\Sigma_v = d_v$, and the argument behind (**) leads to

$$\begin{aligned} LS(G,S) &\leq \left\lfloor \frac{d_u - 1}{p_u} \right\rfloor + \left\lfloor \frac{d_v - 1}{p_v} \right\rfloor + 1 \\ &\leq \left\lfloor \frac{d_u}{p_u} \right\rfloor - 1 + \left\lfloor \frac{d_v}{p_v} \right\rfloor - 1 + 1 \\ &< \left\lfloor \frac{d_u}{p_u} \right\rfloor + \left\lfloor \frac{d_v}{p_v} \right\rfloor \leq 2 \cdot OPT(G) \end{aligned}$$

The lower bound examples are once again the trees H_k as k goes to ∞ . ■

List Scheduling algorithms, such as the one we have been discussing, have the drawback that they are at the mercy of whoever or whatever is ordering the edges. In the above proofs we used this fact in constructing our worst-case examples. The question naturally arises, what advantage might we gain if we had the power to order the edges ourselves? We saw in Theorem 11 that it may be that no re-ordering of the list of edges will yield an optimal schedule. However, a very natural ordering, easy to compute and often used to good advantage in other scheduling problems, does yield an improved bound. The idea is to re-order the edges in “decreasing” order, i.e., so that $L(e_1) \geq L(e_2) \geq \dots \geq L(e_m)$. Because of ties between edges of equal length, there may be more than one possible decreasing order, so we shall evaluate this approach with a conservative, “worst-case” measure. Let

$$DLS(G) \equiv \max\{LS(G,S) : S \text{ is a decreasing ordering of the edges of } G\}.$$

We then have the following result.

THEOREM 13. *For any file transfer graph G with maximum port capacity $p \geq 2$,*

$$\frac{DLS(G)}{OPT(G)} \leq \left(\frac{5}{2} - \frac{1}{p} \right)$$

Moreover, this bound is tight in the sense that for each p there exist trees and orderings for which the ratio approaches the given bound.

Proof. Consider Fig. 10 in Theorem 12, where e is the last edge to finish in our decreasing-list schedule. If $DLS(G) > OPT(G)$, e cannot have started at time 0. Therefore, at time 0 either all the ports of vertex u were busy or else all the ports of vertex v were busy. Assume without loss of generality that the latter is true. Since this is a decreasing-list schedule, this means that v must be an endpoint for at least p_v edges $e' \neq e$ with $L(e') \geq L(e)$. But this means that v is an endpoint of $p_v + 1$ edges of length at least $L(e)$, and hence at least two of them must be scheduled in disjoint intervals, which means that $OPT(G) \geq 2L(e)$. Substituting this inequality into Theorem 12 gives the desired upper bound.

Proving the bound to be tight will be somewhat more difficult than for our previous results (indeed, we initially suspected that an upper bound of 2 might hold). For each $p \geq 2$ we shall define a family $\{G_{p,\epsilon} : \epsilon > 0\}$ of file transfer graphs. The edges of the graphs will not have integral lengths, although this can be rectified by appropriate scaling. Each $G_{p,\epsilon}$ will be described as a rooted tree for which we can show

$$DLS(G_{p,\epsilon}) \geq OPT(G_{p,\epsilon}) \left(\frac{5}{2} - \frac{1}{p} \right) - f(\epsilon),$$

where $f(\epsilon)$ approaches 0 as ϵ does. The parameter ϵ does not affect the structure of the tree, only the lengths of its edges. $G_{2,\epsilon}$ is illustrated in Fig. 12.

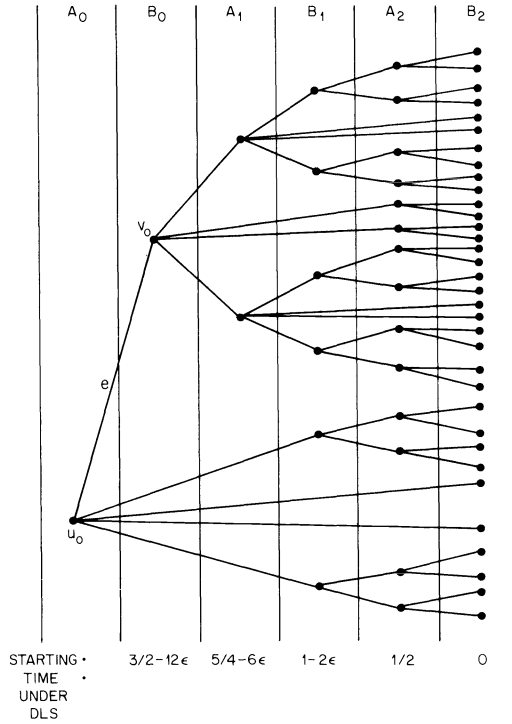


FIG 12. File transfer graph $G_{2,\epsilon}$ for Theorem 13.

For convenience in the definition of $G_{p,\epsilon}$, the vertices of the tree will be organized into 2 sequences of sets A_0, A_1, \dots, A_p and B_0, B_1, \dots, B_p , with the root of $G_{p,\epsilon}$ in $A_0 = \{u_0\}$. B_0 will also have only a single vertex, and it will be denoted v_0 . The numbers of vertices in the remaining sets A_i and B_i will increase with the index, i .

The structure of $G_{p,\epsilon}$ will be defined by successively constructing all edges, starting with those incident on the root. The numbers of vertices in the sets A_i and B_i will be determined implicitly from this construction.

We start by placing an edge between u_0 and v_0 . This edge corresponds to e , the last edge to finish under both schedules and the only edge to be named explicitly. Next, edges are placed between u_0 and p distinct vertices in each of B_1, B_2, \dots, B_p . Then, from v_0 edges are formed to p distinct vertices in each of A_1, A_2, \dots, A_p .

Proceeding now to A_1 , we place p edges from each vertex in A_1 to p (new) vertices in each of B_1, B_2, \dots, B_p . Next, we place p edges from each vertex in B_1 to p (new) vertices in each of A_2, A_3, \dots, A_p . This continues for $i = 1, 2, \dots, p-1$, so that p edges from each vertex in A_i go to p new vertices in each of B_i, \dots, B_p , and immediately after this assignment, p edges from each vertex in B_i are joined to p new vertices in each of A_{i+1}, \dots, A_p , $1 \leq i \leq p-1$. After assigning the edges from vertices in B_{p-1} to vertices in A_p , the process of constructing the edges of $G_{p,\epsilon}$ concludes with the placement of p edges from each vertex in A_p to p new vertices in B_p .

Although we are assuming that all port constraints are p , note that the port constraint at vertices in B_p cannot be important, since they each have but one incident edge.

It can be verified that just the number of edges from vertices in A_p to vertices in B_p grows faster than p^{p+2} . This not only explains why a graph more illustrative than G_2 was not drawn, but it suggests that a complete indexing of the edges be avoided if possible. For this latter problem we shall classify an edge by the higher-indexed set containing one of its endpoints. Specifically, to economize on notation, A_i ($1 \leq i \leq p$) will also be used to represent all those edges with one endpoint in A_i and one endpoint in some B_j with $j < i$. Thus, the vertices of A_i are in one-to-one correspondence with the edges in A_i . In terms of Fig. 12, A_i is simply the set of edges from vertices to the left of A_i to vertices in A_i . Similarly, we also use B_i , $0 \leq i \leq p$, to denote the set of edges between vertices in B_i and vertices in sets A_j for $j \leq i$.

We now wish to assign edge lengths that ensure the following properties:

(i) In the *DLS* schedule the edges are scheduled set by set in the sequence $B_p, A_p, B_{p-1}, A_{p-1}, \dots, B_1, A_1, B_0 = \{e\}$, with the edges within a set all having approximately the same length, and these lengths being chosen so that the makespan is approximately $5/2 - 1/p$.

(ii) $L(e)$ is within ϵ of being the maximum edge length in $G_{p,\epsilon}$, even though by (i) the edge e cannot be ready to be scheduled until the edges in A_1 are finished.

(iii) An optimal schedule can be found with makespan approximately 1.

To show that the scheduling sequence in (i) is valid, we must verify that it is consistent with the structure of $G_{p,\epsilon}$ as well as its edge lengths. With respect to the structure question, we note that if the edges of B_p are scheduled at $t = 0$, no other edges in $G_{p,\epsilon}$ can be ready. This follows from the fact that p edges from every vertex in A_0, A_1, \dots, A_p terminate in B_p . Thus the port capacities of all vertices in these sets are "used up" by the edges of B_p . Since all remaining edges have an endpoint in one of the vertex sets A_j , $1 \leq j \leq p$, no other edges can be ready.

Now suppose the edges in B_p have finished and those in A_p have started. As before, p edges at each vertex in B_j , $0 \leq j \leq p-1$, terminate in A_p ; therefore the port capacities of all these vertices are used up, and since all remaining edges have an endpoint in a set B_j for some j , $0 \leq j \leq p-1$, none can be ready. Inductively, this holds for the scheduling of edges in each A_i and B_i , $1 \leq i \leq p$, in the sequence given in (i); i.e. no other edges are ready during their execution, assuming that we have arranged edge lengths so that only the edges in A_j (respectively B_j) are in execution after the edges in B_j (respectively A_{j+1}) have finished.

As shown in Table 3 the lengths of edges in B_p, A_p and $B_0 = \{e\}$ will all be about $1/2$, and all remaining edges will have a length approximately $1/(2p)$. With the sequence in (i) this will give (roughly)

$$DLS(G_{p,\epsilon}) \approx \frac{1}{2} + \frac{1}{2} + 2(p-1)\frac{1}{2p} + \frac{1}{2} = \frac{5}{2} - \frac{1}{p}$$

Later, we shall show that $OPT(G_{p,\epsilon}) \approx 1$.

Excluding edge e in B_0 , the lengths of edges in all other sets are in a decreasing sequence from set to set, so that the edge sequence $B_p, A_p, \dots, B_1, A_1$ will conform to a *DLS* schedule. As shown in Table 3, the decreasing sequence is achieved by successive subtraction from $1/2$ and $1/(2p)$ of an increasing $O(\epsilon)$ term.

To keep edge e from being ready until all others are finished, we must properly fix the lengths of the edges between its endpoints u_0 and v_0 and the vertices of A_i and B_i ,

TABLE 3
 Message lengths for Fig. 12. A_i^0 and B_i^0 contain just those edges of A_i and B_i involving v_0 and u_0 , respectively.

| Messages in | Lengths | Subscript Range |
|-------------------|---------------------------|---------------------|
| $B_0 = \{e\}$ | $1/2$ | |
| B_p^0 | $1/2 + \epsilon$ | |
| $B_p - B_p^0$ | $1/2$ | |
| A_p^0 | $1/2 - \epsilon$ | |
| $A_p - A_p^0$ | $1/2 - 2\epsilon$ | |
| B_{p-i}^0 | $1/(2p) - (4i-1)\epsilon$ | $1 \leq i \leq p-1$ |
| $B_{p-i} - B_i^0$ | $1/(2p) - (4i)\epsilon$ | $1 \leq i \leq p-1$ |
| A_{p-i}^0 | $1/(2p) - (4i+1)\epsilon$ | $1 \leq i \leq p-1$ |
| $A_{p-i} - A_i^0$ | $1/(2p) - (4i+2)\epsilon$ | $1 \leq i \leq p-1$ |

$i \geq 1$. Denoting the sets of these edges by $A_i^0 \subseteq A_i$ and $B_i^0 \subseteq B_i$, we give each a length ϵ greater than that of the other edges in the sets A_i and B_i , respectively (see Table 3). To see that this works, consider first the scheduling of B_p edges, which must begin at $t = 0$ because of the file lengths of $(1/2) + \epsilon$ between u_0 and vertices in B_p . (The reader may wish to refer to Fig. 12 in this discussion.) At time $t = 1/2$, all of the B_p edges except the p to u_0 are finished; these latter ones have ϵ to go, and therefore u_0 is still not available for scheduling e . Therefore, the smaller edges of A_p begin at time $t = 1/2$, thus occupying vertex v_0 and preventing edge e from being scheduled during $[1/2, 1-\epsilon]$.

But at time $1-2\epsilon$ all edges of A_p except the p edges involving vertex v_0 are finished, and hence at $1-2\epsilon$ the edges in B_{p-1} begin, and they occupy vertex u_0 during $[1-2\epsilon, 1 + 1/(2p) - 5\epsilon]$, while all the other edges in B_{p-1} are finished by time $1 + 1/(2p) - 6\epsilon$. It should now be clear how this alternating process of occupying vertices u_0 and v_0 is made possible by the ϵ increment in length for edges between u_0 and vertices in B_i , and the similar increment for edges between v_0 and vertices in A_i , $i \geq 1$. In general, the edges in B_{p-i} , $1 \leq i \leq p-1$, finish at time $1 + (2i-1)/2p - b_i\epsilon$ (with the edges to u_0 finishing ϵ later), while the edges in A_{p-i} finish at time $1 + 2i/2p - a_i\epsilon$ (with those to v_0 finishing ϵ later). The values of a_i and b_i are determined by the recurrence

$$\begin{aligned}
 b_1 &= 6, \\
 a_i &= b_i + 4i + 2, \quad 1 \leq i \leq p-1, \\
 b_i &= a_{i-1} + 4i, \quad 2 \leq i < p-1.
 \end{aligned}$$

Solving, we obtain

$$a_{p-1} = 6 + \sum_{j=3}^{2p-1} 2j = 4p^2 - 2p,$$

which means that the edges from v_0 to A_1 do not finish until time

$$1 + \frac{2(p-1)}{2p} - (4p^2 - 2p)\epsilon + \epsilon.$$

Since u_0 and v_0 are constantly occupied until all edges in A_i and B_i , $1 \leq i \leq p$, are

finished, e will not be scheduled until this time, and hence

$$DLS(G_{p,\epsilon}) = \frac{5}{2} - \frac{1}{p} - (4p^2 - 2p - 1)\epsilon. \tag{***}$$

We turn now to optimal schedules for $G_{p,\epsilon}$. We shall not actually construct an optimal schedule, but instead will show that a schedule exists with makespan of $1 + \epsilon$, an upper bound on $OPT(G_{p,\epsilon})$ that will be good enough for our purposes. To work out such a schedule we shall determine schedules individually for the set of edges at each different vertex type in the order $A_0, B_0, A_1, B_1, \dots$. From the tree structure of $G_{p,\epsilon}$ we know that, when scheduling edges at a vertex v ($v \neq u_0$) in this sequence, exactly one of its edges, say $\{u, v\}$, has already been scheduled (there is only one such edge to an earlier vertex). Therefore, in defining the schedule for the as yet unscheduled edges at v , we will show how to make a port available at v during the time interval that $\{u, v\}$ has already been scheduled.

1. The root, u_0 . This vertex has p edges of length $1/2 + \epsilon$ with endpoints in B_p , p edges of length at most $1/(2p)$ with endpoints in each of B_1, B_2, \dots, B_{p-1} , and the edge e of length $1/2$. The total edge length is approximately

$$\frac{(p+1)}{2} + \frac{p(p-1)}{2p} = p,$$

which at a p -port vertex can be accommodated in about one unit of time. Specifically, we schedule the p edges with endpoints in B_p on p ports during $[0, 1/2 + \epsilon]$. Then we schedule edge e on one port during $[1/2 + \epsilon, 1 + \epsilon]$, and the remaining $p(p-1)$ edges of length at most $1/(2p)$ on the other $p-1$ ports during $[1/2 + \epsilon, 1 + \epsilon]$.

2. Vertex v_0 . This vertex is similar to u_0 . It has p edges of length $1/2 - \epsilon$ with endpoints in A_p , p edges of length at most $1/(2p)$ with endpoints in each of A_1, \dots, A_{p-1} , and finally it has edge e . We schedule the p edges of length $1/2 - \epsilon$ during $[0, 1/2 - \epsilon]$. The remaining $p(p-1)$ unscheduled edges of length at most $1/(2p)$ are scheduled on $p-1$ ports during $[1/2 - \epsilon, 1 + \epsilon]$, leaving one port free during $[1/2 + \epsilon, 1 + \epsilon]$ for edge e , which has already been scheduled at u_0 during this interval. See Fig. 13.

3. A vertex $v \in A_i, 1 \leq i \leq p-1$. This vertex has p edges of length $1/2$ with endpoints in B_p and p edges of length at most $1/(2p)$ with endpoints in each of B_i, \dots, B_{p-1} . It also has an edge of length at most $1/(2p)$ which has already been scheduled during $[1/2, 1]$ at some vertex, say u , in B_j , for some $j < i$. As before we schedule the p edges of length $1/2$ in $[0, 1/2]$. Then, the at most $p(p-i) \leq p(p-1)$ remaining unscheduled edges of length at most $1/(2p)$ are scheduled in $[1/2, 1]$ on $p-1$ ports, leaving one port idle during $[1/2, 1]$ to accommodate the edge that u has already scheduled with v . Note that there will be slack time in schedules for vertices in $A_i, i \geq 1$, whereas the schedules for u_0 and v_0 were essentially packed full. Moreover, this slack time increases with i .

4. A vertex $v \in B_i, 1 \leq i \leq p-1$. A vertex in B_i is similar to one in $A_i, 1 \leq i \leq p-1$, except that corresponding edge lengths are slightly greater at $v \in B_i$. Thus, edges at the B_i vertices for $1 \leq i \leq p-1$, can be scheduled in the same manner as those in A_i .

5. Vertices in A_p . These vertices have $p+1$ edges of length at most $1/2$, one of which has already been scheduled during $[0, 1/2 - \epsilon]$ in step 2 or step 4 above. We simply schedule the p edges with endpoints in B_p during $[1/2, 1]$.

Vertices in B_p do not have to be considered, for their edges have all been scheduled in steps 1, 3, and 5.

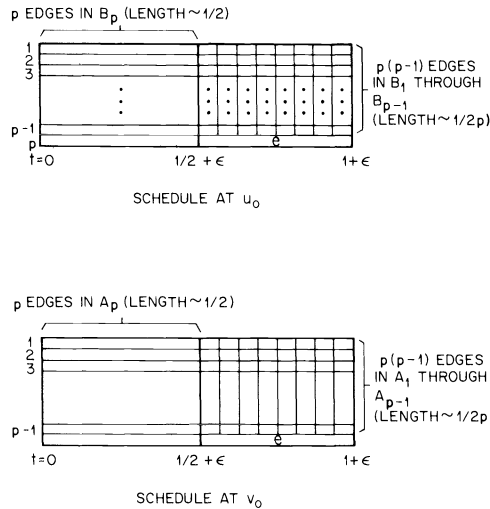


FIG 13. Optimal schedules at u_0 and v_0 in Theorem 13.

Finally, it is clear that the combination in $[0, 1+\epsilon]$ of all of the above schedules constitutes a schedule with makespan $1+\epsilon$. From (***) we can thus conclude that

$$\lim_{\epsilon \rightarrow 0} \frac{DLS(G_{p,\epsilon})}{OPT(G_{p,\epsilon})} \geq \frac{5}{2} - \frac{1}{p},$$

thus proving that the upper bound of Theorem 13 is tight in the sense claimed. ■

We have been unable to find any general polynomial time approximation algorithm that provides an improvement on the $2.5OPT(G)$ bound for DLS. However, we have been able to provide better guarantees in special cases. For instance, if all edge lengths are equal and there are no multi-edges, one can apply Vizing's result [1,14] that any graph can be edge-colored using only one more color than the obvious lower bound of $\max_{v \in V} d_v$. The proof of this result yields a polynomial time algorithm for generating such a coloring, which in turn yields a scheduling algorithm that is guaranteed to come within an *additive* constant of $OPT(G)$ in the case where all port capacities are 1. If multi-edges are allowed, we can still improve on DLS in the single port case, although the additive constant now is multiplied by the maximum edge multiplicity. For high edge multiplicities, this can be replaced by an algorithm guaranteeing $\lceil (3/2)OPT(G) \rceil$ that is based on an edge-coloring result of Shannon [1,13]. Recent edge-coloring results of Goldberg [9,10] suggest that a more complicated but still polynomial algorithm will provide a guarantee of $(9/8)OPT(G) + 1$. (Recall that DLS only guarantees $2OPT(G)$ for such instances.)

A second class of instances for which improvements over List Scheduling are possible are those in which the file transfer graph is a tree. Recall that the lower bounds for both *LS* and *DLS* hold even if the file transfer graph is a tree. We shall show how

to beat the $(2.5)OPT(G)$ bound of DLS by tailoring our algorithms specifically to trees. We show how any heuristic for the multiprocessor scheduling problem can be adapted to the tree problem in such a way that its performance guarantee (in the multiprocessor scheduling problem) also adapts to the tree problem. Recall from the discussion preceding Theorem 2 that file transfer scheduling and multiprocessor scheduling are identical for 2-vertex graphs. The association of ports with processors established there will continue to apply in the sequel.

For the following theorems we need some notation. Let $A(E,p)$ denote the length of the multiprocessor schedule on p processors (ports) produced by algorithm A for a set E of tasks (edges). We let $OPT(E,p)$ denote the length of a corresponding optimal schedule.

THEOREM 14. *For a fixed integer $p > 0$, suppose A is a polynomial-time algorithm for multiprocessor scheduling such that for any set E of tasks, there is a constant B_p such that*

$$A(E,p) \leq B_p \cdot OPT(E,p).$$

whenever $p' \leq p$. Then there is a polynomial-time algorithm A' for file transfer scheduling that, given a tree G without multi-edges and with maximum port capacity p , guarantees

$$A'(G) \leq (B_p + 1)OPT(G).$$

Proof. Using algorithm A , we first construct for each vertex v of G a separate schedule for just those edges incident on v , using p_v ports. Thus each edge will be scheduled in exactly two of these schedules, one for each of its endpoints. By our assumption on the bound for A , the schedule for v will satisfy $A(E_v, p_v) \leq B_p \cdot OPT(E_v, p_v)$. Then we proceed to combine these schedules, adding them one at a time to the partial overall schedule constructed so far, and each time adjusting the one being added so that it is consistent with the overall schedule. The key is to show that the adjustments can always be performed in such a way as to obey the stated bound.

So suppose we have computed the individual schedule for all vertices of G using algorithm A . Choose an arbitrary vertex v_0 , and any ordering v_0, v_1, \dots, v_n of the vertices of G with the property that, for $1 \leq i \leq n$, exactly one neighbor of v_i precedes v_i in the ordering. The existence of such an ordering follows easily from the fact that G is a tree. We will combine the individual schedules for the vertices according to this ordering.

We begin with the schedule for vertex v_0 as our partial overall schedule. It clearly satisfies the desired bound, since $B_p \cdot OPT(E_{v_0}, p_{v_0}) \leq B_p \cdot OPT(G)$.

In general, suppose we have combined the schedules for vertices v_0, v_1, \dots, v_i , $0 \leq i < n$, into a valid schedule for all edges incident on any of those vertices and that this partial schedule satisfies our desired bound. We show how to "merge" the individual schedule for $v = v_{i+1}$ into this schedule to obtain a partial overall schedule for v_0, v_1, \dots, v_{i+1} that satisfies the bound.

Let u be the single neighbor of v that occurs earlier in our ordering, and observe that the only edge incident on v that has already been scheduled in the partial overall schedule is the single edge $\{u, v\} = e$. Thus all we need to do is to suitably modify the schedule for v so that e is scheduled at the same time as in the partial overall schedule, and we can then simply combine the two schedules. We do this by modifying only that portion of v 's schedule that involves the single port of v to which e has

been assigned. Because v 's schedule ignores neighbor constraints at this point, we may assume, without loss of generality, that the schedule for v executes tasks consecutively, with no inserted idle time, on this port.

Let t be the time at which e is scheduled to start in the partial overall schedule, and let t' be the time at which e is scheduled to start in the schedule for v . If $t' \leq t$, change the schedule for v by starting e at t , decreasing by $L(e)$ the starting time for each edge that was started after e (on e 's port) and was completed by $t + L(e)$, and increasing by $t + L(e) - s^*$ the starting time for each edge that was completed after $t + L(e)$ (on e 's port), where s^* is the starting time of the first edge e' that was completed on the port after $t + L(e)$ (and hence $t + L(e) - s^* < L(e')$). See Figure 14a. We then combine the resulting schedule for v with the given partial overall schedule to obtain our expanded partial overall schedule. To see that the desired bound continues to be satisfied, observe that if e is now the last edge to be started on e 's port, then the makespan for the new overall schedule either is the same as it was (and hence satisfies the bound by assumption) or is achieved on one of the ports of v for which v 's schedule was left unchanged, and hence is at most $A(E_v, p_v) \leq B_p \cdot OPT(G)$. On the other hand, if e is not now the last edge started on its port, the makespan for v 's schedule was increased by at most $L(e') \leq OPT(G)$, so the new schedule for v has makespan at most $A(E_v, p_v) + OPT(G) \leq (B_p + 1) \cdot OPT(G)$. Thus, since the old partial overall schedule satisfied this bound, the new partial overall schedule must continue to do so.

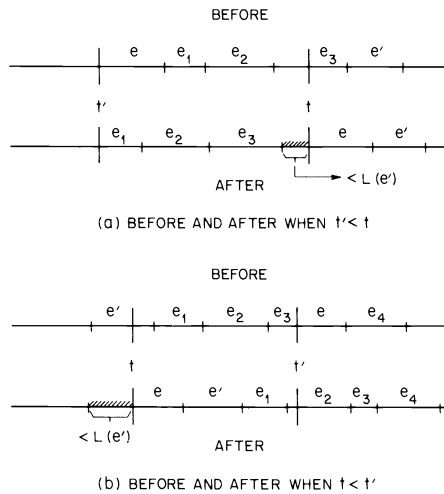


FIG 14. Illustrations for Theorem 14.

If $t < t'$, change the schedule for v (on e 's port only) by starting e at time t , increasing by $L(e) + t - s(e)$ the starting time for each edge completed after t and at or before t' , where e' denotes the first edge completed after t , and increasing by $t - s(e') < L(e')$ the starting time for each edge completed after t' . See Figure 14b. This increases the makespan of the schedule for v by at most $L(e') \leq OPT(G)$. Thus, as above, the makespan for the new schedule for v will be at most $(B_p + 1) \cdot OPT(G)$.

Combining this new schedule for v with the given partial overall schedule thus again gives us a new partial overall schedule satisfying the desired bound.

The theorem follows by induction. ■

Note that if in Theorem 14 we have $p_v = 1$ for all v in the tree G , then there is an A' such that $A'(G) \leq 2OPT(G)$, since it is trivial to design a multiprocessor scheduling algorithm that finds optimal schedules when there is just one processor, and hence has $B_1 = 1$. Note, however, that List Scheduling yields the same guarantee for *arbitrary* graphs with unit port capacity. We do gain improvements over LS when port capacities exceed 1:

COROLLARY 14.1. *For any $p > 1$ and any $\epsilon > 0$, there exists a polynomial time algorithm A' such that*

$$A'(G) \leq (2 + \epsilon)OPT(G)$$

for all single-edge trees with port capacities bounded by p .

Proof. In [12] it is shown that for any fixed number m of processors and any $\epsilon > 0$, there is a polynomial time multiprocessor scheduling algorithm that guarantees a solution with makespan at most $1 + \epsilon$ times optimal. Theorem 14 thus yields the desired result, although unfortunately the algorithm's running time is exponential in m (and hence in p). ■

COROLLARY 14.2. *There exists an $O(|E| \log |E| \log(\sum_{e \in E} L(e)))$ algorithm A' such that*

$$A'(G) \leq (2.2)OPT(G)$$

for all single-edge trees.

Proof. The MULTIFIT DECREASING algorithm of [2] for multiprocessor scheduling runs in the above time (when $|E|$ is the number of tasks) and has been shown in [5] to guarantee a makespan at most $6/5$ times optimal, no matter what the number of processors. Theorem 14 thus applies. Moreover, it is straightforward to see that the use of a multiprocessor scheduling algorithm in the manner described in the proof of Theorem 14 for a file transfer graph with E edges yields the same time complexity as running the algorithm in its normal mode for a multiprocessor scheduling instance with $|E|$ tasks. ■

Theorem 14 thus provides a nontrivial improvement over *DLS*'s guarantee of $(2.5)OPT(G)$, although it is restricted to single-edge trees. We can also improve upon *DLS*'s guarantee for multi-edge trees, but this requires that we restrict ourselves to the case in which all port capacities are the same.

THEOREM 15. *Suppose A is a polynomial time multiprocessor scheduling algorithm as in the hypothesis of Theorem 14. Then there is a polynomial time algorithm A' for file transfer scheduling that, given a tree G (possibly with multi-edges), all of whose port capacities equal p , guarantees*

$$A'(G) \leq (2B_p)OPT(G).$$

Proof. Choose an arbitrary vertex v_0 as the root of the tree, "color" it red and "color" each remaining vertex either red or blue in such a way that no two adjacent vertices have the same color. Each edge then joins two differently colored vertices, one being the parent and the other being the child in the rooted tree; color the edge with the same color as the parent vertex.

Now we form two separate schedules, the "red" schedule for the red edges and the "blue" schedule for the blue edges. For the red schedule, we just use

algorithm A to construct for each red vertex a p -processor (p -port) schedule for the red edges incident on that vertex and take the union of these individual schedules. The port constraint at each red vertex is obeyed in this schedule by construction (and by the fact that no two red vertices are adjacent). The port constraint at each blue vertex is also obeyed in this schedule, because all red edges incident on a blue vertex go to the same other vertex, its parent in the rooted tree, and the blue vertex has the same port constraint as its red parent. The makespan for the red schedule is the maximum of the makespans of the individual schedules, which is at most $\max_{v \in V} \{A(E_v, p)\} \leq B_p \cdot OPT(G)$, where the maximum can be taken over just the red vertices. The blue schedule is constructed in the same way, being the union of individual schedules for the blue edges incident on each blue vertex, and similarly has makespan at most $B_p \cdot OPT(G)$.

To construct the required overall schedule, we simply concatenate the red and blue schedules constructed above, starting the blue schedule at the time that the last edge of the red schedule is completed. Since each of the red and blue schedules has makespan at most $B_p \cdot OPT(G)$, the concatenated schedule has makespan at most $2B_p \cdot OPT(G)$, as claimed. ■

As with Theorem 14, we have two main corollaries, based on the same multiprocessor scheduling algorithms.

COROLLARY 15.1. *For any $p > 1$ and any $\epsilon > 0$, there exists a polynomial time algorithm A' such that*

$$A'(G) \leq (2 + \epsilon)OPT(G)$$

for all (multi-edged) trees with all port capacities equal to p .

COROLLARY 15.2. *There exists an $O(|E| \log |E| \log(\sum_{e \in E} L(e)))$ algorithm A' such that*

$$A'(G) \leq (2.4)OPT(G)$$

for all (multi-edged) trees with all port capacities equal.

4. Distributed scheduling. The scheduling algorithms of the previous section have all assumed the existence of some central processor that knows in advance all the files to be transferred and their lengths. An even more basic assumption is that once a schedule has been constructed it can be carried out without a hitch, i.e., the various nodes will be able to commence sending and receiving files at the predetermined times and all transfers will take their predicted times. In many of the envisioned applications of our problem, neither of these assumptions will apply. No central processor will exist, transfer times may not be known exactly in advance, and indeed the set of files to be transferred may not be entirely known when the first transfer takes place. The question thus arises: How much do our results for the idealized file transfer scheduling problem have to say about the actual transfer of files in distributed systems?

The NP-hardness results continue to tell us something. If finding optimal makespan schedules is difficult when one processor knows everything in advance, it certainly does not become any easier when the problem must be solved in a distributed fashion while subject to imperfect information. Surprisingly, however, there is also some carry-over of our algorithmic results, in particular, those concerning the List Scheduling approximation algorithm. The algorithm itself, requiring a central processor, does not apply. However, the proof of its guarantee can be adapted to yield a similar result for a kind of schedule that *can* be generated by a distributed process.

(Note that any process for transferring files, even one that does not involve pre-assigned starting times, can be viewed as defining a schedule *ex post facto*; one need only observe when transfers actually began and how long they took to be able to construct both the schedule and the file transfer graph.)

Let us call a schedule s for file transfer graph G a *demand schedule* if there is no interval of time $[t, t']$, $0 \leq t < t'$, such that for some vertices u and v with an edge e between them, e does not start before t' even though both u and v have an idle port during the interval $[t, t']$. If it is not a demand schedule, define for each pair u, v of distinct vertices in G the quantity $delay_s(u, v)$ to be the total length of intervals during which the above is violated for u and v . A schedule s will be called a Δ -*delayed demand schedule*, $\Delta \geq 0$, if Δ equals the maximum value of $delay_s(u, v)$ for all pairs u, v . Note that a 0-delayed demand schedule is simply a demand schedule. A careful examination of the proof of Theorem 12 shows that it implies the following:

THEOREM 16. *If s is a Δ -delayed demand schedule for G , then*

$$MAKESPAN(s) \leq 2OPT(G) + L(e) \left[1 - \frac{1}{2p} \right] + \Delta \leq 3OPT(G) + \Delta$$

where e is the last edge to finish and p is the maximum port capacity.

Now every schedule is a Δ -delayed demand schedule for some Δ ; what we need are schedules with small Δ 's. Fortunately, distributed scheduling algorithms are fairly easy to devise that construct Δ -delayed demand schedules for Δ 's that one can reasonably expect to be small in relation to the overall makespan. Consider the application in which ports correspond to telephone lines, as in a network of home computers. The only way to communicate with another node is to call it up, using the same phone lines over which the files themselves are to be transferred. If a vertex has more than one port, we shall assume that an automatic call routing mechanism sends an incoming call to a free port if one exists. The following distributed algorithm suggests itself. It assumes that file transfers, once initiated, occur in parallel with the operation of the scheduling protocols, terminating when the transfer is finished by disconnecting the phone link and freeing the port used for the transfer.

Each vertex v has a set E_v of the edges (files) it wants to send or receive. (We do not require that both endpoints of an edge have that edge in their sets, only that one of them does.) Each vertex v maintains a queue Q_v of those files it currently wants to send or receive. Each vertex v then executes the following protocol:

DEMAND PROTOCOL 1 (DP1).

Repeat until Q_v is empty:

1. If v has an idle port, attempt to call the other endpoint u of the first edge e in Q_v .
 - 1.1. If u answers, initiate the transfer of e and delete e from Q_v .
 - 1.2. If busy or no answer, move e to the end of the queue.
2. If v has an idle port, wait some prespecified time ϵ for an incoming call.
 - 2.1. If a call is received from a neighbor u , initiate the requested transfer, delete the corresponding edge e from Q_v (if present), and end Step 2.
 - 2.2. If no call is received after waiting for ϵ , end Step 2.

END (DEMAND PROTOCOL 1)

The main problem faced here does not concern file transfer scheduling but rather the difficulties of communication inherent in this distributed "dial-up" model. How does one locate a neighbor with a free port when there is always the possibility of

“false” busy signals, i.e., busy signals caused because the node being called is itself making a call that will eventually receive a busy signal or no answer? By appropriately adjusting the length of the wait in Step 2 (with perhaps different values for different vertices), one can presumably minimize the amount of time one expects to spend in such churning. (Step 1.2, by moving an edge that got a busy signal to the end of the queue, prevents us from endlessly calling a truly busy neighbor when idle neighbors still exist.) More complicated schemes for determining when one calls and when one waits might yield further improvements. However, since the avoidance of busy signals is not the main subject of the current paper, we shall merely attempt to isolate this “communication delay” from the delays caused by the scheduling heuristic.

Given a protocol P and a distributed network represented by a file transfer graph G , let $\delta(P, G)$ be the maximum time for the network, using P at each node, to initiate some transfer, given that there is an untransmitted edge whose endpoints both have free ports. Although no such upper bound may exist that holds over *all* executions of the protocol, such a bound can at least be computed *ex post facto* for any particular scheduling of G using P (assuming there is no infinite sequence of false busy signals).

The situations of interest are, of course, those in which $\delta(P, G)$ is small relative to $OPT(G)$. In this case, the following corollary of Theorem 16, which follows from that result and the above definitions, becomes relevant.

COROLLARY 16.1. *If s is a schedule constructed by Demand Protocol 1 with *ex post facto* file transfer graph G , maximum port capacity p , and last-finishing edge e , then*

$$\begin{aligned} \text{MAKESPAN}(s) &\leq 2OPT(G) + L(e) \left[1 - \frac{1}{2p} \right] + \delta(DP1, G) \cdot |E| \\ &\leq 3OPT(G) + \delta(DP1, G) \cdot |E|. \end{aligned}$$

Thus if communication delays can be kept small relative to file transfer time, the guarantees of the previous section can be carried over to the distributed milieu. Moreover, given any specified scheme for avoiding unnecessary busy signals, there is a sense in which DP1 can be expected to minimize communication delay, since it requires only one completed call per pair involved in a file transfer. The number of completed calls required could be further reduced by transferring *all* files that are to go between u and v as soon as the two vertices are connected. However, this latter approach may have its own drawbacks. It would prevent the parallel transfer of multiple files between the same pair of vertices, and hence could give rise to a Δ -delayed demand schedule where Δ exceeds $\delta(P, G) \cdot |E|$, at least in those cases where port capacities exceed 1.

Note that protocol DP1 does not require that the lengths of the files be precisely known in advance. The protocol can also be easily adapted to the case where the required file transfers are not all known at the start of the transfer procedure, but arrive on-line, thus being added to the lists Q_v in midstream and causing the protocol to be restarted if Q_v was already empty at the time. (An explicit mechanism for this is illustrated in the next protocol we present.) The value of the Theorem 16 guarantee is considerably lessened, however, since now delays can occur because of the late arrival of a transfer request, not just because of communication delays. However, one can still get a guarantee:

COROLLARY 16.2. *If s is a schedule constructed by Demand Protocol 1 (modified to handle late arrivals), G is the *ex post facto* file transfer graph, and if one*

takes $OPT(G)$ to be the minimum makespan possible, given the late arrival of certain requests, then

$$MAKESPAN(s) < 4OPT(G) + \delta(DP1, G) \cdot |E|.$$

Proof. Simply partition the schedule into the part *before* the arrival of the last transfer request, which clearly has length less than $OPT(G)$, and the part starting when the last file transfer request arrives, from which point Theorem 16 applies with Δ equal to the remaining communication delay. ■

The protocol can also be adapted to tolerate unreliable vertices, in a manner similar to that in the next protocol, although there is some question as to how appropriate makespan is as an optimization criterion in either the on-line situation or the case of unreliable processes — see Section 5. (As it stands, DP1 can handle vertices that die and stay dead, in which case it still insures that all transfers not involving the dead vertices are eventually completed, assuming that there is no infinite sequence of false busy signals and that the death of a neighbor frees all ports it was using at the time.)

If the system is such that communication delays are very small in comparison to file transfer time, one might be willing to spend more time in protocol communication in exchange for an improved bound such as the $(2.5)OPT(G)$ of Decreasing List Scheduling. In the remainder of this section we describe a protocol that does just this. As with the previous protocol, this one does not depend on precise knowledge of the edge lengths in the file transfer graph. (However, the 2.5 result will require that both endpoints of an edge know of its existence at time 0 and have identical estimates of its length. The extent to which an estimate can be “off” will determine the quality of the guarantee.) The protocol we describe can also handle on-line requests and tolerate unreliable vertices, although these may degrade its performance and the guarantees we prove will depend on the system behaving sensibly and reliably.

To focus on scheduling issues rather than the problems inherent in distributed communication, we shall push issues of communication and reliability even further into the background than we did for DP1, by postulating the existence of certain lower level protocols that behave as follows:

(1) There is a *Communications Protocol* that any vertex can invoke to send a query to a neighbor, and that will return an appropriate response (one of those specified by our scheduling protocol) or a “busy” signal, the latter only occurring if the neighbor is currently using all its ports for file transfers (or is either dead or so slow in responding that it *appears* to be dead).

(2) There are file transfer protocols *FileSend* and *FileReceive* that, given an agreement between two vertices to transfer a file, can be invoked by the sender and receiver respectively to effect the transfer. (The order of invocation is irrelevant; they take care of their own synchronization.) On completion of the transfer (or premature termination), each protocol informs its invoker of the transfer’s success or failure.

We also assume that queries, responses, and reports on successful or unsuccessful file transfers are all placed on a First-In, First-Out *Protocol Message Queue* as they arrive, along with any new internally-generated requests for transfers.

Our old protocol could be reinterpreted in this model, with the one type of query being “Will you agree to perform a transfer?” and the allowed answers being “No” (when all my ports are full or I myself have an outstanding request) or “Yes” (in which case the requester and the responder both initiate the appropriate file transfer protocols). However, note that such an implementation wastes the power of our assumption that all queries receive answers, by making one of the answers equivalent

to receiving a busy signal in our original model. The power of that assumption, as we shall see, is that it allows us to prove that there cannot be an infinite sequence of operations without progress being made, as is at least possible in the original model.

The protocol we now describe is the distributed analogue of List Scheduling. Based on our results about it we can derive a result analogous to that for Decreasing List Scheduling as a corollary. Corresponding to the “list” in List Scheduling, there will be a total ordering “ $>_G$ ” on the edges (including late arrivals), with the “greatest” edge corresponding to the first edge in the list. We assume that an individual vertex can determine the relative ordering between any two edges of which it is an endpoint.

The protocol uses just one type of query: “Are you prepared to perform the transfer represented by edge e ?” which we shall denote by *Query* (v, e, u), where e is the name of the edge and u and v are its endpoints, v being the vertex sending the query, and u being the addressee. There are three permitted answers: (1) “Yes,” denoted *Yes* (e, u), (2) “Call back later, I’m waiting to see whether I can execute a higher priority edge,” denoted *Later* (e, u), and (3) “No, all my ports are in use (or I am dead),” denoted *No* (e, u). The invocation of the FileSend and FileReceive protocols for edge e are denoted by *FileSend* (e) and *FileReceive* (e) respectively. Reports from these protocols are of the form *Success* (e) or *Failure* (e). An internal request for a new transfer represented by the edge e has the form *Add* (e).

Each vertex v maintains two lists of incident edges. The first is the *A-list* and contains those edges whose transfers have not yet begun and whose other endpoints are not thought to be busy. It is ordered according to $>_G$ with the largest element first. The second is the *B-list* and contains all the rest of the as yet untransferred edges, ordered in a First-in, First-out fashion. We assume that these are initialized so that the B-list is empty and all transfer requests initially known by (and involving) v are in the A-list, marked as “unqueried,” i.e., as not being the subject of any query sent by v but not yet answered. In addition, the variable N_{QUERIED} , initialized to 0, gives the current number of outstanding unanswered queries, and the variable N_{FREEPORT} , initialized to p_v , contains the current number of free ports. We shall assume N_{QUERIED} is updated automatically, whereas we shall explicitly update N_{FREEPORT} so that it reflects the number of ports *committed* to transfers, even though some of the transfers may not yet be taking place.

We are now in a position to describe the protocol.

DEMAND PROTOCOL 2 (DP2).

Repeat forever:

1. If the Protocol Message Queue is not empty, remove the first message M and do the following:
 - 1.1. If $M = \text{No}(e, u)$, move e from its current position to the end of the B-list and mark it as “unqueried.”
 - 1.2. If $M = \text{Later}(e, u)$, mark e as “unqueried” and move it to the A-list if it is not already there.
 - 1.3. If $M = \text{Yes}(e, u)$ then
 - 1.3.1. If e is in either the A-list or B-list, delete e from its list, reduce N_{FREEPORT} by 1, and invoke the appropriate one of *FileSend* (e) and *FileReceive* (e).
 - 1.3.2. If e is not in either list, do nothing. (We have already invoked the appropriate one of *FileSend* (e) and *FileReceive* (e) in

- response to a query from u .)
- 1.4. If $M = \text{Query}(u, e, v)$ then
 - 1.4.1. If $N_{\text{FREEPORT}} = 0$, send the message $No(e, v)$, add e to the A-list marked "unqueried" (if it is not already there), and delete it from the B-list (if present).
 - 1.4.2. Else if e is in the A- or B-list marked "queried," or if N_{FREEPORT} exceeds the number of edges e' that either (i) are currently marked "queried" or (ii) are marked "unqueried," satisfy $e' >_G e$, and are in the A-list, then send the message $Yes(e, v)$ to u , delete e from its list (if it is on one), reduce N_{FREEPORT} by 1, and invoke the appropriate one of $FileSend(e)$ and $FileReceive(e)$.
 - 1.4.3. Else send the message $Later(e, v)$ to u , add e to the A-list marked "unqueried" (if it is not already there), and delete it from the B-list (if present).
 - 1.5. If $M = \text{Success}(e)$, increase N_{FREEPORT} by 1.
 - 1.6. If $M = \text{Failure}(e)$ or $M = \text{Add}(e)$, add e to the A-list marked "unqueried" (if not already there), delete it from the B-list (if present), and increase N_{FREEPORT} by 1.
2. While $N_{\text{FREEPORT}} > N_{\text{QUERIED}}$ and there is an edge in the A-list or the B-list that is marked "unqueried," do the following
 - 2.1. Let e be the first edge marked "unqueried" in the concatenation of the A-list followed by the B-list, and let u be its other endpoint.
 - 2.2. Send $Query(v, e, u)$ and mark e in its list as "queried".
 END (While $N_{\text{FREEPORT}} \dots$)
- END (DEMAND PROTOCOL 2)

It is easy to verify (by induction or case analysis) that certain elementary properties are obeyed by this protocol. For instance, N_{QUERIED} is never more than N_{FREEPORT} ; a vertex, once informed of the existence of an edge, never forgets it until the corresponding transfer is successfully completed (if we count the fact that the file transfer protocols "remember" the edge while a transfer is being attempted, and reinitiate it should the transfer fail); if one endpoint of an edge invokes a file transfer protocol for an edge, then so must the other unless it dies, i.e., stops making steps in its protocol loop. Our major claim, which follows from such elementary observations, is the following "correctness" result.

THEOREM 17. *Suppose that the network is being governed by DP2 and is in a state where there is an unstarted edge, both of whose endpoints are alive and have a free port. Then so long as no vertex dies and the Communications and File Transfer protocols operate as assumed, one of the following must happen:*

- (a) an edge starts,
- (b) an edge finishes, or
- (c) a new edge gets internally requested at some vertex.

Proof. Suppose we are in a configuration where none of (a), (b), or (c) will ever occur again and no further vertices will die, and that, contrary to the theorem, there is an unstarted edge, both of whose endpoints are alive and have free ports (in what follows we shall call such an edge a *free edge*). We first show that a free edge must eventually be placed on some vertex's A-list.

So suppose no free edge is on an A-list. Then all free edges must be on B-lists. Let e be a free edge, and let it be the first such on the B-list for some vertex v . Let m

be the number of edges in the concatenation of v 's A- and B-lists that are in front of e . By assumption, the other endpoints of all these edges either are dead or have no free ports (and never will have, since no more edges will ever finish). Thus the response to any query concerning such an edge must be "No," which results in the edge being moved to the end of the B-list and the reduction of m by 1. Each time such an answer is received, we also reduce N_{QUERIED} by 1, so that in Step 2 a new query is posed, either for e or some edge ahead of e in the concatenation of the A- and B-lists. Since all queries receive answers, and all edges ahead of e must receive the answer "No," this means that eventually the number of edges ahead of e will be reduced to the point where a query must be posed for e . The other endpoint of e , being neither busy nor dead, must thus answer "Later" (it can't answer "Yes" as that would cause e to be started), which causes v to put e on its A-list.

So now let us suppose that there are free edges on A-lists. Let e be the greatest free edge (under $>_G$) having this property, and let u and v be its endpoints, with at least v having e on its A-list. We shall show that either e gets started, or else some other free edge e' with $e' >_G e$ is promoted to an A-list. Suppose not, and hence e remains the greatest free edge, in perpetuity. By an argument like that of the previous paragraph, we can assume that e eventually reaches the head of v 's A-list (after all non-free edges have received "No" answers and been demoted). Since a request for e can never receive a "No" answer, it will never leave its A-list. Assuming no other edge e' is promoted, e must thus remain at the head of its A-list in perpetuity. At some point it thus must be requested, and thereafter each time Step 2 of the protocol is reached, it will either already be under request or else will be re-requested (this happens when a "Later" response was received in Step 1). Hence, from some point on, e will always be under request when Step 1 is entered. If, from this point on, v ever reads a request $Q(u, e, v)$ off its Protocol Message Queue in Step 1, it must answer "Yes" and start a transfer. Thus e 's other endpoint u must never query for e . However, u must answer v 's queries, and it must answer them with "Later," thus insuring that e gets on u 's A-list. As soon as this happens it is only a matter of time before u must query for e , since no edge ahead of e on u 's A-list can be free by our choice of e .

Thus we have a contradiction to our assumption that e is never started nor superseded as "greatest" free edge on an A-list. Hence one of these two possibilities must occur. Since the replacement of the greatest free edge can only occur a finite number of times (there are only a finite number of edges and by assumption no edge ever loses its freeness), eventually an edge must start, the final contradiction. The theorem follows. ■

As a consequence of Theorem 17, we can conclude that, so long as dead vertices are not revived and our assumptions about the underlying protocols are valid, protocol DP2 guarantees that all files involving surviving vertices will eventually be transferred successfully. Moreover, a statement analogous to Corollary 16.1 can be made. Let us assume that no vertex ever dies, no transfer fails, and that all edges are known to their endpoints at time 0. Then Theorem 17 says that any time an edge becomes free there is a finite amount of time before either an edge starts or another edge finishes. Let $\delta(DP2, G)$ be the maximum amount of time for this to occur given the network corresponding to the file transfer graph G .

COROLLARY 17.1. *Assuming that all edges are known to their endpoints at time 0 and that no vertex ever dies and no transfer ever fails, then if s is the schedule produced by Demand Protocol 2 with ex post facto message graph G , e is the last edge finished, and p is the maximum port capacity, we have*

$$\begin{aligned} \text{MAKESPAN}(s) &\leq 2\text{OPT}(G) + L(e) \left[1 - \frac{1}{2p} \right] + 2\delta(\text{DP2}, G) \cdot |E| \\ &\leq 3\text{OPT}(G) + 2\delta(\text{DP2}, G) \cdot |E|. \end{aligned}$$

Note that the delay term includes a factor of 2 since we must include $\delta(\text{DP2}, G)$ twice for each edge (once for its start and once for its finish), whereas in Corollary 16.1 we needed to count $\delta(\text{DP1}, G)$ only once per edge. The delay term is probably even worse than twice that for DP1, however, since $\delta(\text{DP2}, G)$ does not directly correspond to $\delta(\text{DP1}, G)$. The model is different, involving many more protocol messages. Moreover, as hinted in our proof of Theorem 17, $\delta(\text{DP2}, G)$ may itself be proportional to $|E|$, since we may have to wait for all the free edges to be “promoted” before we actually get around to starting any of them. However, if $\delta(\text{DP2}, G)$ can be kept small relative to $\text{OPT}(G)$, then this protocol can be adapted to approximate the $2.5\text{OPT}(G)$ guarantee of Decreasing List Scheduling. The key is in an appropriate definition for $>_G$.

In order to make this definition, we shall impose just a few more (reasonable) restrictions on the model. We assume that each vertex v has a unique identification number $ID(v)$, that each copy e of a multi-edge has a distinct name $Name(e)$ (presumably the name of the corresponding file), and that both endpoints of an edge have identical estimates of its length. (We do not require that the actual *ex post facto* length agree precisely with this estimate, however.) Given these definitions it is straightforward to devise an ordering $>_G$ that will behave as desired and whose restriction to the edges involving a given vertex can be computed locally, so long as that vertex knows the ID’s of its neighbors. Define “ $>_D$ ” as follows. Let $e = \{v, u\}$ and $e' = \{v', u'\}$ be two distinct edges with $ID(v) > ID(u)$ and $ID(v') > ID(u')$. Then $e >_D e'$ if (a) $L(e) > L(e')$, or if (b) $L(e) = L(e')$ and $ID(v) > ID(v')$, or if (c) $L(e) = L(e')$, $v = v'$, and $ID(u) > ID(u')$, or if (d) $L(e) = L(e')$, $v = v'$, $u = u'$, and $Name(e)$ is lexicographically prior to $Name(e')$.

COROLLARY 17.2. *Suppose that the situation is as described in Corollary 17.1, that no actual edge length differs from the edge’s estimated length by more than ϵ , and that the order relation $>_G$ used in DP2 is $>_D$. Then we have*

$$\text{MAKESPAN}(s) \leq \frac{5}{2} \text{OPT}(G) + 2\epsilon + 2\delta(\text{DP2}, G) \cdot |E|.$$

Proof. Given Corollary 17.1, the result will follow if we can show that the last edge e to finish has actual length no more than $(1/2)\text{OPT}(G) + 2\epsilon$. Suppose not. Then the estimated length of e exceeds $(1/2)\text{OPT}(G) + \epsilon$, and any edge e' with $e' >_D e$ must have actual length exceeding $(1/2)\text{OPT}(G)$. We may also assume that e started after time $2\delta(\text{DP2}, G) \cdot |E|$, since otherwise the result would follow from the fact that the actual length of e is at most $\text{OPT}(G)$. Let us examine what happened at time 0 and immediately thereafter. Initially e was on the A-list for both its endpoints u and v . We may assume that it was among the first p_v edges on the A-list for v , as otherwise v would have had $p_v + 1$ edges of length exceeding $(1/2)\text{OPT}(G)$, which is impossible. Similarly, e must have been among the first p_u edges on the A-list for u . So consider the first time v executed Step 1 of DP2. If the Protocol Message Queue was not empty and the first message was a query from u concerning e , v would have had to say “Yes” as $N_{\text{FREEPORT}} = p_v$ and $N_{\text{QUERIED}} = 0$. Otherwise, at Step 2 v must have queried u about e . Turning now to u , we note that u must also have either said

“Yes” to a query about e the first time it entered Step 1, or sent a query itself about e at Step 2. If neither u nor v says “Yes” to a query about e during its first pass through the protocol’s main *while* loop, then both will have sent a query about e . One of the two must thus receive the other’s query before it receives the answer to its own. Suppose without loss of generality that this happens to v . By Step 1.4.2 of DP2, v must then say “Yes.” We conclude that one of u and v must say “Yes” to e , and hence the transfer must take place, without a query about e ever receiving a “No” answer. In other words, e remains free from time 0 until the time it is transferred. Thus by Theorem 17 and the definition of $\delta(DP2, G)$, e must start no later than time $2\delta(DP2, G) \cdot |E|$, a contradiction.

We thus can conclude that the actual length of the last task to finish is at most $(1/2)OPT(G) + 2\epsilon$, and the Corollary follows. ■

Note that the proof of Corollary 17.2 depends heavily on the fact that all vertices start executing DP2 at the same time (time 0). Otherwise u or v might have received a “No” answer from the other endpoint because it was not yet awake. This would wreak havoc with the argument of Corollary 17.2, although it would have only a limited effect on Theorem 17 and Corollary 17.1, as long as the spread between vertex start-up times is itself limited.

5. Directions for further study. In this paper we have introduced the problem of File Transfer Scheduling. As far as we know, there has been little previous theoretical analysis of this problem (although a more-limited variant, posed in terms of a link-testing problem, has been discussed in [4]). There are a wide variety of directions for further research, such as improving on our results, extending them to more general versions of the model, and investigating other optimization criteria.

Within the model, the major question we see is whether a polynomial time heuristic can be devised that will provide a better guarantee than the $(5/2)OPT(G)$ of Decreasing List Scheduling. (For those interested in the arcana of NP-completeness, there is also the question of the complexity of FILE TRANSFER SCHEDULING for message graphs that are odd cycles of equal-length multi-edges.)

The most important extension of the model includes the possibility of *forwarding*. This becomes relevant if u wants to send a file to v , but has no direct link, and hence must send it to an intermediary, say w , who will then send it on to v . Forwarding may be helpful even when direct links are available. Suppose both u and v are so busy that times when they simultaneously have free ports are rare. In this case a common neighbor w , which has a free port most of the time, might take the file from u when u has the time and then hold it until v is free. Can our results be extended to include one or both of these types of forwarding?

Another extension of the model would be to include the real-life distinction that often arises between dial-out and dial-in ports, as in the case of a computer system with just one automatic call unit, but many incoming lines. In such a situation it might be that two nodes each have many ports, but can only be involved jointly in two simultaneous file transfers, since each transfer occupies one of the dial-out lines.

Finally, there is the question alluded to earlier about appropriate optimization criteria. Makespan is a reasonable standard in the case when one is trying to perform all transfers during a particular time interval, or when all the nodes are going to be tied up in the transfer process until all transfers are completed. However, in many systems, file transfers may be going on all the time, or nodes may be free to do other things once their own transfers are complete. The latter case suggests a criterion such as “average finishing time” (over all nodes), and the former suggests a more dynamic

measure such as the "average time in system" for a requested transfer, i.e., the average delay between the time a node decides it wants the transfer until the transfer is accomplished. For such criteria, average case analysis may well be more meaningful than the worst case analysis performed here, especially if one wishes to analyze the system when nodes or transfers may fail.

Acknowledgment. We are indebted to B. Gopinath for originally posing the basic file transfer scheduling model to us and for encouraging us to address the issues discussed in this paper.

REFERENCES

- [1] C. BERGE, *Graphs and Hypergraphs*, North-Holland Publishing Company, Amsterdam, 1973.
- [2] E. G. COFFMAN, JR., M. R. GAREY, AND D. S. JOHNSON, *An application of bin-packing to multiprocessor scheduling*, SIAM J. Comput., 7 (1978), pp. 1-17.
- [3] R. COLE AND J. HOPCROFT, *On edge coloring bipartite graphs*, SIAM J. Comput., 11 (1982), pp. 540-546.
- [4] S. EVEN, O. GOLDREICH, AND P. TONG, *On the NP-completeness of certain network-testing problems*, Report No. 230, Department of Computer Science, Technion, Haifa, Israel, 1981.
- [5] D. K. FRIESEN, *Tighter bounds for the MULTIFIT processor scheduling algorithm*, SIAM J. Comput., 13 (1982), pp. 170-181.
- [6] H. N. GABOW, private communication (1982).
- [7] H. N. GABOW AND O. KARIV, *Algorithms for edge coloring bipartite graphs and multigraphs*, SIAM J. Comput., 11 (1982), pp. 117-129.
- [8] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, 1979.
- [9] M. K. GOLDBERG, *Edge-coloring of multigraphs: recoloring technique*, J. Graph Theory, to appear.
- [10] M. K. GOLDBERG, private communication (1983).
- [11] I. HOLYER, *The NP-completeness of edge-coloring*, SIAM J. Comput., 10 (1981), pp. 718-720.
- [12] S. SAHNI, *Algorithms for scheduling independent tasks*, J. Assoc. Comput. Mach., 23 (1976), pp. 116-127.
- [13] C. E. SHANNON, *A theorem on colouring the lines of a network*, J. Math. Phys., 28 (1949), pp. 148-151.
- [14] V. G. VIZING, *On an estimate of the chromatic class of a p-graph*, Diskret. Analiz., 3 (1964), pp. 25-30. (In Russian.)

DATA STRUCTURES FOR ON-LINE UPDATING OF MINIMUM SPANNING TREES, WITH APPLICATIONS*

GREG N. FREDERICKSON†

Abstract. Data structures are presented for the problem of maintaining a minimum spanning tree on-line under the operation of updating the cost of some edge in the graph. For the case of a general graph, maintaining the data structure and updating the tree are shown to take $O(\sqrt{m})$ time, where m is the number of edges in the graph. For the case of a planar graph, a data structure is presented which supports an update time of $O((\log m)^2)$. These structures contribute to improved solutions for the on-line connected components problem and the problem of generating the K smallest spanning trees.

Key words. connected components, data structures, edge insertion and deletion, K smallest spanning trees, minimum spanning tree, on-line computation, planar graphs

1. Introduction. Consider the following on-line update problem: A minimum spanning tree is to be maintained for an underlying graph, which is modified repeatedly by having the cost of an edge changed. How fast can the new minimum spanning tree be computed after each update? In this paper we present novel graph decomposition and data structures techniques to deal with this update problem, including a useful characterization of the topology of a spanning tree. Furthermore, while dynamic data structures have been applied with success to various geometric problems [OV], [WL], our results are among the first [ST], [H11] in the realm of graph problems.

Let m be the number of edges in the graph, and n the number of vertices. The current best time to find a minimum spanning tree is $O(m \log \log_{(2+m/n)} n)$ [CT], [Y]. If only straightforward descriptions of the underlying graph and its current minimum spanning tree are maintained, then it has been shown in [SP] that the worst-case time to perform an edge-cost update is $\Theta(m)$. The problem of determining the replacement edges for all edges in the spanning tree can be solved in $O(m\alpha(m, n))$ time [T2], where $\alpha(\cdot, \cdot)$ is a functional inverse of Ackermann's function [T1]. However, that solution is essentially static, so that actually performing replacements can necessitate considerable recomputation.

We show how to maintain information about the graph dynamically so that edge costs can be updated repeatedly with efficiency. After each edge cost change, the change in the minimum spanning tree is determined, and the data structures are updated. We are able to realize an $O(\sqrt{m})$ update time. Moreover, if the underlying graph is planar, we show how to achieve an $O((\log m)^2)$ update time. Our structures require $O(m)$ space and $O(m)$ preprocessing time, aside from the time to find the initial minimum spanning tree. These compare favorably with those developed recently in [H12], which realize $O(n \log n)$ update times.

Our results are both of practical and theoretical interest. On the one hand, a minimum spanning tree may be used to connect the nodes of a communications network. Variable demand, or transmission problems, may cause the cost of some edge in the network to change, and the tree will need to be reconfigured dynamically. On the other hand, by focusing on edge cost changes, we have formulated a natural version of the problem of updating a minimum-cost base of a matroid [W]. (In this case, the matroid is a graphic matroid.) Our work leads naturally into the updating of

* Received by the editors August 1, 1983, and in revised form May 25, 1984. This research was supported in part by the National Science Foundation under grant MCS-8201083.

† Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907.

minimum-cost bases of certain simple matroid intersections. These are investigated in [FS1], [FS2], in which our data structures are used extensively. The problem of maintaining a minimum spanning tree when vertices are inserted and deleted has been studied in [SP], [CH], but the best performance to date is $O(n^2)$. This suggests that because of its connection to matroids, the edge-update problem is perhaps more natural than the vertex-update problem.

We also show how to apply our data structures to a number of related problems to yield improved performance bounds. We cast the problems of edge insertion and deletion into an edge update framework, and realize $O(\sqrt{m_t})$ update times, where m_t is the current number of edges in the graph. Using this, we improve on the update time for the on-line connected components problem in a graph in which edges are being inserted and deleted. The problem is to maintain a data structure so that a query asking if two vertices are in the same connected component can be answered in constant time. A version involving deletions only was examined in [ES], for which the total time for m updates was $O(mn)$. A more general version has been discussed recently in [HI1], for which $O(n)$ time per individual update was realized. Our solution uses $O(\sqrt{m_t})$ time per update.

Our data structures can also be used in generating the K smallest spanning trees in increasing order [G]. The best published solution [KIM] requires $O(m \log \log_{(2+m/n)} n + Km)$ time and $O(K + m)$ space. Quite recently, this has been improved in [HI2] to $O(Kn(\log n)^2 + m \log n)$ time at the expense of $O(Kn \log n + m \log n)$ space. We improve the time complexity for instances with relatively small K . If K is $O(\sqrt{m})$, our solution uses $O(m \log \log_{(2+m/n)} n + K^2\sqrt{m})$ time and $O(m)$ space. If the graph is planar, then the solution in [KIM] uses $O(Kn)$ time and $O(K + n)$ space. If K is $O(n/(\log n)^2)$ and the graph is planar, our solution uses $O(n + K^2(\log n)^2)$ time and $O(n)$ space.

A preliminary version of this paper appeared in [F].

2. Preliminaries. There are several cases to be handled in edge-cost updating. The cost of an edge may either be increased or decreased, and this edge may currently be either in the minimum spanning tree or not in the tree. If the cost of a tree edge is decreased, or the cost of a nontree edge is increased, then there will be no change in the minimum spanning tree.

In the two remaining cases, the minimum spanning tree may be forced to change. However, at most one edge will leave the tree, and one edge will enter the tree. If the cost of a nontree edge (v, w) is decreased, then this edge may enter the tree, forcing out some other edge. This case may be detected by determining if the maximum cost of an edge on the cycle that (v, w) induces in the tree has greater cost than $c(v, w)$. An obvious implementation of this test would use $\Theta(n)$ time. A faster approach uses the dynamic tree structures of Sleator and Tarjan [ST]. A maximum cost edge (x, y) can be found using the operations $evert(v)$ and $findmax(w)$. The operation $evert(v)$ makes v the root of the dynamic tree structure, and $findmax(w)$ finds the maximum cost edge on the path from w to the root. The dynamic tree may be updated using $cut(x, y)$ and $link(v, w)$. The operation $cut(x, y)$ deletes edge (x, y) from the tree, and $link(v, w)$ adds edge (v, w) . As discussed in [ST], the worst-case time required to perform these operations is $O(\log n)$.

The most interesting case is if the cost of a tree edge (x, y) increases. Then the edge may be replaced by some nontree edge. This case may be detected by determining if the minimum cost nontree edge (v, w) that connects the two subtrees created by removing (x, y) has cost less than $c(x, y)$. In the worst case, there can be $\Omega(m)$ edges

that are candidates for the replacement edge. Consequently, this case appears to be the most troublesome to deal with.

Our structures are designed to handle graphs in which no vertex has degree greater than three. Given a graph $G_0 = (V_0, E_0)$, we shall produce a graph $G = (V, E)$ in which each vertex satisfies this degree constraint. A well-known transformation in graph theory [Hy, p. 132] is used. For each vertex v of degree $d > 3$, where w_0, \dots, w_{d-1} are the vertices adjacent to v , replace v with new vertices v_0, \dots, v_{d-1} . Add edges $\{(v_i, v_{(i+1) \bmod d}) \mid i = 0, \dots, d-1\}$, each of cost 0, and replace the edges $\{(w_i, v) \mid i = 0, \dots, d-1\}$ with $\{(w_i, v_i) \mid i = 0, \dots, d-1\}$, of corresponding costs.

Let $n' = |V|$ and $m' = |E|$. Then it is not hard to see that $n' \leq 2m$ and $m' \leq 3n'/2 \leq 3m$. Thus there are $\Theta(m)$ vertices in the new graph G , and $\Theta(m)$ storage is required. Given a minimum spanning tree $T_0 = (V_0, E_{0t})$ for G_0 , it is easy to find a minimum spanning tree $T = (V, E_t)$ for G . For each new vertex v , include $\{(v_i, v_{i+1}) \mid i = 0, \dots, d-2\}$, and replace any edge (w_i, v) with the corresponding edge (w_i, v_i) . In §§ 3-7 of this paper, we shall assume that we are dealing with graph of $O(m)$ vertices, in which each vertex has degree no greater than 3.

3. Topological partitions of the vertex set. In this section we examine a simple solution to our problem that allows for $o(m)$ update times. We first give a procedure for organizing vertices into clusters, based on the topology of the minimum spanning tree. Using this partition, we show how to achieve $O(m^{2/3})$ update times.

We partition the vertices of the minimum spanning tree T on the basis of the topology of the tree. Let z be a positive integer to be specified later. Let E' be a set of edges whose removal from T leaves connected components with between z and $3z-2$ vertices. The vertex set of each resulting connected component will be called a *vertex cluster*, and the collection of clusters will be called a *topological partition of order z* . Such a partition always exists and is in general not unique.

Given a tree with more than $3z-2$ vertices, and of maximum degree 3, a topological partition may be generated as follows. Perform a depth-first search of T starting at any leaf vertex, which shall be identified as the root. Now call $csearch(\text{root})$, where $csearch(v)$ partitions v and its descendants into zero or more clusters of size between z and $3z-2$, and one set of size between 0 and $z-1$. The set is returned to the calling procedure.

```

proc csearch(v)
  local clust
  clust ← {v}
  for each child w of v do clust ← clust ∪ csearch(w) endfor
  if |clust| < z then return(clust)
  else print(clust); return(ϕ) endif
endproc

```

Let a procedure FINDCLUSTERS be the procedure that initially calls $csearch$. If $csearch$ returns a nonempty set to FINDCLUSTERS, FINDCLUSTERS should union it in with the last cluster printed.

LEMMA 1. Procedure FINDCLUSTERS partitions the vertex set of a spanning tree with maximum degree 3 into vertex clusters of cardinality between z and $3z-2$ in $O(m)$ time.

Proof. It is not hard to see that the clusters which are output do form connected components with respect to tree T . Since vertices are of degree no greater than 3, and the root has degree 1, each vertex in T will have at most two children. Since sets of

size at most $z - 1$ are returned by *csearch*, and any vertex will have at most two children, any cluster formed at a vertex v will have size at most $2z - 1$. A set of at most $z - 1$ vertices can be returned to FINDCLUSTERS, and when this set is unioned with the last cluster printed out, a cluster of size at most $3z - 2$ will result. Thus all clusters are within the prescribed size bounds. If the sets are implemented as linked lists, then the whole procedure will require time proportional to the size of T . \square

The number of vertex clusters will be $\Theta(m/z)$. If $z \geq \sqrt{m}$, then there will be $O(\sqrt{m})$ vertex clusters. Once the vertices are partitioned, partition the edges in $E - E_t$ into sets E_{ij} such that an edge in E_{ij} has one endpoint in vertex cluster V_i and the other endpoint in vertex cluster V_j . Thus there will be $O(m)$ sets E_{ij} . For each set E_{ij} , a minimum cost edge is determined. Both of these tasks can be performed in $O(m)$ time. Thus, once a minimum spanning tree for G_0 is determined, all other initialization will take $O(m)$ time. The amount of space used may be seen to be $O(m)$.

We now describe how to handle the two more interesting update operations. Suppose the cost of a nontree edge (v, w) is decreased, so that tree edge (x, y) must be removed from the tree, and (v, w) must be added. Edge (x, y) can be determined in $O(\log m)$ time, as discussed in § 2. Several cases are possible. If x, y, v and w are in the same cluster, or if x and y are in different clusters, then the cluster need not be changed.

The crucial case is x and y are in the same vertex cluster, say V_i , which does not contain both v and w . Then this vertex cluster must be split into V'_i and V''_i , and the sets E_{ij} must be split for all j . Since $|V_i|$ is $O(z)$ and each vertex is of degree no greater than 3, $|\cup_j E_{ij}|$ is $O(z)$. Thus the splitting may be carried out in $O(z)$ time. If either V'_i or V''_i has fewer than z vertices, then combine it with a neighboring vertex cluster. If this neighbor now has more than $3z - 2$ vertices, it can be split into two clusters by using *csearch*. The total time to determine and perform whatever splits are necessary will be $O(z)$.

If the cost of a tree edge (x, y) is increased, then a minimum cost replacement edge $(v, w) \neq (x, y)$ must be found. To find (v, w) , do the following. If (x, y) connects two vertices in the same cluster V_i , split V_i into V'_i and V''_i , and adjust the sets E_{ij} , as above. Removing (x, y) will partition the vertex clusters into two sets. Check the minimum cost edges between every pair of vertex clusters V_i and V_j , where V_i and V_j are in different sets of the partition. Choose the minimum of these to be (v, w) . There can be $\Theta(m/z)$ vertex clusters in each set of the partition, so that the time required to check all pairs of vertex clusters will be $\Theta(m^2/z^2)$. As before, splitting V_i into V'_i and V''_i will use $O(z)$ time.

We may realize best performance for this approach if we choose $z = \lceil m^{2/3} \rceil$. This structure is called structure I.

THEOREM 1. *Structure I allows the on-line edge-update problem for minimum spanning trees to be solved in $O(m^{2/3})$ time per update, using $O(m)$ space and $O(m)$ preprocessing time, aside from the time required to find the initial minimum spanning tree.*

Proof. The preprocessing requirements have already been established. The update times are dominated by $O(z + m^2/z^2)$. Choosing $z = \lceil m^{2/3} \rceil$ gives the desired result. \square

4. Topology trees. In the previous section we showed how to partition the vertices into clusters to improve update times. In this section we show how to build clusters of clusters, yielding a hierarchical characterization of the minimum spanning tree. This characterization is then used in the next section to aggregate edge set information.

Given a spanning tree T in which each vertex has degree no greater than three, we define a data structure that describes the topology of the tree in a convenient

manner. Let the *external degree* of a vertex cluster be the number of spanning tree edges with exactly one endpoint in the vertex cluster. A *multi-level topological partition* of the set of vertices satisfies the following:

1. For each level i , the vertex clusters at level i will form a partition of the set of vertices.
2. A vertex cluster at level 0 will contain a single vertex.
3. A vertex cluster at level $i > 0$ is either
 - a. the union of 2, 3 or 4 vertex clusters of level $i - 1$, where the clusters are connected together in one of the three ways shown in Fig. 1, and the external degree is no greater than 3; or
 - b. a vertex cluster of level $i - 1$ whose external degree is 3.

A *topology tree* for spanning tree T is a tree in which each internal node has at most four children, and all leaves are at the same depth, such that:

1. a node at level i in the topology tree represents a vertex cluster in level i of the multi-level topological partition, and
2. a node at level $i > 0$ has children which represent the vertex clusters whose union is the vertex cluster it represents.

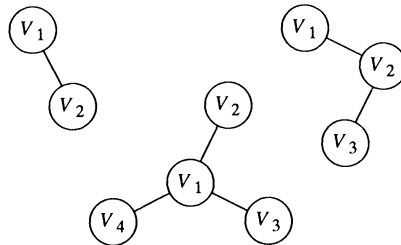


FIG. 1. The allowable topologies for vertex clusters that may be unioned together.

Given the vertex clusters for level $i - 1$, we can determine how the vertex clusters are unioned together to give vertex clusters at level i . Consider a spanning tree T_{i-1} derived from T by collapsing each vertex cluster of level $i - 1$ to a single vertex. Apply procedure FINDCLUSTERS to the tree T_{i-1} , with parameter $z = 2$. This will identify clusters of vertices in the tree T_{i-1} of cardinality two, three or four, grouped as in Fig. 1. For each cluster in T_{i-1} that would have external degree greater than 3, subdivide the cluster so that the resulting subsets each have degree 3. The vertices in T_{i-1} so grouped, represent the vertex clusters of level $i - 1$ that should be unioned to get vertex clusters on level i . An example of tree T is shown in Fig. 2. The corresponding topology tree is shown in Fig. 3.

LEMMA 2. Let n be the number of vertices in a spanning tree T . The height of a corresponding topology tree will be $\Theta(\log n)$.

Proof. Consider the generation of the vertex clusters of level $i > 0$, using the vertex clusters of level $i - 1$ and the corresponding tree T_{i-1} . Over half the vertices in T_{i-1} will be of degree less than three, and all of them will participate in a unioning from level $i - 1$ to i . Since one vertex cluster will replace at least two for each vertex cluster that is unioned, fewer than

$$n - \frac{1}{2}(\frac{1}{2}n) = \frac{3}{4}n$$

vertex clusters will remain after the unionings.

Since the number of vertex clusters unioned at each level is at least a constant fraction of the remaining number, the number of levels until a single vertex is reached is $O(\log n)$. It follows that the topology tree is of height $O(\log n)$. \square

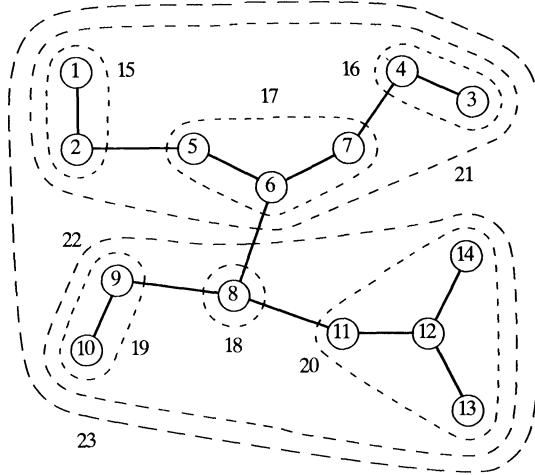


FIG. 2. A multilevel partition of the vertices in a spanning tree.

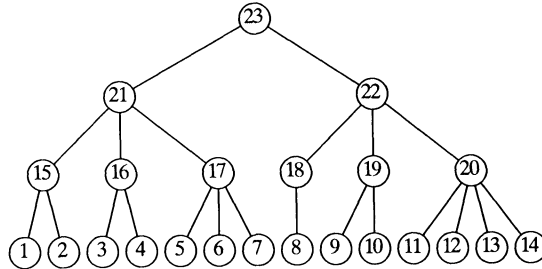


FIG. 3. The topology tree corresponding to the topological partition in Fig. 2.

LEMMA 3. A topology tree can be generated for a given spanning tree T in time proportional to the number of vertices in T .

Proof. Let n be the number of vertices in T . The first iteration will require $O(n)$ time. From the proof of Lemma 2, at least $\frac{1}{4}$ of the remaining vertices are removed on any iteration. Thus total time will be $O(\sum_{i=0}^{\infty} n(\frac{3}{4})^i)$, which is $O(n)$. \square

We are interested in the operations of deleting an edge from the minimum spanning tree, and connecting two trees via an edge into a minimum spanning tree. These spanning tree operations will force the corresponding operations of splitting a topology tree and merging two topology trees. We shall show that each of these topology tree operations can be performed in $O(\log m)$ time.

At first glance, merging and splitting of topology trees would appear similar to the merging and splitting of 2-3 trees [AHU]. However the topology trees represent clusters that satisfy, among other things, degree constraints and thus must be handled carefully. Adding an edge to merge two trees into a spanning tree may cause the external degree of a vertex cluster to increase from 3 to 4. In this case the vertex cluster must be split, and the tree must be restructured accordingly. On the other hand, deleting an edge may make it possible to include a vertex cluster in some union at a lower level than before.

We first discuss in detail the merging of two topology trees. Consider the edge that is added to connect the two corresponding trees to give the spanning tree. If some

vertex cluster has its external degree increased from 3 to 4, choose the most deeply nested such cluster, say W . It must be the union of at least two clusters, and its constituent clusters can be regrouped into two adjacent vertex clusters, W' and W'' , such that the external degree of each is now three. We thus replace a vertex cluster W , originally of external degree 3, and now of degree 4, with two vertex clusters, each of degree 3. An example in which a cluster must be split into two clusters is shown in Fig. 4a, and the resulting clusters are shown in Fig. 4b. The resulting clusters may force the cluster in which they are located to be split, and this effect may propagate upwards in the multilevel partition. The next level up from the cluster in Fig. 4a is shown in Fig. 4c, with the result in Fig. 4d.

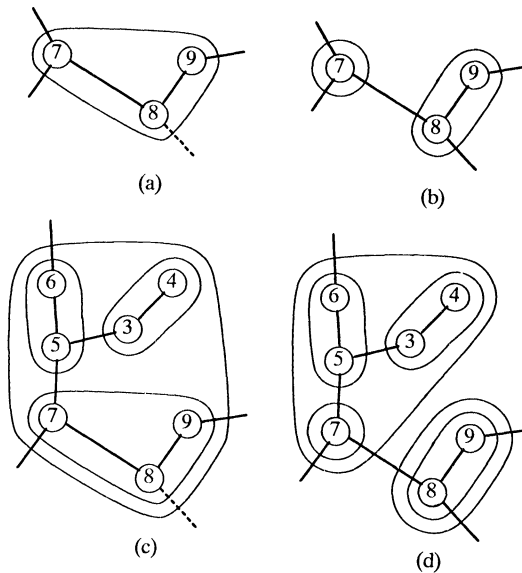


FIG. 4. An example of external degree increasing from 3 to 4.

Once any critical change in external degree has been handled, the root of the topology tree of smaller height can be joined to the appropriate level of the other topology tree. The operation is similar to inserting a node as a child of some node in a 2-3 tree, in that the insertion of the new node may force the parent and children to be reorganized so that there are two parents, and this effect may then propagate upward. It is not hard to see that nodes along only one path to the root are affected. An example is shown in Fig. 5, with levels beneath the root of the smaller topology tree not shown in either tree. The multilevel partition and topology trees are shown before the edge insertion in Figs. 5a and 5b, and after the insertion in Figs. 5c and 5d. The set V_9 becomes a child of V_{12} , which is then split into V_{14} and V_{15} , which then forces the splitting of V_{13} into V_{16} and V_{17} .

We now discuss the splitting of a topology tree. The edge is deleted, and all clusters containing that edge are split. These clusters are represented by nodes on a path in the topology tree. The pieces of the topology tree are merged back into two trees, in a fashion similar to what is done when fragments of a 2-3 tree are merged after a splitting. Here again the constraints on the clustering shown in Fig. 1 must be preserved. An example of clusters that are split is shown in Fig. 6a, and the resulting clusters for the two trees are shown in Fig. 6b.

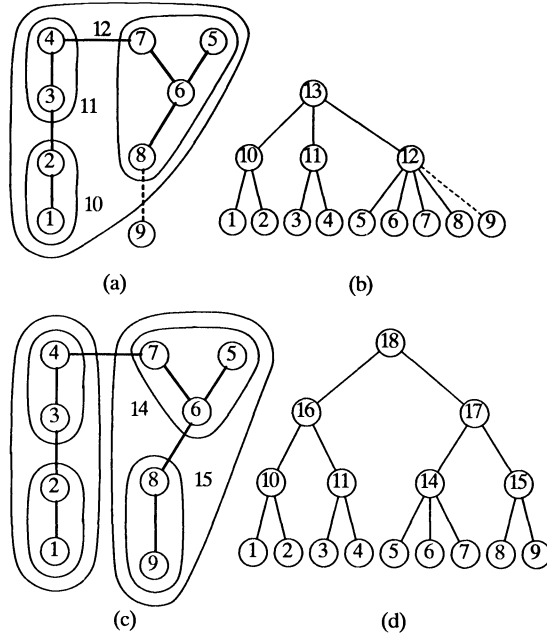


FIG. 5. An example of inserting a tree edge and merging two topology trees.

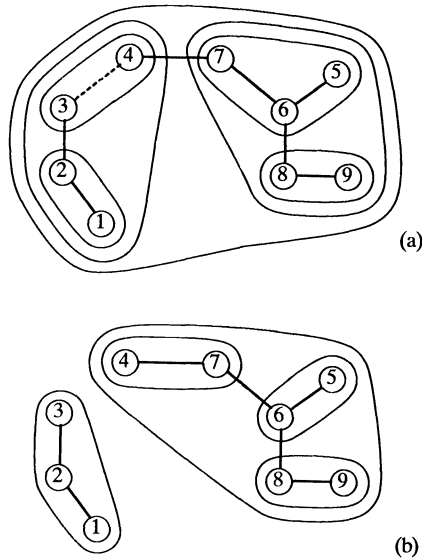


FIG. 6. An example of deleting a tree edge and splitting a multilevel partition.

Suppose there is a vertex cluster that is an only child and has had its external degree drop from 3 to 2. Choose the most deeply nested such cluster, say W . Identify the cluster W' at the same level as W in the multilevel partition that has the lowest common ancestor with W of such clusters in the topology tree. Combine W and W' , rearranging the enclosing clusters as necessary. The newly formed cluster may need to be split, because it is not one of the three forms in Fig. 1. An example of this is shown in Figs. 7a and 7b. Otherwise, there will be one fewer node, and this may cause the combinations to propagate back up in the tree. An example is shown in Fig. 7c,

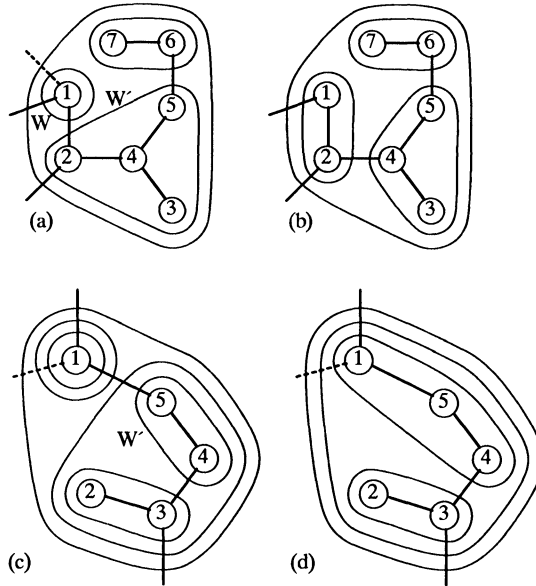


FIG. 7. Examples of external degree decreasing from 3 to 2.

with an intermediate result shown in Fig. 7d. (The outermost cluster shown must still be unioned with some other cluster at its level.)

THEOREM 2. *The time required to perform a split of a topology tree, caused by the deletion of an edge in a spanning tree, or to merge two topology trees, caused by adding an edge to create a spanning tree, is $O(\log n)$.*

Proof. From the previous discussion, it may be seen that a constant amount of work is done for each node along a constant number of paths in the topology tree. The theorem then follows. \square

5. Aggregating edge costs using topology trees. In § 3 we outlined a first strategy for updating minimum spanning trees on-line, using a partition of the vertices based on the topology of the minimum spanning tree. We determined that an expensive operation is finding an edge to replace a tree edge that has increased in cost. This operation could take time proportional to the square of the number of vertex clusters. In this section we use the topology tree described in the last section and show how to avoid examining so many edge sets, by aggregating edge set information in a manner based on the topology tree. Using this approach, we show how to achieve $O(\sqrt{m \log m})$ update times.

We would like to generate a data structure in the following manner. Shrink each vertex cluster in a topological partition to a single vertex, yielding a shrunken tree T_r . Now generate a topology tree for T_r . Unfortunately, this is not in general possible, since vertices in T_r may have degree greater than 3. The difficulty is in our rather simple definition of a topological partition, which we now extend to a *simply-connected topological partition*. Such a partition consists of $\Theta(m/z)$ vertex clusters of size $O(z)$, such that any cluster is adjacent to at most three other clusters in the spanning tree, and any cluster with fewer than z vertices must have external degree 3.

Procedure *csearch* from § 3 can be modified to generate the desired partition. Besides returning a set of vertices, the procedure should return the current external degree of the set. The size and external degree of a set generated at v can then be

determined. If this set has at least z vertices or has external degree 3, then it should be printed out. The set generated at v will never have external degree greater than 3, for the following reason. As before, each vertex in T will have at most two children. Suppose nonempty sets of vertices are returned from recursive calls to each child. Each of these sets will have external degree at most two. But of this degree of at most two, one was contributed by the child's adjacency to v , which will not be counted. Hence the external degree of the set generated at v will be at most three: at most one from each of the at most two children, plus one for the adjacency of v with its parent (if any).

The simply-connected partition will induce shrunken tree T_s . Note that each leaf in T_s will represent a vertex cluster of size between z and $3z - 2$. This follows since such a cluster, generated at vertex v , will have in effect no external degree contributed by its children. Such a set will not have external degree equal to three, so it is output only because its size is at least z . Hence there will be $O(m/z)$ leaves in T_s . Every vertex cluster of cardinality less than z will be represented by a vertex of degree 3 in T_s . Since there will be fewer vertices of degree 3 than leaves in T_s , there will be $\Theta(m/z)$ vertex clusters in a simply-connected partition. We call these vertex clusters *basic vertex clusters*.

We may now generate a topology tree for tree T_s , the tree resulting from shrinking basic vertex clusters in a simply-connected topological partition. We show how to use these structures to improve update times. For each basic vertex cluster V_i , we maintain an image of the topology tree. At the leaf representing basic vertex cluster V_j in tree i , store the set E_{ij} , along with the minimum cost edge in that set. If there is no such edge, then assume a default cost of ∞ . At each internal node in the topology tree, maintain the minimum value from among its children. Thus the topology tree is augmented to maintain a heap on edge costs. The space required by the topology tree for one cluster V_i will be $\Theta(m/z)$ for the nodes, and $\Theta(z)$ for the elements in $\cup_j E_{ij}$. Thus total space requirements for $\Theta(m/z)$ trees for all the clusters will be $\Theta(m^2/z^2 + m)$, which is $\Theta(m)$ if $z \geq \sqrt{m}$.

Given a basic vertex cluster V_j , suppose we wish to find a path from the root to the leaf representing V_j in V_i 's copy of the topology tree. It is sufficient to maintain an original copy of the topology tree with pointers from children to parents. The location of basic vertex cluster V_j in V_i 's copy can be found by tracing up from V_j in the original copy of the topology tree. Thus locating the path from the root to basic vertex cluster V_j will use $O(\log(m/z))$ time.

We now consider handling the two more interesting update operations. If the cost of a nontree edge is decreased, then finding the edge to replace, splitting a basic vertex cluster, and recombining the pieces is similar to that discussed before, except that now there are consequences in terms of the structure of the topology tree. We have already discussed how to split and merge topology trees. In particular, a topology tree can be split on a leaf representing basic vertex cluster V_i in $O(\log(m/z))$ time. Merging two topology trees that are to be joined via an edge will use $O(\log(m/z))$ time to adjust external degrees, and $O(h_1 - h_2)$ time to merge the topology trees, where h_1 and h_2 are their heights. It is straightforward to maintain the heap property on the topology trees as they are merged or split. Since each image of the topology tree must be changed, total time is $O((m/z) \log(m/z))$ for all the topology tree manipulations.

In the case in which the cost of a tree edge is increased, we can use the topology trees to find the replacement edge for (x, y) more quickly than before. If x and y are in the same basic vertex cluster V_i , split V_i into V_i' and V_i'' . If either is too small, given its external degree, combine it with a neighboring basic cluster, if there is one, and

adjust the upper levels of the topology tree as necessary. Now split each copy of the topology tree on edge (x, \cdot) to give two topology trees for each copy before. This split induces a partition of the set C of basic vertex clusters into C' and C'' . In Fig. 6b, for example, C' would consist of basic clusters 1, 2 and 3, while C'' would consist of the remaining basic clusters 4 through 9. For each basic vertex cluster V_i , one of its now two topology trees will be a heap on edge costs for edges with one endpoint in V_i and the other endpoint in a cluster in C' , and the other tree will be a heap on edge costs for edges with one endpoint in V_i and the other in a cluster in C'' .

We find the minimum cost replacement edge (u, v) as follows. For each basic vertex cluster V_i in one of the sets, say C' , consider V_i 's topology tree for the other set C'' . Take the minimum value from among those in the roots of all such topology trees. If this value is smaller than the new cost of edge (x, y) , then the edge corresponding to this value becomes the replacement edge.

Once the minimum cost replacement edge (u, v) has been chosen, the topology trees can be merged on this edge. Choosing $z = \lceil \sqrt{m \log m} \rceil$, we get structure II.

THEOREM 3. *Structure II allows the edge-update spanning tree problem to be solved in $O(\sqrt{m \log m})$ time per update, using $O(m)$ space and $O(m)$ preprocessing time, aside from the time to find the initial minimum spanning tree.*

Proof. Splitting and merging the basic vertex sets will use time $O(z)$. Splitting $\Theta(m/z)$ copies of the topology tree on an edge will take time $O((m/z) \log(m/z))$, and merging them will take the same. The time needed to examine the roots of $O(m/z)$ topology trees for the replacement edges will be $O(m/z)$. \square

6. The 2-dimensional topology tree. It is possible to improve the update time over that of structure II by doing the following. In structure II there is a separate copy of the topology tree for every basic vertex cluster V_i . If we combine all the images of the topology tree into one large tree, we can realize slightly faster update times. The leaves of the large tree will be essentially the same as the set of leaves in all copies of the topology tree, with one leaf for each pair of basic vertex clusters. The root of the large tree may be viewed as the union of the roots of all of the copies. Other internal nodes may be viewed as the unions of various internal nodes in the copies of the topology trees. The organization of the large tree will be such that the time to split or merge the structure will be $\Theta(m/z)$, rather than the $\Theta((m/z) \log(m/z))$ of structure II.

We define the *2-dimensional topology tree* in terms of the topology tree. Let V_α and V_β be vertex clusters represented by nodes at the same level in the topology tree. Then there is a node labelled with $V_\alpha \times V_\beta$ in the 2-dimensional topology tree, which represents the set of edges in $E - E_t$ with one endpoint in V_α and the other in V_β . Since edges are undirected, we shall understand $V_\beta \times V_\alpha$ to denote the same node as $V_\alpha \times V_\beta$. The root of the 2-dimensional topology tree is labelled $V \times V$ and represents the set of all edges in $E - E_t$. If a node in the 2-dimensional topology tree represents $V_\alpha \times V_\alpha$, where V_α has children $V_{\alpha_1}, V_{\alpha_2}, \dots, V_{\alpha_r}$ in the topology tree, then $V_\alpha \times V_\alpha$ has children $\{V_{\alpha_i} \times V_{\alpha_j} \mid 1 \leq i \leq j \leq r\}$. Similarly, if a node represents $V_\alpha \times V_\beta$, where $\alpha \neq \beta$ and V_β has children $V_{\beta_1}, V_{\beta_2}, \dots, V_{\beta_s}$ in the topology tree, then $V_\alpha \times V_\beta$ has children $\{V_{\alpha_i} \times V_{\beta_j} \mid 1 \leq i \leq r, 1 \leq j \leq s\}$. A portion of the 2-dimensional topology tree corresponding to the topology tree in Fig. 2 is given in Fig. 8. In our structure III, leaves of the 2-dimensional topology tree will store the edge sets E_{ij} , along with the minimum cost edge of each set. Internal nodes will have the minimum of the values of their children.

We discuss how to modify a 2-dimensional topology tree when its corresponding topology tree is modified. Each modification in the topology tree affects nodes along

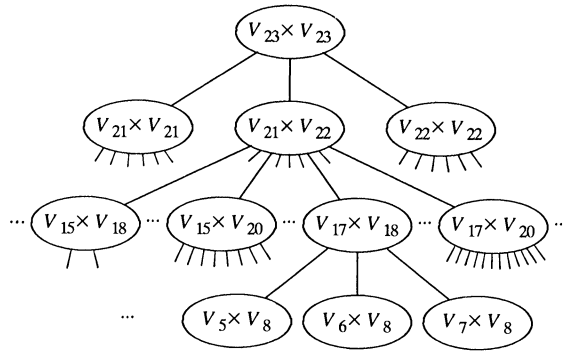


FIG. 8. A portion of the 2-dimensional topology tree corresponding to the topology tree in Fig. 3.

a path from the root to some node representing a vertex cluster V_α , which may or may not be a basic vertex cluster. In the corresponding 2-dimensional topology tree, nodes are affected along paths from the root to nodes of the form $V_\alpha \times V_\beta$ for all clusters V_β for which node $V_\alpha \times V_\beta$ exists. For any node V_γ on the path to V_α in the topology tree, all nodes of the form $V_\gamma \times V_\delta$ will be on these paths in the 2-dimensional topology tree. (It is straightforward to verify that for each node V_δ on the same level of the topology tree as V_γ , there will be a node $V_\gamma \times V_\delta$ in the 2-dimensional topology tree.) These nodes $V_\gamma \times V_\delta$ together form a subtree T_α of the 2-dimensional topology tree. In fact the subtree T_α will be isomorphic to that subtree of the topology tree with nodes at the same level as V_α or above. Hence there are $O(m/z)$ nodes in T_α .

Since each node has a number of children bounded by a constant, the time to modify or replace each node in the subtree T_α will be constant. Thus the operations of merging or splitting a 2-dimensional topology tree can be done in time proportional to the number of nodes in T_α , which is $O(m/z)$. To find a replacement edge, one must examine the values in appropriate nodes once the 2-dimensional topology tree has been split. Suppose that the topology tree has been split into two trees, whose vertex sets are the clusters V_α and V_β , with the V_β set having no fewer levels than the V_α set. The replacement edge can be found by examining the values at the nodes $V_\alpha \times V_\gamma$ for all such nodes, and taking the minimum. It will take $O(m/z)$ time to find and examine these nodes.

Choosing $z = \sqrt{m}$, we get our structure III.

THEOREM 4. *Structure III allows the on-line edge-update problem for minimum spanning trees to be solved in $O(\sqrt{m})$ time per update, using $O(m)$ space and $O(m)$ preprocessing time, aside from the time to find the initial minimum spanning tree.*

Proof. As before, splitting and merging the basic vertex sets will use $O(z)$ time. All other operations will take $O(m/z)$ time. \square

7. A data structure for planar graphs. As stated previously, we have been able to do much better when the underlying graph is planar. In this case we do not deal at all with basic vertex clusters, but merely use the multilevel partition. Thus we use a topology tree for the minimum spanning tree T , augmented with additional information. Consider an internal node of the topology tree representing vertex cluster W , whose children represent vertex clusters W_1, W_2, \dots, W_r . Since the graph is planar, it may be laid out so that each cluster W_i is in its own connected region of the plane. All edges between a pair of vertices in any one W_i may be laid out so that they are wholly contained in the appropriate region. An example of such a correspondence is shown

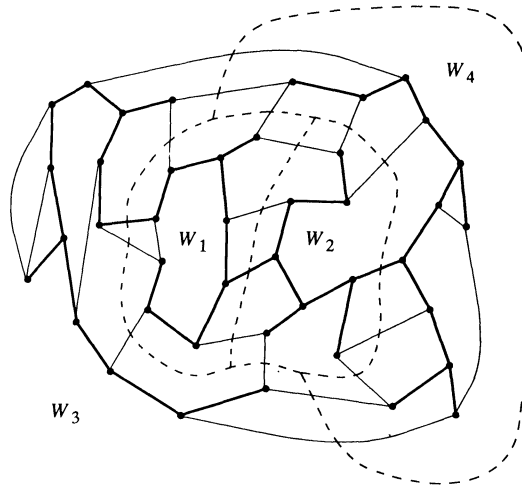


FIG. 9. A planar graph, with the tree edges shown in bold, the nontree edges in solid, and the vertices grouped with dashed lines.

in Fig. 9, with the tree edges shown as bold lines, the nontree edges as solid lines, and the boundary of the regions shown with dashed lines.

Given a planar embedding of the graph, consider any vertex cluster W_i and its region. The region is either simple, or it has between one and three "holes" in it. One such example is shown in Fig. 10a, in which region W_1 has a closed curve bounding it, which separates W_1 from W_2 , W_3 and W_4 . For each closed curve bounding a region W_i , the edges with one endpoint in W_i and the other not in W_i may be ordered in a natural way, e.g., clockwise around the closed curve. A *boundary* between two regions is a maximal set of edges between the regions, which is consecutive in ordering with respect to both regions. It is possible that two regions have more than one boundary between them. For example, note that in Fig. 10b the clusters W_3 and W_4 have two boundaries between them. For $r \geq 3$ regions, it is not hard to show that there are at most $3r - 6$ boundaries between them. Since no vertex cluster will have more than four children in the topology tree, there will be at most six boundaries between the children. Two such cases are shown in Figs. 9 and 10b.

The operations that we shall perform on boundaries are splitting a boundary, concatenating two boundaries, and finding a minimum-cost nontree edge in the boundary. We thus represent the boundaries with mergeable heaps, such as those in [AHU]. The merging and splitting of regions are similar to what has already been discussed with respect to topology trees, except now boundaries of regions must also be

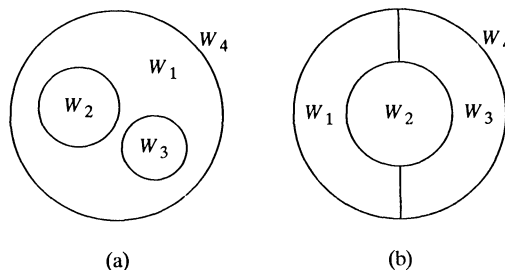


FIG. 10. Examples of relationships between regions.

maintained. We shall discuss the splitting of a region in some detail, leaving the simpler operation of merging to the reader.

We first consider how to split a vertex cluster W in the topology tree, assuming that edge e , a tree edge with endpoints in W , is removed. Our split routine will return two clusters W' and W'' , and boundary B between them. Let W be the union of subsets W_1, W_2, \dots, W_r . If e is in some boundary between a pair of the W_i 's, then do the following. Determine which of the W_i 's will still be connected to each other, i.e., which W_i 's will be in W' , and which will be in W'' . Determine those boundaries between W_i 's that will form the boundary between W' and W'' , and concatenate them together to form B . Each remaining boundary between the W_i 's will separate subclusters of either W' or W'' . Return W' , W'' and the boundary B between them.

As an example, consider the region W_1 from Fig. 9, shown by itself in Fig. 11a. Suppose it consists of two subregions, W_{11} with the upper four vertices, and W_{12} with the lower five vertices. Suppose that W_1 is to be split on the dashed edge between W_{11} and W_{12} . The resulting two regions $W'_1 = W_{11}$ and $W''_1 = W_{12}$ are shown in Fig. 11b, along with the boundary between them, shown as a dotted line.

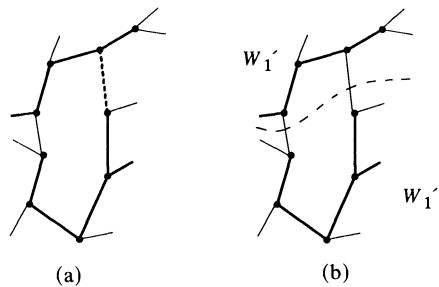


FIG. 11. Splitting a planar region.

If e is not in the boundary between a pair of the W_i 's, then it is contained in one of the W_i 's, say W_j . Recursively split W_j on edge e , which should return W'_j , W''_j and a boundary between them. Split as necessary the boundaries that W_j had with any of the other W_i 's. Determine which of the W_i 's, along with W'_j and W''_j , are connected, to form the basis of W' and W'' . If W'_j is of level one less, merge it with a neighbor. Handle W''_j similarly. Determine the boundaries between the new W_i 's that will form the boundary between W' and W'' . As before concatenate these boundaries, and assign remaining boundaries to W' and W'' .

Now suppose that the whole graph W in Fig. 9 is to be split on the same edge as in Fig. 11a. The boundary that W_1 shared with W_3 must be split, as well as the boundary that W_1 shared with W_2 . Note that W'_1 will be merged with W_3 , absorbing the boundary between them into the result W' , and W''_1 will be merged with W_2 and W_4 , yielding the result W'' . The boundary between W' and W'' will be the concatenation of the boundaries between W_3 and W_4 , W'_1 and W_4 , W'_1 and W_2 , W'_1 and W_1 , W_3 and W''_1 , and W_3 and W_2 , in that order.

The time to split vertex cluster V on edge e may be seen to be $O((\log m)^2)$. As established earlier, the height of the topology tree will be $O(\log m)$. For each level, the number of boundaries that will be split or concatenated will be no greater than some constant. Since each split and concatenation will take $O(\log m)$ time, this work is bounded by $O(\log m)$ per level. Merging vertex clusters together, as in a small W'_j with a larger neighbor, will require $O((h - h') \log m)$, where h' is the level of W'_j , and h is the level of its larger neighbor. The level of the resulting vertex cluster will be at

least h . Thus the total of the difference in levels will be $O(\log m)$. Thus the merging will be $O((\log m)^2)$ also.

THEOREM 5. *The edge-update spanning tree problem may be solved in $O((\log m)^2)$ time per update, using $O(m)$ space and $O(m)$ preprocessing time.*

8. Edge insertion and deletion, and maintaining connected components. It is not hard to cast the problems of edge insertion and deletion into an edge update framework. When an edge is inserted, the degree of the incident vertices in the original graph increases. If the degree of such a vertex has become four, then the transformation discussed in § 2 must be applied to the vertex. If the degree has become greater than four, then the transformation from § 2 has already been applied but now must be modified. In both cases, the number of new edges and vertices introduced is a small constant. Similar transformations may be performed in reverse if an edge is deleted.

When edges are being inserted or deleted, the number m of edges is of course changing. Let m_t be the number of edges in the graph at time t . We claim that an update at time t can be carried out in time $O(\sqrt{m_t})$. This can be achieved as follows. Let $z_t = \lceil \sqrt{m_t} \rceil$. We shall also allow basic vertex clusters of size $3z_t - 1$, and basic vertex clusters of external degree less than three of size $z_t - 1$. When the value of z changes due to an insertion or deletion, there will be at least $\sqrt{m_t}$ updates before z advances to the next value up to or down in the same direction. The idea is to adjust a small constant number of basic vertex clusters each time that there is a new update. Since there will be no more than $\sqrt{m_t}$ clusters that need to be adjusted, the adjustment may be accomplished before a new round of adjustments is initiated. Thus every time an insertion occurs, the clusters can be scanned to find any cluster that is too small and this cluster can be combined with a neighbor as necessary. Similar operations are performed upon a deletion.

THEOREM 6. *A minimum spanning tree may be maintained under the operations of insertion and deletions of edges in $O(\sqrt{m_t})$ time per update, where m_t is the current number of edges.*

If the graph is planar, then things are even easier, since no parameter z will be adjusted. Thus edge insertion and deletion can be performed in $O((\log m)^2)$ time, provided that the graph remains planar.

Using the above modifications to our basic structure, we can solve the problem of maintaining connected components of a graph on-line. Given a graph in which edges are being inserted and deleted, a data structure must be maintained so that a query about whether two vertices are in the same connected component can be answered in constant time. In addition to our above structure we use the following. Let each edge in the graph have cost 1. In addition, keep a sufficient number of "dummy" edges of cost 2 in the graph to link together the connected components. Any dummy edge included in the augmented graph must be in the minimum spanning tree of the graph.

Give each connected component a number. In each basic vertex cluster, maintain lists of vertices that are in the same connected component. An array *listnum* will give for each vertex v , the index of the list holding v . A second array *compnum* will give for each list l the index of the connected component containing the vertices in l . To answer a query on vertices u and v , compare *compnum* (*listnum* (u)) and *compnum* (*listnum* (v)) for equality.

To insert an edge (u, v) do the following. If the edge is currently a dummy edge, make it a real edge by decreasing its cost to 1. Otherwise insert it with cost 1. If u and v were in different components, merge the components by doing the following. First, identify the at most one basic vertex cluster that contains vertices from both components,

and concatenate the lists for these components, changing the *listnum* values of the vertices on one list to be the smaller of the two component numbers. Then in each basic vertex cluster containing a list in the higher numbered component, change the *compnum* value of the list to be the index of the lower numbered component. Since there are $O(\sqrt{m_t})$ vertices in any basic vertex cluster, and $O(\sqrt{m_t})$ basic vertex clusters altogether, the time required will be $O(\sqrt{m_t})$. Inserting edge (u, v) may force a dummy edge out of the minimum spanning tree. Delete this edge. This will require work proportional to the total size of a constant number of basic vertex clusters, or $O(\sqrt{m_t})$.

The ideas for deletion are similar. If the edge e to be deleted is not in the minimum spanning tree, delete it. If it is in the tree, and there is a replacement edge for e of cost 1, delete e . Otherwise, increase the cost of e from 1 to 2. Then renumber the component that has split off, split the at most one list in some basic vertex cluster that has vertices in both resulting components, and give the new number to the $O(\sqrt{m_t})$ lists (at most one per basic vertex cluster) containing vertices in the new component.

THEOREM 7. *The on-line connected components problem can be solved using data structures that allow edge insertion and deletion times of $O(\sqrt{m_t})$.*

9. Generating the K smallest spanning trees. In this section we show how to use our data structures to generate the K smallest spanning trees of a graph in increasing order of cost. Each tree in the sequence except for the first can be described in terms of a preceding tree in the sequence, with one tree edge swapped out and replaced by a nontree edge. Thus our output will be in the following form. The minimum spanning tree will be output first, followed by a succinct description of each of the remaining trees. Each remaining tree will be characterized by its cost, a reference to the tree from which it can be derived using a single swap, and that swap.

Our approach is based on a branch-and-bound technique described in [L] and used in [G], [KIM]. The set of all spanning trees not yet selected is partitioned on the basis of the inclusion or exclusion of certain edges. When the minimum spanning tree is selected, the set is partitioned as follows. For each edge e_i in the minimum spanning tree, there is a replacement edge f_i of minimum cost. Without loss of generality, assume the swap pairs (e_i, f_i) are indexed in increasing order of $c(f_i) - c(e_i)$. We assume that all such costs are unique, with ties broken by lexicography, if necessary. The set of remaining spanning trees is partitioned into $n - 1$ subsets with the i th subset containing all spanning trees with edge e_i excluded and $\{e_1, \dots, e_{i-1}\}$ included.

When the next smallest spanning tree T' is chosen from one of these subsets, the remainder of the subset is partitioned as follows. Let $\{e'_i \mid i = 1, \dots, n' - 1\}$ be the set of edges in tree T' that are not required to be included in T' because of membership in the subset, and $\{f'_i\}$ the corresponding set of replacement edges of minimum cost that are not required to be excluded from spanning trees in that subset. Once again assume that the pairs (e'_i, f'_i) are in order of increasing cost. The subset is partitioned into $n' - 1$ subsets with the i th subset containing all spanning trees satisfying the previous conditions plus e'_i excluded and $\{e'_1, \dots, e'_{i-1}\}$ included.

The above discussion seems to imply that every time the next smallest spanning tree is chosen, a large number of replacement edges must be found. However, the determination of some replacement edges may be delayed, as the next lemma suggests. This will allow us to realize our improved strategy when K is small.

LEMMA 4. *Let T be a minimum spanning tree of graph (V, E) . Let f_i and f_j be the replacement edges for e_i and e_j , resp., in T , and let \hat{f} be the replacement edge for e_j in the minimum spanning tree $T - e_i + f_i$ in $(V, E - e_i)$. If $c(f_i) - c(e_i) < c(f_j) - c(e_j)$, then*

$$c(T - e_i + f_i - e_j + \hat{f}) > c(T - e_j + f_j).$$

Proof. Since T is a minimum spanning tree, $c(f_i) > c(e_i)$. Thus the proof reduces to showing that $c(\hat{f}) \geq c(f_j)$. If $\hat{f} = f_j$, then we are done. Otherwise, e_j must be on the cycle induced by f_i in T . Since e_j has a replacement edge of f_j in T chosen among edges including f_i , $c(f_i) \geq c(f_j)$. Since $\hat{f} \neq f_j$, $T - e_i + \hat{f}$ is a spanning tree. Since e_i has replacement edge f_i chosen instead of \hat{f} , $c(\hat{f}) > c(f_i)$. The lemma then follows. \square

Our approach is as follows. First use a fast algorithm to find the minimum spanning tree T_1 . Generate our data structure for T_1 . For each tree edge, find its replacement edge, using the algorithm in [T2]. Each such swap infers a spanning tree. Name each spanning tree, label it with a reference to T_1 and the swap that generates it. Create a heap on the costs of these spanning trees. Set up a list L_1 of such trees resulting from T_1 that have already been chosen. Initially, L_1 is empty.

We then iterate the following step until $K - 1$ additional spanning trees have been chosen. Suppose $i - 1$ trees have already been chosen. Select the minimum value out of the heap. The corresponding spanning tree will be T_i . Let T_j be the spanning tree from which T_i is generated, e_i the edge removed, and f_i the replacement edge. Generate our data structure for T_i from that of T_j , setting the cost of e_i in the graph to be ∞ . Traverse the list L_j . For each T_l on the list, determine the replacement edge for e_i in T_l . Name the corresponding spanning tree, label it with T_l and the new swap, and enter its cost into the heap. Now add T_i to the list L_j . Find the replacement edge for f_i in T_i . Name the new spanning tree, and as before label it and enter its cost into the heap. Set up a list L_i , initially empty. Repeat until $i = K$.

The correctness of the above algorithm may be seen as follows. The inclusion-exclusion strategy is being implemented, with an edge excluded by setting its cost to ∞ in the data structure that is the source of the appropriate subset of spanning trees. Inclusion of edges is enforced by the mechanism of building and traversing the lists $\{L_j\}$. The only edges in tree T_i that can be swapped out are f_i and those edges involved in swaps with respect to T_j that are more expensive than (e_i, f_i) . Lemma 4 guarantees that a spanning tree arising from such a swap need not be examined until the corresponding list has been traversed during execution of the algorithm.

We now consider the time complexity of our algorithm. Finding the minimum spanning tree requires $O(m \log \log_{(2+m/n)} n)$ time. The time to find all replacement edges in the minimum spanning tree is $O(m\alpha(m, n))$, which is dominated by the above time. We bound the iteration time as follows. Every time an iteration is performed, the length of some list is increased by one. The total number of elements on all lists when tree T_i is chosen is $i - 2$. Thus at most $i - 1$ replacement edges in various trees must be found after selecting tree T_i . Since the time to find a replacement edge is $O(\sqrt{m})$, the i th iteration requires $O(i\sqrt{m})$ time. For $K - 1$ iterations, this time is $O(K^2\sqrt{m})$. When K is $o(\sqrt{m})$ the resulting time is $o(Km)$.

As presented, the time and space to generate the T_i 's is $O(Km)$, since the space for our basic data structure is $O(m)$. However it is possible to save space in the following way. Since the time required to generate our data structure for T_i by modifying the structure for T_j is $O(\sqrt{m})$, the number of new nodes is $O(\sqrt{m})$. The idea is to not destroy any nodes of the structure for T_j , but simply share the appropriate subtrees. This reduces the space to $O(m + K\sqrt{m})$, which is $O(m)$ when K is $O(\sqrt{m})$.

THEOREM 8. *The K smallest spanning trees of a graph can be found in $O(m \log \log_{(2+m/n)} n + K^2\sqrt{m})$ time and $O(m + K\sqrt{m})$ space. \square*

As discussed in the introduction, this result is better than previous results in [KIM] whenever K is $o(\sqrt{m})$ and $\omega(\log \log_{(2+m/n)} n)$. Our results are better than those in [H12] whenever $K\sqrt{m}$ is $o(n(\log n)^2)$ or K is $o(m^{1/4}(\log n)^{1/2})$. If the given graph is planar, then our corresponding structures for planar graphs should be used. These

results are better than the corresponding ones for [KIM] whenever K is $o(n/(\log n)^2)$. They are also never worse than those in [HI2], and are better when K is $o(n)$.

THEOREM 9. *The K smallest spanning trees of a planar graph can be found in $O(n + K^2(\log n)^2)$ time and $O(n + K(\log n)^2)$ space. \square*

REFERENCES

- [AHU] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
- [CT] D. CHERITON AND R. TARJAN, *Finding minimum spanning trees*, this Journal, 5 (1976), pp. 724-742.
- [CH] F. CHIN AND D. HOUCK, *Algorithms for updating minimum spanning trees*, J. Comput. System. Sci., 16 (1978), pp. 333-344.
- [ES] S. EVEN AND Y. SHILOACH, *An on-line edge deletion problem*, J. Assoc. Comput. Mach., 28 (1981), pp. 1-4.
- [F] G. N. FREDERICKSON, *Data structures for on-line updating of minimum spanning trees*, Proc. 15th ACM Symposium on Theory of Computing, Boston, April 1983, pp. 252-257.
- [FS1] G. N. FREDERICKSON AND M. A. SRINIVAS, *On-line updating of degree-constrained minimum spanning trees*, Proc. 22nd Allerton Conference on Communication, Control, and Computing, October 1984, to appear.
- [FS2] ———, *Data structures for on-line updating of matroid intersection solutions*, Proc. 16th ACM Symposium on Theory of Computing, Washington, DC, April 1984, pp. 383-390.
- [G] H. N. GABOW, *Two algorithms for generating weighted spanning trees in order*, SIAM J. Computing, 6 (1977), pp. 139-150.
- [HI1] DOV HAREL, *On line maintenance of the connected components of dynamic graphs*, manuscript, 1982.
- [HI2] ———, *personal communication*, 1983.
- [Hy] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.
- [KIM] N. KATOH, T. IBARAKI AND H. MINE, *An algorithm for finding K minimum spanning trees*, this Journal, 10 (1981), pp. 247-255.
- [L] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.
- [OV] M. H. OVERMARS AND J. VAN LEEUWEN, *Maintenance of configurations in the plane*, J. Comput. System. Sci., 23 (1981), pp. 166-204.
- [ST] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System. Sci., 26 (1983), 362-391.
- [SP] P. M. SPIRA AND A. PAN, *On finding and updating spanning trees and shortest paths*, this Journal, 4 (1975), pp. 375-380.
- [T1] R. E. TARJAN, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215-225.
- [T2] ———, *Applications of path compression on balanced trees*, J. Assoc. Comput. Mach., 26 (1979), pp. 690-715.
- [W] D. J. A. WELSH, *Matroid Theory*, Academic Press, New York, 1976.
- [WL] D. E. WILLARD AND G. LUEKER, *A transformation for adding range restriction capability to data structures*, J. Assoc. Comput. Mach., to appear.
- [Y] A. C. YAO, *An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees*, Inform. Proc. Lett., 4 (1975), pp. 21-23.

DECOMPOSING A POLYGON INTO SIMPLER COMPONENTS*

J. MARK KEIL†

Abstract. The problem of decomposing a polygon into simpler components is of interest in fields such as computational geometry, syntactic pattern recognition, and graphics. In this paper we consider decompositions which do not introduce Steiner points. The simpler components we consider are convex polygons, spiral polygons, star-shaped polygons and monotone polygons. We apply a technique for improving the efficiency of dynamic programming algorithms in order to achieve polynomial time algorithms for the problems of decomposing a simple polygon into the minimum number of each of the component types. Using the same technique we are able to exhibit polynomial time algorithms for the problems of decomposing a simple polygon into each of the component types while minimizing the length of the internal edges used to form the decomposition. When the polygons are allowed to contain holes many of the problems become NP-hard.

Key words. polygon decomposition, convex polygon, star-shaped polygon, spiral polygon, monotone polygon, dynamic programming

1. Introduction. Let P be a simple polygon in the plane having vertices v_1, v_2, \dots, v_n clockwise on its boundary. There are many ways to decompose such a polygon into simpler components [15], [26]. The simpler components we will consider are convex polygons, spiral polygons, star-shaped polygons and monotone polygons. Using a dynamic programming approach we will develop polynomial time algorithms for decomposing a polygon into the minimum number of each of these simple components. Using the same technique we are able to exhibit polynomial time algorithms for the problems of decomposing a simple polygon into each of the component types while minimizing the length of the internal edges used to form the decomposition.

There are several motivations for considering polygon decomposition problems. An arbitrary polygonal shape can be recognized more easily once its component parts have been identified. This property can be used in pattern recognition schemes [9], [23], [26]. In computational geometry, a problem can often be solved on a general polygon by applying efficient specialized algorithms to the component parts of the decomposed polygon. Other application areas for polygonal decompositions include database systems [20] and graphics [21].

Decomposing a simple polygon into nonoverlapping component parts can be done with or without introducing additional vertices which are commonly called Steiner points. A decomposition which does not increase the number of vertices is preferable when the components are to be processed further. Figure 1.1 shows decompositions of simple polygons into the minimum number of convex polygons with and without Steiner points. The algorithms we describe in this paper do not introduce Steiner points.

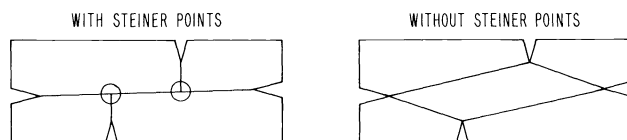


FIG. 1.1. Minimum decompositions.

* Received by the editors October 1, 1982, and in final revised form March 20, 1984.

† Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A7. This work was supported by the Natural Sciences and Engineering Research Council of Canada.

If the interior angle at a vertex is reflex the vertex is called a notch. Experimental observation from graphics [21] and pattern recognition [23] shows that, in practice, the number of notches in a polygon is much smaller than the number of vertices. We let N denote the number of notches in a polygon and we describe and analyze decomposition algorithms with respect to both n and N .

Before we consider the polygon decomposition problems we make the following definitions. A *Convex Polygon* P is a simple polygon in which any two boundary points can be joined by a segment that lies completely within P . A *Spiral Polygon* is a simple polygon whose boundary chain contains at most one concave subchain. A *Monotone Polygon* is a simple polygon in which there exists two extreme vertices in a preferred direction such that they are connected by two polygonal chains monotonic in the preferred direction. A *Star-shaped Polygon* is a simple polygon in which the entire polygon is visible from at least one fixed (possibly interior) point of the polygon.

In § 2 we develop our dynamic programming approach. We use it in § 3 to develop a polynomial time algorithm for the problem of decomposing a simple polygon into the minimum number of convex components. In § 4 we develop a polynomial time algorithm for decomposing a simple polygon into the minimum number of star-shaped components. We consider minimum edge length decompositions in § 5 and other decomposition problems in § 6.

2. Dynamic programming. Since their introduction by Bellman [2], dynamic programming (DP) algorithms have been used to solve a wide variety of discrete optimization problems. To apply DP one must represent such a problem by a decision process which proceeds from state to state in a series of stages. At each stage a decision is made that will lead to a state in the next stage at a certain cost. A DP algorithm decomposes the problem into a number of smaller subproblems each having fewer stages than the original problem and it gains its efficiency by avoiding recomputing solutions to common subproblems. For a survey of the use of DP in computer science see [3].

A state consists of variables that describe the condition of the system. A state must contain enough information so that current decisions depend only on the current state and not on the particular history of previous states and decisions. This property is called the state separation property. The state space consists of the set of all possible states in which the system may exist and a valid state space for DP must have the state separation property.

A problem for DP is rarely presented in terms of states and decisions and often such a representation is not obvious. The most difficult part in applying DP is usually the definition of the state space. If too little information is included in a state the state space will not be valid and no correct DP algorithm can be developed. If too much information is included in a state the state space will be large and the DP algorithm will produce an unnecessary amount of computation.

Reducing the size of a valid state space can reduce the time required by a DP algorithm. Karp and Held [13] suggested state space minimization as a good heuristic for speeding up DP algorithms but were unable to provide a method of doing it. Later Ibaraki [12] showed that in general the problem of minimizing the number of states in a valid state space is not decidable.

Elmaghraby [8] introduced the concept of equivalent states which is useful in reducing the size of a valid state space. A policy is a finite ordered sequence of decisions that leads from one state through others to a destination state. Let X be the set of all policies. Then two states s_1 and s_2 are said to be equivalent if and only if the state

reached from s_1 after policy x is the same state reached from s_2 after policy x for all permissible policies $x \in X$. In a minimum state space there will be no pair of equivalent states.

To reduce the size of a valid state space Elmaghraby [8] suggests constructing classes of equivalent states. Once these classes have been constructed only one representative member from each class need be kept. This method has not been widely used because it is difficult to find these classes of equivalent states. In this paper we systematically use the equivalence class method of state reduction to exhibit polynomial time algorithms for each of the decomposition problems we consider.

Let us look at the problem of decomposing a polygon into the minimum number of convex polygons and attempt to develop a DP algorithm for it. To do this we need some definitions which are illustrated in Fig. 2.1. Two points of a polygon P are said to be *visible* if the line segment joining them lies wholly inside P . A *subpolygon* P_{ij} of P with vertices v_i, v_{i+1}, \dots, v_j exists when $1 \leq i < j - 1 \leq n - 1$, and $v_i v_j$ are visible. The *base convex polygon* C_{ij} of a minimum decomposition (MD) D of P_{ij} is that convex polygon of D that contains the edge $v_i v_j$ in its boundary. We call $v_i v_j$ the *base edge* of P_{ij} . A triangle T_{imj} can *merge* with a convex decomposition A of P_{im} if $C_{im}(A) \cup T_{imj}$ is a convex polygon. The *size* of a MD of P_{ij} is the number $|D|$ of convex polygons in a MD of P_{ij} .

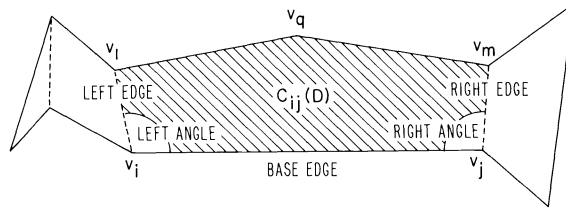


FIG. 2.1. Definitions. T_{imj} merges left with any MD A of P_{im} with $C_{im}(D) = v_i v_l v_q v_m$.

Let us define the state space by letting the states be of the form s_{ij} where s_{ij} has the interpretation that subpolygon P_{ij} has been decomposed minimally. A decision is then based on a pair of states s_{im} and s_{mj} , $i < m < j$, and leads to the state s_{ij} in the next stage. The minimum cost of a policy that leads to s_{ij} is equal to the size of MD of P_{ij} . A MD of subpolygon P_{ij} is found by taking a MD of P_{im} and P_{mj} for some $i < m < j$ and merging T_{imj} with C_{im} or C_{mj} if possible.

An attempt to use this DP formulation will reveal that the state space is not valid. There can be many MDs of a subpolygon P_{im} and it is not sufficient for a state to contain information about only one of them. Figure 2.2 shows two MDs of P_{im} only one of which merges with T_{imj} .

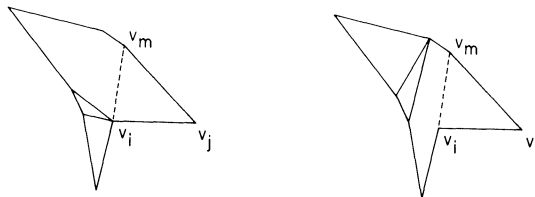


FIG. 2.2. The MD of P_{im} shown on the left merges with T_{imj} while the MD of P_{im} shown on the right does not.

Since storing information about one MD of a subpolygon P_{ij} is insufficient we can try creating a state for each MD of a subpolygon. With this definition the problem encountered due to lack of information has vanished and indeed the state space is valid. We therefore have a correct DP formulation and the only question remaining is the efficiency of the algorithm. Unfortunately, the state space we have defined is very large. In fact, as illustrated in Fig. 2.3 there can be an exponential number of ways of decomposing a simple polygon into convex polygons. Since we are seeking a polynomial time algorithm we must try to reduce the size of the state space.

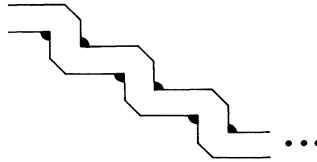


FIG. 2.3. There are $2^{n/3-4}$ MDs of the above polygon. Each notch has an independent choice of two interior edges that will remove it.

As indicated previously we will use equivalence classes of states. To describe these we make a few more definitions which are illustrated in Fig. 2.1. In a subpolygon P_{ij} the edge preceding $v_i v_j$ in the clockwise representation of $C_{ij}(D)$ is the *right edge* of MD D . The *left edge* of a MD is defined analogously. The *left angle* of a MD D is the angle between the left edge and the base edge of D . The *right angle* of a MD is defined analogously.

The reason why we needed to keep more than one MD of a subpolygon P_{im} was that the MDs varied in their ability to merge with triangles T_{imj} . But it is only the left and right edges of a MD that affect its ability to merge. This suggests that all MDs with a given pair of left and right edges are equivalent under this DP formulation and this equivalence is established in the following lemma.

LEMMA 2.1. *Any members of the class of MDs of a subpolygon P_{im} with a given pair of left and right edges are equivalent in terms of constructing a MD of a valid subpolygon $P_{ij}(j > m)$ by merging with T_{imj} .*

Proof. Let A be a MD of P_{im} . T_{imj} can merge left with A if $C_{im}(A) \cup T_{imj}$ is a convex polygon. Since $C_{im}(A)$ and T_{imj} are convex $C_{im}(A) \cup T_{imj}$ is convex if and only if the angles between $v_i v_j$ and the left edge of A and between $v_m v_j$ and the right edge of A are less than 180 degrees. Therefore two MDs of P_{im} with the same left edge and right edge are equivalent in terms of constructing a MD of a valid subpolygon $P_{ij}(j > m)$.

We can therefore reduce the size of the state space by including only one representative from each class of MDs of a subpolygon with a given pair of left and right edges. We have retained enough information to perform merges and the state space remains valid. Since there are only $O(n^2)$ possible pairs of left and right edges of MDs in a subpolygon there are now only a polynomial number of states in the state space and our DP algorithm runs in polynomial time. However we are not yet finished eliminating states and improving the efficiency of the algorithm.

Even in the reduced state space there are states that will not be used in constructing the optimal solution. The following lemma helps identify some of them.

LEMMA 2.2. *No MD D of P contains an interior edge which connects two vertices of P which are not notches.*

Proof. Straightforward.

This means we need not consider states associated with subpolygons P_{ij} where neither v_i nor v_j is a notch. A subpolygon is called a *valid subpolygon* when either $i = 1$

or $j = n$ or at least one of v_i and v_j is a notch. In the rest of this section and in § 3 when we say subpolygon we will mean valid subpolygon. We can remove those states that are not associated with valid subpolygons. The following lemma will enable us to identify classes of states that can be replaced in a valid state space by one representative state.

LEMMA 2.3. *If MD A of P_{im} has left angle ϕ_1 and right angle θ_1 and MD B of P_{im} has left angle ϕ_2 and right angle θ_2 where $\phi_1 \leq \phi_2$ and $\theta_1 \leq \theta_2$, then MD B merging with T_{imj} where P_{ij} is a valid subpolygon ($j > m$), implies that A also merges with T_{imj} .*

Proof. Straightforward.

If two states are associated with the MDs A and B as in Lemma 2.3 the state associated with MD B need not be kept in the state space. To find the MDs that can be similarly eliminated we first define an equivalence relation R over the set of MDs of a subpolygon P_{ij} so that all those MDs of P_{ij} with right angle θ will belong to the equivalence class $R(\theta)$ of R . A MD in $R(\theta)$ with the minimum left angle is called a *left minimum* of $R(\theta)$. Using Lemma 2.3 we now seek the smallest set of MDs of P_{ij} that will be sufficient to represent the right angle equivalence classes. To begin with, Lemma 2.3 implies that any MD that is not a left minimum of one of the classes $R(\theta)$ is not essential. Normally a class of MDs, $R(\theta)$, will be represented by its left minimum. However if two left minimum MDs A and B have angles as in Lemma 2.3 then MD A is used to represent both $R(\theta_1)$ and $R(\theta_2)$. A set of MDs with only these essential right angle class representatives is said to have the *right representative* (RR) property.

In the algorithm a set X with the RR property is stored in a binary search tree so that a MD with a given right angle can be found in $O(\log(\text{size}(X)))$ time. The definition of the RR property implies that a set X with the RR property sorted in increasing order of right angle will also be sorted in decreasing order of left angle. The *left representative* (LR) property is defined analogously.

By identifying equivalent classes of states and eliminating unnecessary states we have substantially reduced the size of the state space. In the next section we will describe the DP algorithm in more detail.

3. Convex decomposition algorithm. Of all the polygon decomposition problems, the problem of decomposing a polygon into the minimum number of convex components has received the most attention. Chazelle and Dobkin [4], [5], [6] were the first to exhibit a polynomial time algorithm for a minimum polygonal decomposition problem. Their $O(n + N^3)$ time algorithm decomposes a polygon into convex parts and allows Steiner points.

Until recently, when Steiner points are disallowed, no polynomial time exact solution was known to the convex decomposition problem thus motivating the development of approximation algorithms. Feng and Pavlidis [9] describe an $O(N^3n)$ time algorithm which does not generally yield a minimum decomposition. Schachter's [25] $O(Nn)$ time decomposition algorithm, based on the Delaunay triangulation, also does not generally yield a minimum decomposition. Recently, Greene [10] has developed an $O(n \log n)$ time algorithm that finds a decomposition that is within 4 times of the minimum decomposition. Note however that any convex decomposition that does not contain unnecessary edges will be within 4 times of the minimum decomposition.

Recently, Greene [10] independently discovered an $O(N^2n^2)$ time exact algorithm for the convex decomposition problem. Our algorithm for that problem runs in $O(N^2n \log n)$ time.

In the previous section we defined the state space for a DP formulation of the convex decomposition problem. In order to complete the algorithm description we

make a few definitions. A *reference vertex* is a vertex of P which is either a notch or v_1 or v_n . Nonreference vertices are called *convex vertices*. A valid subpolygon P_{ij} will be one of three types: *type (a)* if both v_i and v_j are reference vertices, *type (b)* if v_i is a reference vertex and v_j is a convex vertex, or *type (c)* if v_i is a convex vertex and v_j is a reference vertex. A *base triangle* T_{imj} of P_{ij} is a triangle $v_i v_m v_j$ ($i < m < j$) where each of $v_i v_m$ and $v_j v_m$ is either a base edge of a valid subpolygon or an original side of P .

Preprocessing. Some of the work is best done before the main DP procedure begins.

1. Determine which vertices are notches in $O(n)$ time.
2. For each reference vertex x of P determine the set of vertices of P visible from x . For convex vertices store the set of visible reference vertices. Denote such a visibility list sorted by angle about vertex x as $V(x)$. All of this can be done in $O(nN)$ time using the visibility polygon algorithm of El-Gindy and Avis [7].
3. Since each visibility pair $v_i v_j$ ($i < j$) determined in step 2 is the base edge of a valid subpolygon these valid subpolygons can be sorted in ascending order by the size measure $j - i$ in time $O(nN \log(nN))$.

4. For each valid subpolygon P_{ij} form the set of base triangles. In the $O(N^2)$ subpolygons of type (a) this can be done in $O(n)$ time by computing $T = \{(V_i \cup \{v_{i+1}\}) \cap (V_j \cup \{v_{j-1}\})\}$. In the $O(nN)$ subpolygons of type (b) or (c) this can be done in $O(N \log n)$ time by computing T using a binary search.

DP procedure.

1. Consider each valid subpolygon P_{ij} in the order computed in step 3 of the preprocessing.

2. Compute a set XR_{ij} of MDs of P_{ij} with the RR property (and similarly a set XL_{ij} of MDs of P_{ij} with the LR property) as follows.

Let M denote the size of a MD of P_{ij} . Initially set M to n . If $j - i = 2$, XR_{ij} will contain a single triangle.

Otherwise, for each base triangle T_{imj} of P_{ij} :

(a) select the MD A from the set XL_{im} of P_{im} with the minimum left angle such that T_{imj} can merge left with $C_{im}(A)$. This is done by binary search. If no such A exists select any MD A of P_{im} . Take the decomposition A selected together with T_{imj} (merged if possible) and a MD B of P_{mj} to form a decomposition D of P_{ij} with right angle $(v_j v_m, v_j v_i)$. If $|D| > M$ discard D as it is not a member of XR_{ij} . If $|D| = M$ then insert D into XR_{ij} as the member with right angle $(v_j v_m, v_j v_i)$. If $|D| < M$ then empty XR_{ij} and insert D as the only member of XR_{ij} and reset M to $|D|$.

(b) select the MD B' from the set XR_{mj} with the minimum right angle that will merge with T_{imj} . Take decomposition B' merged with T_{imj} together with a MD A' of P_{im} to form a decomposition D' of P_{ij} . As in part (a) compare $|D'|$ to M to see if D' belongs in XR_{ij} .

At this point XR_{ij} will consist of a set of MDs with unique right angles. However XR_{ij} may still not have the RR property. To remove MDs from XR_{ij} that are inconsistent with the RR property begin by sorting XR_{ij} by right angle. Set ϕ to the left angle of the MD in XR_{ij} with the smallest right angle. Now scan XR_{ij} in order of increasing right angle. If the MD D under scan has left angle $\phi' \cong \phi$ delete D from XR_{ij} . If D has left angle ϕ' smaller than ϕ then set ϕ to ϕ' . When the scan is complete XR_{ij} has the RR property. See Fig. 3.1.

3. The subpolygon $P_{1n} = P$ will be the last one considered. Select one member of XL_{1n} or XR_{1n} as a MD of P .

Proof of correctness. We need to show that we correctly compute a set of MDs, with the RR(LR) property, for each subpolygon P_{ij} . We use a proof based on an

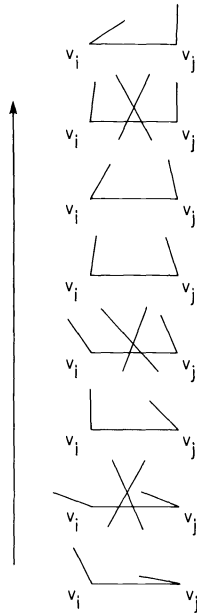


FIG. 3.1. A set of MDs with the RR property results from the scan that removes the crossed MDs.

induction on the size, $j - i$, of the subpolygons. If $j - i = 2$, P_{ij} is a triangle and there is only one MD. Otherwise, as an example, let us consider how we compute a MD of a subpolygon P_{ij} of type (a) with right side $v_j v_i$. Clearly we need only consider single merges to the left if we use base triangle T_{irj} . If any MD, say A , of P_{ir} merges with T_{irj} then clearly A merged with T_{irj} taken together with a MD of P_{rj} will form a MD of P_{ij} . If no MD of P_{ir} merges with T_{irj} then we may take any MD of P_{ir} together with T_{irj} and a MD of P_{mj} to form a MD of P_{ij} . We are similarly able to compute any MD that we require by single merging at an appropriate base triangle using MDs of smaller subpolygons that have already been calculated. The following lemma shows how an arbitrary MS, as a member of a set with the RR property, can be constructed.

LEMMA 3.1. *Let D be a MD in the set XR_{ij} of MDs of P_{ij} with the RR property. Furthermore when a base triangle T_{imj} exists let XL_{im} be a set of MDs of P_{im} with the LR property and XR_{mj} be a set of MDs of P_{mj} with RR property. Then there exists an $O(\log n)$ time algorithm which will construct a MD D' of P_{ij} equivalent to D using only $v_i v_l$ (the left side of D), $v_j v_r$ (the right side of D) and XL_{im} and XR_{mj} where T_{imj} is a base triangle of P_{ij} with either $v_m = v_l$ or $v_m = v_r$.*

Proof.

Case 1. If P_{ij} is of type (a) or (b) then T_{imj} exists so that $v_m = v_r$. The decomposition D' required can be found by computing decomposition D' in parts (i) and (ii) and selecting the smaller.

(i) If $C_{ij}(D)$ is a triangle then the MD D' formed by taking T_{imj} together with any MD A of P_{im} and any MD B of P_{mj} will have the same left and right edges as decomposition D and will be, by Lemma 2.1, equivalent to D .

(ii) If $C_{ij}(D)$ is not a triangle let polygon Q be $P_{im} \cup T_{imj}$. Clearly D restricted to Q is a MD of Q . Let A be D restricted to P_{im} . If A is not a MD of P_{im} then any MD of P_{im} , A'' , would have the property that $C_{im}(A'') \cup T_{imj}$ is not a convex polygon. But this would imply that there exists a MD of P_{ij} with the same right angle as D but with a smaller left angle. This is a contradiction to XR_{ij} having the RR property. Since

A is a MD of P_{im} when we select the MD A' from XL_{im} in $O(\log n)$ time with the smallest left angle that will merge with T_{imj} , A' will have the same left edge as A by Lemma 2.1. Taking T_{imj} merged with A' together with a MD B of P_{mj} will yield a MD D' of P_{ij} equivalent to D .

Case 2. If P_{ij} is of type (c) then T_{imj} exists so that $v_m = v_i$. Again the decomposition D' required is found by computing decomposition D' in parts (i) and (ii) and selecting the smaller.

(i) If $C_{ij}(D)$ is a triangle the MD D' can be found as in Case 1.

(ii) If $C_{ij}(D)$ is not a triangle in $O(\log n)$ time select a MD B' from XR_{ij} with the minimum right angle that will merge with T_{imj} . The MD D' of P_{ij} formed by taking a MD A of P_{im} together with T_{imj} merged with the MD B' of P_{mj} will have the same left angle as D and at least as small a right angle as D . Since D is in XR_{ij} with the RR property the right edge of D' will in fact equal the right edge of D and again by Lemma 2.2 D' will be equivalent to D .

The next lemma shows how the required set of MDs for each subpolygon can be found.

LEMMA 3.2. *A set XR_{ij} (likewise XL_{ij}) of MDs of P_{ij} with the RR (likewise LR) property can be correctly constructed in $O(n \log n)$ time if P_{ij} is of type (a) and in $O(N \log n)$ time if P_{ij} is of type (b) or (c). The procedure uses only a set XL_{im} of MDs of P_{im} with the LR property and a set XR_{mj} of MDs of P_{mj} with the RR property, where m is such that T_{imj} is a base triangle of P_{ij} .*

Proof. Each member decomposition D of XR_{ij} can be found using a different base triangle T_{imj} as in Lemma 3.1. If P_{ij} is of type (a) then $O(\log n)$ work is performed at each of the $O(n)$ base triangles. If P_{ij} is of type (b) or (c) then $O(\log n)$ work is performed at each of the $O(N)$ base triangles. With this method some MDs will be placed into XR_{ij} that are not consistent with the RR property. If we treat XR_{ij} as a set of two-dimensional vectors with first component (180° —right angle) and second component (180° —left angle) then the members of XR_{ij} that are consistent with the RR property will be the maximal elements of XR_{ij} with respect to the natural partial order. The scan procedure described in the algorithm implements the algorithm of Kung, Luccio and Preparata [16] for finding the maxima of a set of vectors. This requires $O(n \log n)$ time if P_{ij} is of type (a) and $O(N \log N)$ time if P_{ij} is of type (b) or (c).

Using the above lemmas we can prove the following.

THEOREM 3.3. *The algorithm finds a MD of a simple polygon P in $O(N^2 n \log n)$ time in the worst case.*

Proof. The preprocessing requires $O(N^2 n \log n)$ time. There are $O(N^2)$ subpolygons of type (a). There are $O(nN)$ subpolygons of type (b) or (c). To calculate a set XL_{ij} of MDs of P_{ij} with the LR property and a set XR_{ij} of MDs of P_{ij} with the RR property, as in Lemma 3.2, $O(n \log n)$ time must be spent at subpolygons of type (a) and $O(N \log n)$ time must be spent at subpolygons of type (b) or (c). Altogether $O(N^2 n \log n)$ time is required in the worst case to compute XR_{1n} and XL_{1n} . Since $P_{1n} = P$ any MD in XR_{1n} or XL_{1n} is a MD of P .

4. Star-shaped decompositions. Recall that a star-shaped polygon is a simple polygon in which the entire polygon is visible from at least one (possibly interior) fixed point of the polygon. Avis and Toussaint [1] give an $O(n \log n)$ algorithm that finds a decomposition into at most $n/3$ components and does not use Steiner points. However their algorithm does not generally yield a minimum decomposition. Before we can formulate a DP algorithm for the minimization problem we need a few definitions.

The set of points from which the entire polygon P is visible is called the *kernel* of P . For star-shaped decompositions we need to define two types of merging. Let $S_{im}(A)$ be the base star-shaped polygon of the minimum star-shaped decomposition A of P_{im} . A triangle T_{imj} can *single-merge* with MD A of P_{im} if $S_{im}(A) \cup T_{imj}$ is a star-shaped polygon. A triangle T_{imj} can *double-merge* with a MD A of P_{im} and a MD B of P_{mj} if $S_{im}(A) \cup T_{imj} \cup S_{mj}(B)$ is a star-shaped polygon.

To formulate a DP algorithm we define the state space by letting the states s_{ij} have the interpretation that subpolygon P_{ij} has been decomposed into the minimum number of star-shaped components. As in the convex case, decisions are based on a pair of states s_{im} and s_{mj} , $i < m < j$, and lead to the state s_{ij} in the next stage. A MD of P_{ij} is found by taking a MD of P_{im} and P_{mj} for some m such that $i < m < j$ and determining whether T_{imj} will double-merge or single-merge with S_{im} or S_{mj} . The size of a MD of P_{ij} can be one less, equal to, or one more than the sum of the sizes of a MD of P_{im} and P_{mj} for some m depending on which merges can take place. After our experience with convex decompositions we are not surprised that the state space in this DP formulation is not valid. As before it is not sufficient for the states to contain information about only one MD of a subpolygon P_{ij} . Figure 4.1 shows two star-shaped MDs of P_{im} only one of which single-merges with T_{imj} .

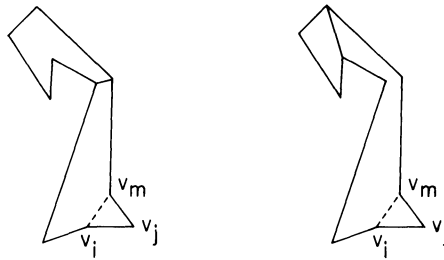


FIG. 4.1. The MD of P_{im} shown on the left merges with T_{imj} while the MD of P_{im} shown on the right does not.

The surprise comes when we discover that creating a state for each MD of a subpolygon is insufficient for a valid state space. Figure 4.2 shows a subpolygon for which the MD, which consists of a single star-shaped polygon, cannot be found by this DP formulation since the smaller subpolygons defined by a base triangle, as in Fig. 4.2b, are not star-shaped. Even though all MDs of smaller subpolygons are kept and all double and single merges are found, the MD of P_{ij} is not found. As illustrated in Fig. 4.3 there can be an exponential number of ways of decomposing a simple

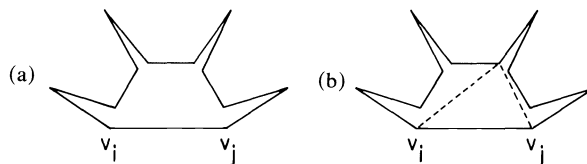


FIG. 4.2

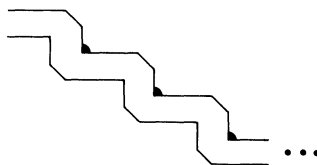


FIG. 4.3. There are $2^{n/6-4}$ MDs of the above polygon.

polygon into star-shaped polygons. If we are to develop a polynomial time algorithm for this problem we certainly cannot expand the state space further.

We will now define a set KER of potential kernel points. We begin by identifying segments that could contain a side of the boundary of the kernel of the base star-shaped polygon of a MD of a subpolygon. If v_a and v_b are vertices of polygon P then there may exist a segment of the line through v_a and v_b , that lies entirely within P , that contains v_b and exactly one other boundary point v_c of P so that v_b lies between v_a and v_c . We denote such a segment l_{ab} when it exists. That is, l_{ab} is the segment drawn inside P beginning at v_b , in the direction away from v_a , until the boundary of P is reached. We define $L = \{l_{ab} | v_b \text{ is a notch and } v_a \text{ is a vertex of } P \text{ visible from } v_b\}$. The segments in L may intersect with each other or with sides of P or both. Let KER denote the set of all such intersections together with the vertices of P as in Fig. 4.4. With these definitions we can present the following lemma which allows us to achieve a valid polynomial sized state space.

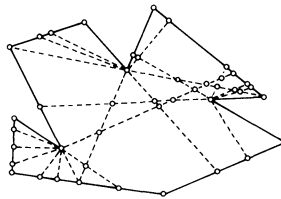


FIG. 4.4

LEMMA 4.1. *The kernel of any star-shaped base polygon of a MD D of a subpolygon P_{ij} contains a point in the set KER. The set KER contains at most $O(n^2N^2)$ points.*

Proof. The kernel of a base star-shaped polygon $S_{ij}(D)$ of a MD D of P_{ij} is the intersection of the halfplanes defined by the sides of $S_{ij}(D)$. The sides of the kernel of $S_{ij}(D)$ will either be contained in sides of $S_{ij}(D)$ or in segments extending from these sides (i.e. segments in the set L). If $S_{ij}(D)$ is convex then vertex v_i , a point in KER, is contained in the kernel of $S_{ij}(D)$. Otherwise at least one side of the kernel of $S_{ij}(D)$ is contained in a segment l_{ab} in the set L . If vertex v_b , the endpoint of l_{ab} that is a vertex of P , lies in the kernel of $S_{ij}(D)$ we are done as v_b is a point in KER. If vertex v_b is not in the kernel of $S_{ij}(D)$ then a side of the kernel adjacent to the side contained in l_{ab} must also be contained in a segment in L . The point of intersection between these segments will be a vertex of the boundary of the kernel of $S_{ij}(D)$ and also a point in KER.

The set L contains $O(nN)$ segments. There will be at most $O(n^2N^2)$ intersections amongst segments in L and $O(nN)$ intersections between segments in L and the sides of P . Therefore KER will contain at most $O(n^2N^2)$ points.

To achieve a valid state space we introduce pseudo star-shaped polygons. A pseudo star-shaped subpolygon PS_{ij} of P based along $v_i v_j$ has the property that there exists a point x in P and not in PS_{ij} so that every point of PS_{ij} can be seen from x through $v_i v_j$. We will allow star-shaped MDs of subpolygons P_{ij} to contain pseudo star-shaped polygons based along $v_i v_j$. Figure 4.2b shows pseudo star-shaped polygons that will double merge to form the correct MD of the polygon. A MD of a subpolygon P_{ij} can be found using our DP formulation if all pseudo star-shaped MDs of P_{im} and P_{mj} , $i < m < j$, are computed.

Lemma 4.1 allows us to compute only $O(n^2N^2)$ star and pseudo star-shaped decompositions for each subpolygon. For each subpolygon P_{ij} and each point x in

KER we find the smallest star-shaped or pseudo star-shaped decomposition D that either contains x as a kernel vertex of the base polygon of D or uses x to see the pseudo star-shaped base polygon of D . We call such a decomposition, (MD_x) , the MD of P_{ij} viewed from x . By associating a state with these MDs we have achieved a valid polynomial sized state space. We have again reduced the size of a state space by finding a small number of representative states.

Figure 4.5 shows a polygon whose only star-shaped MD contains an interior edge which connects two vertices of P which are not notches. Therefore, unlike the convex case, we must consider all subpolygons.

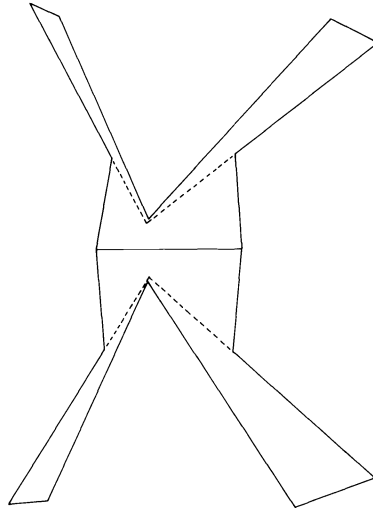


FIG. 4.5

Armed with the state space definition we can now describe the star-shaped decomposition algorithm.

Preprocessing.

1. Form the set KER of potential kernel vertices.
2. For each point $x \in \text{KER}$ store a list of the vertices of P that are visible from x .
3. For each vertex v of P determine the set of vertices of P that are visible from v and store them in a sorted list.
4. Since each visibility pair $v_i v_j$ ($i < j$) determined in step 3 is the base edge of a subpolygon these subpolygons can be sorted by the size measure $j - i$ in time $O(n^2 \log n)$.
5. For each subpolygon P_{ij} of P form the set of base triangles with base edge $v_i v_j$.

DP procedure.

1. Consider each subpolygon P_{ij} in the order computed in step 4 of the preprocessing.
2. Compute the set of MDs of P_{ij} as viewed from each member of KER.

If $j - i = 2$, P_{ij} will consist of a single triangle. Points in KER that lie in the triangle have real MDs of P_{ij} . Points in KER that do not lie in the triangle but are visible from v_i , v_{i+1} and v_j have pseudo MDs of P_{ij} .

Otherwise for each base triangle T_{imj} of P_{ij} , for each point x in KER:

- (a) First check for double merges. If both MD_x of P_{im} and MD_x of P_{mj} exist then they are merged to form a candidate for MD_x of P_{ij} .

(b) If a double-merge for x is not possible at T_{imj} , we check for single-merges. The MD_x of P_{im} single merges with T_{imj} if x is visible from v_j . To form this MD_x of P_{ij} we take the MD_x of P_{im} merged with T_{imj} together with the smallest real MD of P_{mj} . Single merges with MDs of P_{mj} are analogous.

(c) Finally, if we can perform no merges a candidate for MD_x of P_{ij} exists if x lies in T_{imj} or x is visible from v_i , v_m and v_j through edge v_iv_j . This candidate is formed by taking the smallest real MD of P_{im} together with T_{imj} and the smallest real MD of P_{mj} .

As the various candidates for the MD_x of P_{ij} are considered only the smallest is kept. Also the smallest real MD of P_{ij} will be needed when no merge past v_iv_j is performed.

3. The subpolygon $P_{1n} = P$ will be the last one considered. The smallest real MD of P_{1n} is kept as a minimum star-shaped decomposition of P .

THEOREM 4.2. *The above algorithm correctly computes the minimum star-shaped decomposition of a simple polygon.*

Proof. Lemma 4.1 implies that it is sufficient to show that we compute MD_x of P_{ij} correctly, for all $x \in \text{KER}$, for all subpolygons P_{ij} . We shall proceed by induction on the size $(j-i)$ of a subpolygon.

If $j-i=2$, P_{ij} will consist of a single triangle and clearly MD_x of P_{ij} is computed correctly if it exists. Also P_{ij} is its own smallest real MD.

As an inductive assumption let us assume that MD_x of P_{ij} , for all $x \in \text{KER}$, including the smallest real MD, have been computed correctly for all subpolygons P_{ij} with $j-i < q$. Let us consider the computation of MD_x for a subpolygon P_{ij} with $j-i = q$. Clearly there exists at least one base triangle, T_{imj} , that is contained in the base star or pseudo star-shaped polygon $S_{ij}(MD_x)$. If $S_{ij}(MD_x) = T_{imj}$ then, by the inductive assumption, we will have correctly computed MD_x by performing no merges as in part (c) of step 2 of the algorithm. If either the left side of $S_{ij}(MD_x) = v_iv_m$ or the right side of $S_{ij}(MD_x) = v_mv_j$ then, by the inductive assumption, we will have correctly computed MD_x by single merging as in part (b) of step 2 of the algorithm. If neither v_iv_m or v_mv_j lie in the boundary of $S_{ij}(MD_x)$ then, again by the inductive assumption, we will have correctly computed MD_x by double-merging as in part (a) of step 2 of the algorithm.

We can similarly compute all MD_x , for all $x \in \text{KER}$, for any subpolygon P_{ij} with $j-i = q$. That one of the MD_x of P_{ij} is the smallest real MD of P_{ij} is evident from Lemma 4.1. We have therefore shown by induction that all MD_x , for all $x \in \text{KER}$, for all subpolygons P_{ij} will be computed correctly.

Analysis. Step 1 of the preprocessing requires $O(n^2N^2)$ time. Step 5 of the preprocessing requires $O(n^3)$ time. In the DP procedure, given a viewing vertex x a double merge can be found in $O(\log n)$ time at a base triangle if the MDs of a subpolygon are stored sorted by viewing vertex. A single merge between T_{imj} and MD_x of P_{im} can be detected in $O(\log n)$ time by searching for v_j in $V(x)$. Since there are $O(n^2N^2)$ points in KER , $O(n^2N^2 \log n)$ time is spent at each base triangle. Since there are $O(n^3)$ base triangles, altogether the algorithm may require $O(n^5N^2 \log n)$ time to find the minimum star-shaped decomposition.

5. Minimum edge length decompositions. In some applications a decomposition that minimizes the total length of the internal edges used to form the decomposition is useful. Figure 5.1 shows that such a decomposition can be quite different from a decomposition that minimizes the number of components. Lingas et al. [19] have developed an $O(n^4)$ time algorithm which decomposes a rectilinear polygon into

rectangles while minimizing the total internal edge length. Using our DP approach we are able to develop polynomial time algorithms to decompose a simple polygon into each of the component types while minimizing the amount of “ink” necessary.

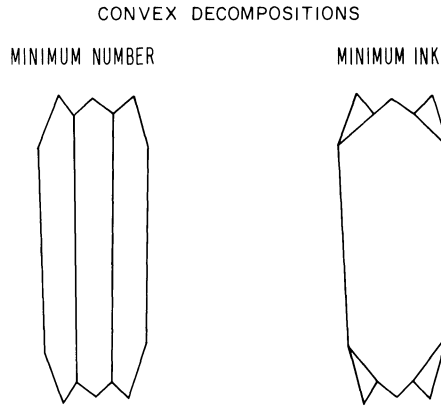


FIG. 5.1. Convex decompositions.

5.1. Convex decompositions. Since the convex minimum “ink” decomposition seems to have the widest range of applications we will treat it first. Most of the terms we defined when dealing with the convex minimum number decomposition problem will also be useful here. Refer to Fig. 2.1. We define the *length* of a convex decomposition D to be the sum of the lengths of the internal line segments forming D . A *free MD* of a subpolygon P_{ij} is a convex decomposition D of P_{ij} such that if D' is any other convex decomposition of P_{ij} then $\text{length}(D') \geq \text{length}(D)$.

Independently Greene [11] has noticed that his algorithm for the convex minimum number problem [10] can be adapted to yield an $O(N^2n^2)$ time algorithm for the convex minimum edge length problem. By developing our algorithm for that problem with our general DP approach we are able to easily adapt the algorithm for convex components to other simple types of component polygons. To begin we define the state space by letting the states be of the form s_{ij} where s_{ij} has the interpretation that subpolygon P_{ij} has been decomposed minimally. That is, we associate a free MD of P_{ij} with s_{ij} . Decisions are as before and the cost of a policy that leads to s_{ij} is equal to the length of a MD of P_{ij} . As we expect this DP formulation is not valid. Figure 5.2 shows that the free MD of P_{im} cannot merge to form the free MD of P_{ij} .

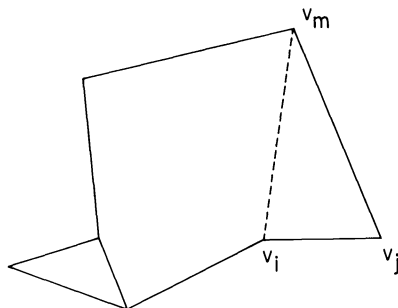


FIG. 5.2

Here we encounter a major difference between the minimum “ink” and the minimum number problems. In the minimum number case if edge $v_i v_m$ was not part of a MD of P_{ij} then one MD of P_{im} would merge with T_{imj} . Figure 5.2 shows that this is not the case with minimum “ink” decompositions. We would therefore gain nothing if we create a state for each free MD of a subpolygon.

Instead we introduce fixed MDs. A *fixed MD* of a subpolygon P_{ij} with a given left edge and right edge is the convex decomposition D of P_{ij} such that if D' is any other convex decomposition of P_{ij} with the same left edge and right edge as D then $\text{length}(D') \cong \text{length}(D)$. The usefulness of fixed MDs is established in the following lemma.

LEMMA 5.1.1. *Any members of the class of convex decompositions of a subpolygon P_{im} with a given pair of left and right edges are equivalent in terms of constructing a decomposition of subpolygon P_{ij} ($j > m$) by merging with T_{imj} .*

Proof. Analogous to that of Lemma 2.1.

Since we are seeking a minimum edge length decomposition it is clear that a state space that includes a state for each fixed MD of a subpolygon is valid. As there are only $O(n^2)$ possible pairs of left and right edges of decompositions in a subpolygon there are only a polynomial number of states in the state space and the resulting DP algorithm runs in polynomial time.

Even in this polynomial state space there are states that will not be used in constructing the optimal solution. Lemma 2.2, which states that no MD of P contains an interior edge which connects two vertices of P which are not notches, remains valid for minimum edge length decompositions. We therefore need only consider states that are associated with a fixed MD of a valid subpolygon.

The following definitions allow us to increase the efficiency of the algorithm by placing an ordering on the states. A set X of fixed MDs of a subpolygon P_{ij} has the *left increasing right fixed (LIRF) property* for a given right angle ϕ if it contains all fixed MDs of P_{ij} with the given right angle ϕ subject to the following constraint. If a fixed MD A of P_{ij} with right angle ϕ has left angle θ_1 and a fixed MD B of P_{ij} with right angle ϕ has left angle θ_2 so that $\theta_1 \cong \theta_2$ then A cannot be a member of X unless $\text{length}(A)$ is less than $\text{length}(B)$. If X is sorted in increasing order of left angle then it is also sorted in decreasing order of length. The *right increasing left fixed (RILF) property* is defined analogously.

We are now ready to describe the DP algorithm in more detail.

Preprocessing. We use the same four preprocessing steps that we used in the algorithm for computing the decomposition of a simple polygon into the minimum number of convex components.

DP procedure.

1. Consider each valid subpolygon P_{ij} in the order computed in the preprocessing.
2. Compute a set of fixed MDs of P_{ij} with the LIRF property for each valid right edge of a decomposition of P_{ij} and compute a set of fixed MDs of P_{ij} with the RILF property for each possible left edge of a decomposition of P_{ij} .

These computations are done as follows.

If $j - i = 2$, the only MD is a triangle.

Otherwise for each base triangle T_{imj} .

(a) Compute a set X of the fixed MDs of P_{ij} with left edge $v_i v_m$ with the RILF property. First all fixed MDs of P_{ij} with left edge $v_i v_m$ are found as follows.

Take a free MD of P_{im} together with

- (i) T_{imj} and a free MD of P_{mj} and

(ii) T_{imj} merged with the smallest MD of P_{mj} that will merge for each possible valid right edge of a MD of P_{mj} . The smallest MD in P_{ij} with a given right edge is found using binary search in a set with the LIRF property for the given right edge.

Now that all fixed MDs of P_{ij} with left edge $v_i v_m$ have been placed in a set X we remove some MDs so that X will have the RILF property. We have the MDs in X sorted in increasing order of right angle. Set a variable L to the length of the MD in X with the smallest right angle. X is scanned in increasing order of right angle. If the MD D under scan has length greater than or equal to L we delete D from X . Otherwise we set L to the length of D . We continue the scan until all MDs in X have been examined.

(b) A set of fixed MDs of P_{ij} with right edge $v_m v_j$ with the LIRF property is found similarly.

At this point there may be valid left edges $v_i v_m$ in subpolygons of type (b) for which no set of MDs has been found and similarly valid right edges in subpolygons of type (c) for which no set of MDs have been found. This will happen when $v_i v_m$ and $v_m v_j$ are not both valid edges so that valid base triangle T_{imj} does not exist. Figure 5.3 illustrates the situation.

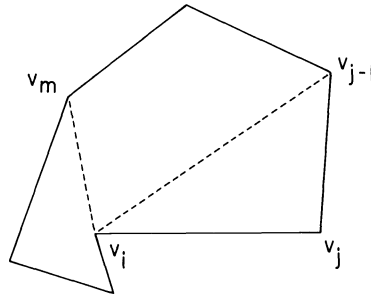


FIG. 5.3. The only base triangle of P_{ij} is $T_{i(j-1)j}$.

Consider a valid left edge $v_i v_m$ in a subpolygon of type (b) such that T_{imj} is not a valid base triangle of P_{ij} . No set of MDs with left edge $v_i v_m$ has been explicitly collected. However MDs with left edge $v_i v_m$ have been encountered while collecting MDs with various right edges. Each time a MD with the valid left edge $v_i v_m$, such that T_{imj} is not a valid base triangle of P_{ij} , is encountered we should set it aside. When all valid base triangles have been processed we will have several sets of MDs corresponding to these stray left edges. These sets can be given the RILF property using the scanning procedure described above. Similarly sets of MDs with the LIRF property are found for valid right edges $v_m v_j$ in subpolygons of type (c) for which T_{imj} is not a valid base triangle. In this way all required MDs of a subpolygon are found.

The free MD of a subpolygon P_{ij} is the smallest of all the fixed MDs found.

3. The subpolygon $P_{1n} = P$ will be the last one considered. A free MD of $P_{1,n}$ is the desired MD of P .

Proof of correctness.

LEMMA 5.1.2. Let D be a fixed MD of a subpolygon P_{ij} with left edge $v_i v_l$ and right edge $v_r v_j$. Then there exists an $O(\log n)$ time algorithm that forms a fixed MD D' of P_{ij} with the same left and right edges as D . The algorithm uses only free MDs of P_{il} , P_{lj} , P_{lr} and P_{rj} and a set of fixed MDs of P_{lr} with the RILF property with left edge $v_i v_l$ and a set of fixed MDs of P_{lj} with the LIRF property with right edge $v_r v_j$.

Proof. Assume P_{ij} is of type (a) or (b). The proof is analogous if P_{ij} is of type (c). When P_{ij} is of type (a) or (b) then T_{imj} exists so that $v_m = v_r$. The decomposition D' required can be found by computing decomposition D' in part i or ii.

(i) If $v_l = v_r$ then $C_{ij}(D)$ is a triangle (i.e. T_{imj}). Clearly the decomposition D' formed by taking a free MD of P_{il} and a free MD of P_{rj} will have the same length, left edge and right edge as D .

(ii) If $v_l \neq v_r$ then C_{ij} is not a triangle and we must consider MDs of P_{im} that merge with T_{imj} . Let A' be the smallest fixed MD of P_{im} with left edge $v_l v_l$ that will merge with T_{imj} . A' is found in $O(\log n)$ time by binary search in the set of MDs of P_{im} with the RILF property with left edge $v_l v_l$. If A is the restriction of D to $P_{im} \cup T_{imj}$ then clearly the length of A' is at least as small as the length of A . The desired MD D' is formed by taking A' merged with T_{imj} together with a free MD of P_{mj} .

The next lemma shows how the required sets of MDs for each subpolygon can be found.

LEMMA 5.1.3. *There is an algorithm that runs in $O(n^2 \log n)$ time for subpolygons of type (a) and in $O(nN \log n)$ time for subpolygons of types (b) or (c) that correctly finds a set with the LIRF property for all valid right edges in P_{ij} and a set with the RILF property for all valid left edges of P_{ij} . The procedure uses only a free MD of P_{im} and P_{mj} and a set with the LIRF property for all valid right edges in P_{mj} and a set with the RILF property for all valid left edges of P_{im} where T_{imj} is a base triangle of P_{ij} .*

Proof. Each fixed MD of P_{ij} can be found in $O(\log n)$ time using the algorithm of Lemma 5.1.2. There are $O(n^2)$ fixed MDs in a subpolygon of type (a) and $O(nN)$ fixed MDs in a subpolygon of type (b) or (c). These MDs are organized into sets with the RILF property for valid left edges of P_{ij} and into sets with the LIRF property for valid right edges of P_{ij} . This is done by sorting the MDs in each set and performing and scanning procedure described in the algorithm. A free MD of P_{ij} is found by keeping track of the smallest fixed MD of P_{ij} .

Using the above lemmas we can prove the following.

THEOREM 5.1.4. *The algorithm finds a minimum edge length decomposition of a simple polygon in $O(N^2 n^2 \log n)$ time in the worst case.*

Proof. The preprocessing requires $O(N^2 n \log n)$ time. There are $O(N^2)$ subpolygons of type (a) at which $O(n^2 \log n)$ time is spent computing fixed MDs as in Lemma 5.1.3. There are $O(Nn)$ subpolygons of type (b) or (c) at which $O(Nn \log n)$ time is spent computing fixed MDs as in Lemma 5.1.3. Altogether $O(N^2 n^2 \log n)$ time is required to compute the fixed and free MDs of each subpolygon. Since $P_{1n} = P$ a free MD of P_{1n} is the desired minimum edge length decomposition of P .

5.2. Star-shaped decompositions. For star-shaped decompositions we define fixed MDs as follows. A fixed MD of a subpolygon P_{ij} with a given kernel point $x \in \text{KER}$ is the star-shaped (or pseudo star-shaped) decomposition D of P_{ij} such that if D' is any other star-shaped (or pseudo star-shaped) decomposition of P_{ij} with the same kernel point x then $\text{length}(D') \geq \text{length}(D)$.

Lemma 4.1 allows us to compute only $O(N^2 n^2)$ fixed star and pseudo star-shaped minimum decompositions of each subpolygon. For each subpolygon we associate one state with each point in KER that can either be a kernel vertex of a base polygon of a fixed MD of P_{ij} or a vertex than can see a fixed pseudo star-shaped base polygon through $v_l v_r$. The resulting state space is valid and of polynomial size.

Figure 4.5 shows a polygon whose star-shaped minimum edge length decomposition contains an interior edge which connects two vertices which are not notches. Therefore, we again must consider all subpolygons as possible components.

The details and analysis of the star-shaped minimum edge length decomposition algorithm are very similar to those of the star-shaped minimum number decomposition algorithm. Altogether $O(N^2 n^5 \log n)$ time may be required to find a star-shaped minimum edge length decomposition.

6. Other problems. Recall that a spiral polygon is a simple polygon whose boundary chain contains at most one concave subchain. That is, a spiral polygon has at most one set of adjacent notches. Feng and Pavlidis [9] give an algorithm for decomposing a simple polygon into spiral polygons. Their algorithm does not introduce Steiner points and does not generally yield a minimum decomposition. Our DP formulation for the spiral decomposition problem is similar to that for the convex decomposition problem. In the spiral case, base angles which are notches are allowed. Some double merges may then be necessary as illustrated in Fig. 6.1. Also some of the convex equivalence classes must be subdivided. A class of MDs with a given pair of left and right angles are not all equivalent in their ability to merge. A MD with a convex base polygon can spiral merge with some base triangles where a MD with a nonconvex base polygon could not. These changes do not significantly affect the DP approach and we are able to develop polynomial time algorithms for both the minimum number and minimum “ink” spiral decomposition problems [14].

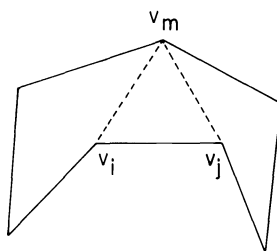


FIG. 6.1

Recall that a monotone polygon contains two extreme vertices in a preferred direction such that they are connected by two polygonal chains monotonic in the preferred direction. Lee and Preparata [17] give an $O(n \log n)$ algorithm for decomposing a simple polygon into monotone polygons without using Steiner points. Their algorithm does not generally yield a minimum decomposition. A valid DP formulation for the monotone decomposition problem could have a state for all monotone MDs of a subpolygon. This state space can be reduced by recognizing that MDs having base polygons monotone with respect to the same directions are equivalent. A convex polygon is monotone with respect to all directions and it is only sides which are adjacent to notches in a polygon that can eliminate directions of monotonicity [24]. The sides of a subpolygon divide the set of all directions into a polynomial number of classes of direction. If a subpolygon is monotone with respect to one direction in a class it is monotone with respect to all directions within that class. Therefore the state space can be further reduced by keeping only one representative MD for each of the classes of direction. Since there are only a polynomial number of such classes a polynomial DP algorithm results. This approach will work for both the minimum number and the minimum “ink” decomposition problems.

The complexity of a convex decomposition problem may increase if we allow the polygon to contain holes that must be avoided. Holes are nonoverlapping “island” simple polygons inside the main polygon. Lingas [18] has shown that if Steiner points

are allowed the problem of decomposing a polygon with polygonal holes into the minimum number of convex components is NP-hard. O'Rourke and Supowit [22] show that if Steiner points are allowed and overlapping of components is allowed decomposing a simple polygon with polygonal holes into the minimum number of convex, star-shaped or spiral components is also NP-hard. These proofs can be adapted [14] to prove that decomposing a polygon with polygonal holes into the minimum number of each of convex, star-shaped or spiral components is NP-hard when Steiner points are disallowed. It can be shown using a component design proof that decomposing a polygon with polygonal holes into the minimum number of monotone components is NP-hard and that decomposing a polygon with polygonal holes into convex components using minimum ink is also NP-hard [14].

7. Further research. If the introduction of Steiner points is allowed, most of the polygon decomposition problems remain open. The only result of this type in Chazelle and Dobkin's [4], [5], [6] polynomial time algorithm for the problem of decomposing a simple polygon into the minimum number of convex components.

Acknowledgment. I would like to thank Derek Corneil for many useful discussions on these problems.

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] R. BELLMAN, *Dynamic Programming*, Princeton Univ. Press, Princeton, NJ, 1957.
- [3] K. Q. BROWN, *Dynamic programming in computer science*, Computer Science Dept. Report CMU-CS-79-106, Carnegie-Mellon Univ., Pittsburgh, 1979.
- [4] B. CHAZELLE AND D. DOBKIN, *Decomposing a polygon into its convex parts*, Proc. 11th Annual ACM Symposium on Theory of Computing, 1979, pp. 38-48.
- [5] B. CHAZELLE, *Computational geometry and convexity*, Ph.D. Thesis, Yale Univ., New Haven, CT, 1980.
- [6] B. CHAZELLE AND D. DOBKIN, *Optimal convex decompositions*, in Computational Geometry, North-Holland, Amsterdam, 1984.
- [7] H. EL-GINDY AND D. AVIS, *A linear algorithm for determining visibility from a point in a polygon*, J. Algorithms, 2 (1981), pp. 186-197.
- [8] S. E. ELMAGHRABY, *The concept of "State" in discrete dynamic programming*, J. Math. Anal. Appl., 29 (1970), pp. 523-557.
- [9] H. FENG AND T. PAVLIDIS, *Decomposition of polygons into simpler components: feature generation for syntactic pattern recognition*, IEEE Trans. Comput., C-24 (1975), 636-650.
- [10] D. GREENE, *The decomposition of polygons into convex parts*, Manuscript, Xerox Parc, 1982.
- [11] ———, Private communication, 1982.
- [12] T. IBARAKI, *Minimal representations of some classes of dynamic programming*, Inform. and Control, 27 (1975), pp. 289-328.
- [13] R. M. KARP AND M. HELD, *Finite-state processes and dynamic programming*, SIAM J. Appl. Math., 15 (1967), pp. 693-718.
- [14] J. M. KEIL, *Decomposing polygons into simpler components*, Ph.D. Thesis, Univ. Toronto, Toronto, 1983.
- [15] J. M. KEIL AND J.-R. SACK, *Minimum decompositions of polygonal objects*, in Computational Geometry, North-Holland, Amsterdam, 1984.
- [16] H. T. KUNG, F. LUCCIO AND F. P. PREPARATA, *On finding the maxima of a set of vectors*, J. Assoc. Comput. Mach., 22 (1975), pp. 469-476.
- [17] D. T. LEE AND F. P. PREPARATA, *Location of point in a planar subdivision and its applications*, this Journal, 6 (1977), pp. 594-606.
- [18] A. LINGAS, *The power of non-rectilinear holes*, Proc. 9th Colloquium Automata, Languages and Programming, Aarhus, 1982.
- [19] A. LINGAS, R. PINTER, R. RIVEST AND A. SHAMIR, *Minimum edge length decompositions of rectilinear figures*, unpublished manuscript, MIT, 1981.

- [20] W. LIPSKI, E. LODI, F. LUCCIO, C. MUGNAI AND L. PAGLI, *On two-dimensional data organization II*, *Fundamenta Informaticae*, 2 (1979).
- [21] W. NEWMAN AND R. SPROULL, *Principles of Interactive Computer Graphics*, Second ed., McGraw-Hill, New York, 1979.
- [22] J. O'ROURKE AND K. J. SUPOWIT, *Some NP-hard polygon decomposition problems*, *IEEE Trans. Information Theory*, IT-29 (1983), pp. 181-190.
- [23] T. PAVLIDIS, *Analysis of set patterns*, *Pattern Recognition*, 1 (1968), pp. 165-178.
- [24] F. P. PREPARATA AND K. J. SUPOWIT, *Testing a simple polygon for monotonicity*, *Inform. Proc. Letters*, 12 (1981), pp. 161-164.
- [25] B. SCHACHTER, *Decomposition of polygons into convex sets*, *IEEE Trans. Comput.*, C-27 (1978), pp. 1078-1082.
- [26] G. T. TOUSSAINT, *Pattern recognition and geometrical complexity*, *Proc. Fifth International Conference on Pattern Recognition*, Miami Beach, 1980, pp. 1324-1347.

A LINEAR-TIME ALGORITHM FOR COMPUTING K -TERMINAL RELIABILITY IN SERIES-PARALLEL NETWORKS*

A. SATYANARAYANA† AND R. KEVIN WOOD‡

Abstract. Let $G = (V, E)$ be a graph whose edges may fail with known probabilities and let $K \subseteq V$ be specified. The K -terminal reliability of G , denoted $R(G_K)$, is the probability that all vertices in K are connected. Computing $R(G_K)$ is, in general, NP-hard. For some series-parallel graphs, $R(G_K)$ can be computed in polynomial time by repeated application of well-known reliability-preserving reductions. However, for other series-parallel graphs, depending on the configuration of K , $R(G_K)$ cannot be computed in this way. Only exponential-time algorithms as used on general graphs were known for computing $R(G_K)$ for these "irreducible" series-parallel graphs. We prove that $R(G_K)$ is computable in polynomial time in the irreducible case, too. A new set of reliability-preserving "polygon-to-chain" reductions of general applicability is introduced which decreases the size of a graph, and conditions are given for a graph admitting such reductions. Combining all types of reductions, an $O(|E|)$ algorithm is presented for computing the reliability of any series-parallel graph irrespective of the vertices in K .

Key words. algorithms, complexity, network reliability, series-parallel graphs, reliability-preserving reductions

1. Introduction. Analysis of network reliability is of major importance in computer, communication and power networks. Even the simplest models lead to computational problems which are NP-hard for general networks [5], although polynomial-time algorithms do exist for certain network configurations such as "ladders" and "wheels" and for some series-parallel structures such as the well-known "two-terminal" series-parallel networks. In this paper, we show that a class of series-parallel networks, for which only exponentially complex algorithms were previously known [7], [8], can be analyzed in polynomial time. In doing this, we introduce a new set of reliability-preserving graph reduction of general applicability and produce a linear-time algorithm for computing the reliability of any graph with an underlying series-parallel structure.

The network model used in this paper is an undirected graph $G = (V, E)$ whose edges may fail independently of each other, with known probabilities. The reliability analysis problem is to determine the probability that a specified set of vertices $K \subseteq V$ remains connected, i.e., the K -terminal reliability of G . Computing K -terminal reliability was first shown to be NP-hard by Rosenthal [12], and it follows from Valiant [17] that the problem is $\#P$ -complete even when G is planar. Two special cases of this reliability problem are the most frequently encountered, the terminal-pair problem where $|K|=2$, and the all-terminal problem where $K = V$. These problems are also $\#P$ -complete [11], in general, although their complexities are unknown when G is planar.

In network reliability analysis, three reliability-preserving graph reductions are well-known: the series reduction, the degree-2 reduction (an extension of the series reduction for problems with $|K| > 2$) and the parallel reduction. From the reliability viewpoint, we classify series-parallel graphs into two types, those which are reducible to a single edge using standard series, parallel and degree-2 reductions, and those

* Received by the editors February 23, 1982, and in final revised form June 15, 1984. This work was conducted at the Operations Research Center, University of California, Berkeley, California 94720, under Air Force Office of Scientific Research grant AFOSR-81-0122 and under U.S. Army Research Office contract DAAG29-81-K-0160.

† Department of Computer Science, Stevens Institute of Technology, Hoboken, New Jersey 07030.

‡ Department of Operations Research, Naval Postgraduate School, Monterey, California 93943.

which are not. The former type is “reducible” and the latter “irreducible.” For example, the series-parallel graph of Fig. 1a is reducible if $K = \{v_1, v_2\}$, but irreducible for $K = \{v_1, v_6\}$. Thus, the reducibility of a series-parallel graph, for the purpose of reliability evaluation, depends on the nature of the vertices included in K . A more detailed exposition of this concept appears in § 2.

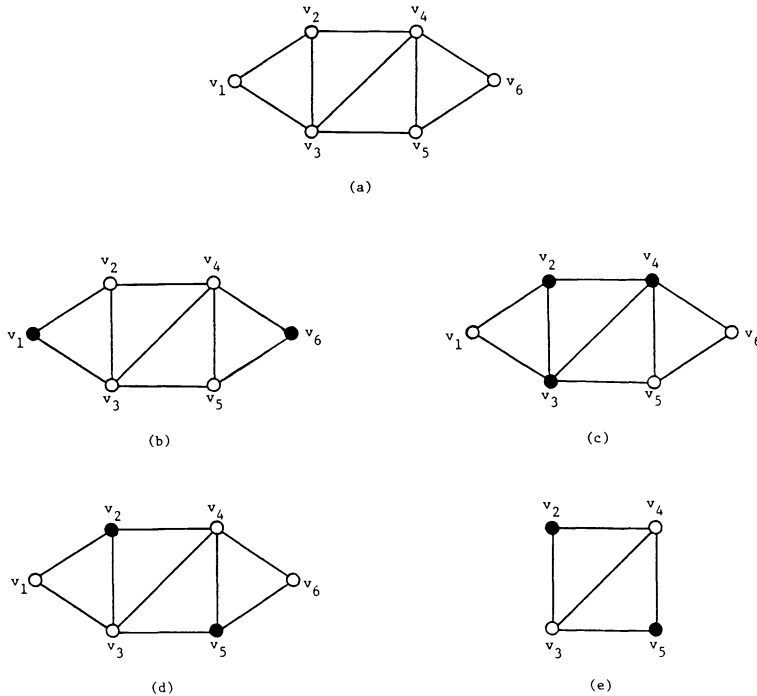


FIG. 1. Reducible and irreducible series-parallel graphs. Note: Darkened vertices represent K -vertices.

The K -terminal reliability of a reducible series-parallel graph can be computed in polynomial time. Several methods exist for the solution of the terminal-pair problem for such a graph, i.e., for a two-terminal series-parallel network [9], [15], and for $|K| > 2$, direct extensions of the methods can be used. However, it has been believed that computing the reliability of irreducible series-parallel graphs is as hard as the general problem. (The use of series-parallel reductions with multi-state edges [13] is applicable to this problem although this has not been recognized. We do not follow this tack because of the simplicity and generality obtained by maintaining binary-state edges.) The purpose of this paper is threefold: (1) to introduce a new set of reliability-preserving graphs reductions called “polygon-to-chain reductions,” (2) to show that by using these reductions, irreducible series-parallel graphs become reducible, and (3) to give a linear-time algorithm for computing the reliability of any graph with a series-parallel structure.

In a graph, a chain is an alternating sequence of vertices and edges, starting and ending with vertices such that end vertices have degree greater than 2 and all internal vertices have degree 2. Two chains with the same end vertices constitute a polygon. In § 3, we show that a polygon can be replaced by a chain and that this transformation will yield a reliability-preserving reduction. We discuss the relationship between

irreducible series-parallel graphs and polygons in § 4. Using the polygon-to-chain reductions in conjunction with the three simple reductions mentioned earlier, a polynomial-time procedure is then outlined which will compute the reliability of any series-parallel graph. This procedure is very simple but not of linear-time complexity, so in § 5 we develop algorithm which is shown to operate in $O(|E|)$ time. This algorithm will compute the K -terminal reliability of any graph having an underlying series-parallel structure. Finally, in § 6, we briefly discuss an extension to the algorithm to reduce a nonseries-parallel graph as far as possible so that the algorithm could be used as a subroutine in a reliability analysis program for general networks.

2. Preliminaries. Consider a graph $G = (V, E)$ in which all vertices are perfectly reliable but any edge e_i may fail with probability q_i or work with probability $p_i = 1 - q_i$. All edge failures are assumed to occur independently of each other. Let K be a specified subset of V with $|K| \geq 2$. When certain vertices of G are specified to be in K , we denote the graph G together with the set K by G_K . We will refer to the vertices of G belonging to K as the K -vertices of G_K . The K -terminal reliability of G , denoted by $R(G_K)$, is the probability that the K -vertices in G_K are connected. K -terminal reliability is a generalization of the common reliability measures, all-terminal reliability and terminal-pair reliability where $K = V$ and $|K| = 2$, respectively.

Reliability of a separable graph. A *cutvertex* of a graph is a vertex whose removal disconnects the graph. A *nonseparable graph* is a connected graph with no cutvertices. A *block* of a graph is a maximal nonseparable subgraph.

Let $G = (V, E)$ be a separable graph and $v \in V$ be any cutvertex in G . G can be partitioned into two connected subgraphs $G^{(1)} = (V_1, E_1)$ and $G^{(2)} = (V_2, E_2)$ such that $V_1 \cup V_2 = V$, $V_1 \cap V_2 = v$, $E_1 \cup E_2 = E$ and $E_1 \cap E_2 = \emptyset$. Also, $E_1 \neq \emptyset$ and $E_2 \neq \emptyset$. Denote $K_1 = K \cap V_1$ and $K_2 = K \cap V_2$. If one of the K_i is null, say $K_1 = \emptyset$, then $G^{(1)}$ is *irrelevant* and $R(G_K) = R(G_{K_2}^{(2)})$. Otherwise, assuming $K_1 \neq \emptyset$ and $K_2 \neq \emptyset$, it is well known that $R(G_K) = R(G_{K_1 \cup v}^{(1)})R(G_{K_2 \cup v}^{(2)})$. ($R(G_K) \equiv 1$ if $|K| = 1$. Therefore, if $K_i = \{v\}$ then $R(G_{K_i \cup v}^{(i)}) \equiv 1$ and the above statement is still true.) Thus the reliability of a separable graph can be computed by evaluating the reliabilities of its blocks separately. For this reason, we henceforth consider only nonseparable graphs.

Simple reductions. In order to reduce the size of graph G_K , i.e. reduce $|V| + |E|$, and therefore reduce the complexity of computing $R(G_K)$, *reliability-preserving reductions* are often applied: Certain edges and/or vertices in G are replaced to obtain G' ; new edge reliabilities are defined; a new set K' is defined; and a multiplicative factor Ω is defined; all such that $R(G_K) = \Omega R(G'_{K'})$. The following three reliability-preserving reductions are well known and are called *simple reductions*.

A *parallel reduction* replaces a pair of edges $e_a = (u, v)$ and $e_b = (u, v)$ with a single edge $e_c = (u, v)$ and defines $p_c = 1 - q_a q_b$, $K' = K$, and $\Omega = 1$.

Suppose $e_a = (u, v)$ and $e_b = (v, w)$ such that $u \neq w$, $\deg(v) = 2$, and $v \notin K$. A *series reduction* replaces e_a and e_b with a single edge $e_c = (u, w)$, and defines $p_c = p_a p_b$, $K' = K$ and $\Omega = 1$.

Suppose $e_a = (u, v)$ and $e_b = (v, w)$ such that $u \neq w$, $\deg(v) = 2$, and $\{u, v, w\} \subseteq K$. A *degree-2 reduction* replaces e_a and e_b with a single edge $e_c = (u, w)$ and defines $p_c = p_a p_b / (1 - q_a q_b)$, $K' = K - v$, and $\Omega = 1 - q_a q_b$.

Series-parallel graphs. The following definition should not be confused with the definition of a "two-terminal" series parallel network in which two vertices must remain fixed. No special vertices are distinguished here. In a graph, edges with the same end vertices are *parallel edges*. Two nonparallel edges are *adjacent* if they are incident on a common vertex. Two adjacent edges are *series edges* if their common vertex is of

degree 2. Replacing a pair of series (parallel) edges by a single edge is called a series (parallel) *replacement*. A series-parallel graph is a graph that can be reduced to a tree by successive series and parallel replacements. Clearly, if a series-parallel graph is nonseparable, then the resulting tree, after making all series and parallel replacements, contains exactly one edge.

We wish to clarify the subtle difference between the term “replacement” used here and the term “reduction” used with respect to simple reductions. Replacement is a strictly graph-theoretic term indicating some edges or vertices from G are removed and then replaced by other edges or vertices to create a new graph G' . A reduction is defined, on the other hand, with respect to G , K , and edge reliabilities. A reduction includes the act of replacing edges or vertices in G to create G' along with defining edge reliabilities, K' , and Ω , all such that $R(G_K) = \Omega R(G'_K)$, i.e. reliability is preserved. For example, in graph G as shown in Fig. 1a, series replacements are possible while no (reliability-preserving) simple reductions are possible in the corresponding G_K for $K = \{v_1, v_6\}$ (Fig. 1b). Motivated by the difference between graphs which allow replacements but, with K and edge reliabilities defined, do not allow reliability-preserving simple reductions, we distinguish between graphs which can and cannot be reduced by simple reductions.

Reducible and irreducible series-parallel graphs. Clearly, if G has no series or parallel edges, then for any K , G_K admits no simple reductions. If G is a series-parallel graph, then a simple reduction might or might not exist in G_K depending upon the vertices of G that are chosen to be in K . For example, consider the series-parallel graph G of Fig. 1a. The graph G_K , for $K = \{v_2, v_3, v_4\}$ as in Fig. 1c, can be reduced to a single edge by successive, simple reductions. On the other hand, for $K = \{v_1, v_6\}$, G_K admits no simple reductions (Fig. 1b). A series-parallel graph G_K is *reducible* if it can be reduced to a single edge by successive, simple reductions. If G_K is reduced to a single edge e_i using m reductions, then $R(G_K) = p_i \prod_{k=1}^m \Omega_k$ where Ω_k is the multiplicative factor defined by the k th reduction. Note that any series-parallel graph G is reducible for the all-terminal problem since any degree-2 vertex in G_V allows a degree-2 reduction.

It is possible for a (nonseparable) series-parallel graph to admit one or more simple reductions for a specified K and still not be completely reducible to a single edge. As an illustration, consider G_K of Fig. 1d. Two series reductions may be applied to this graph to obtain the graph of Fig. 1e, but no further simple reductions are possible. A graph G_K is an *irreducible* series-parallel graph if G_K cannot be completely reduced to a single edge using simple reductions.

Chains and polygons. In a graph, a *chain* χ is an alternating sequence of distinct vertices and edges, $v_1, (v_1, v_2), v_2, (v_2, v_3), v_3, \dots, v_{k-1}, (v_{k-1}, v_k), v_k$, such that the internal vertices, v_2, v_3, \dots, v_{k-1} , are all of degree 2 and the end vertices, v_1 and v_k , are of degree greater than 2. A chain need not contain any internal vertices, but it must contain at least one edge and two end vertices. The length of a chain is simply the number of edges it contains. A *subchain* is a connected subset of a chain beginning and ending with a vertex and containing at least one edge. Both the end vertices of a subchain may be of degree 2. The notation χ will also be used for a subchain with the usage differentiated by context.

Suppose χ_1 and χ_2 are two chains of lengths l_1 and l_2 , respectively. If the two chains have common end vertices u and v , then $\Delta = \chi_1 \cup \chi_2$ is a *polygon* of length $l_1 + l_2$. In other words, a polygon is a cycle with the property that exactly two vertices of the cycle are of degree greater than 2. While this definition allows two parallel edges to constitute a polygon, we will initially require a polygon to be of length at least 3.

3. Polygon-to-chain reductions. In this section a new set of reliability-preserving reductions will be introduced which replace a polygon with a chain and always reduces $|V|+|E|$ by at least 1. Consider a graph G_K which does not admit any simple reductions but does contain some polygon Δ . In general, no such polygon need exist, but, if it does exist, then the number of possible configurations is limited.

Property 1. Let G_K be a graph which admits no simple reductions. If G_K contains a polygon, then it is one of the seven types given in the first column of Table 1.

Proof. This follows from the facts that (i) every degree-2 vertex of G_K is a K -vertex, (ii) there can be no more than two K -vertices in a chain, and (iii) the length of any chain in G_K is at most 3.

Polygon-to-chain transformations. Let Δ_j be a type j polygon in G_K , a graph which admits no simple reductions. Let u and v be the vertices in Δ_j such that $\deg(u) > 2$ and $\deg(v) > 2$. Then, $\Delta_j = \chi'_j \cup \chi''_j$, where χ'_j and χ''_j are chains in G_K with common end vertices u and v . Replacing the pair χ'_j and χ''_j by the corresponding chain χ_j , as in Table 1, is called a *polygon-to-chain transformation*.

In Theorem 1 we will prove that a polygon-to-chain transformation can be used to produce a reliability-preserving, polygon-to-chain reduction. It is useful here, however, to make the distinction between a polygon-to-chain reduction and a polygon-to-chain transformation, in the same manner that simple reductions and replacements are differentiated. A transformation is only a topological mapping of a graph G to a graph G' and ignores all considerations of reliability including K -vertices. A reduction includes the topological transformation as well as all reliability calculations and changes in K -vertices.

The proof technique of Theorem 1 requires that we first discuss the use of conditional probabilities for computing the reliability of a graph in a general context. Let $e_i = (u, v)$ be some edge of G_K and let F_i denote the event that e_i is working and \bar{F}_i denote the complementary event that e_i has failed. Using rules of conditional probability, the reliability of G_K can be written as

$$(1) \quad R(G_K) = p_i R(G_K | F_i) + q_i R(G_K | \bar{F}_i) = p_i R(G'_{K'}) + q_i R(G''_{K''})$$

where

$$G' = (V - u - v + w, E - e_i), \quad w = u \cup v,$$

$$K' = \begin{cases} K & \text{if } u, v \notin K, \\ K - u - v + w & \text{if } u \in K \text{ or } v \in K \end{cases}$$

and

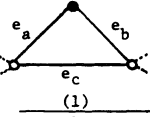
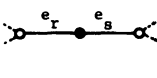
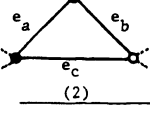
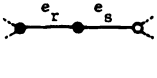
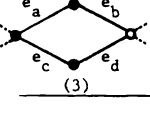
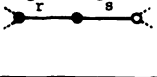
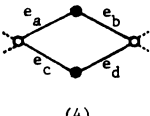
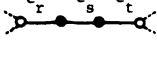
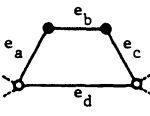
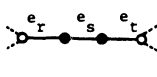
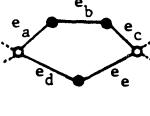
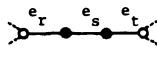
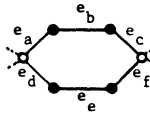
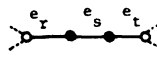
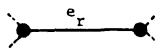
$$G'' = (V, E - e_i),$$

$$K'' = K.$$

F_i and \bar{F}_i are said to “induce” $G'_{K'}$ and $G''_{K''}$ from G_K , respectively. (“Induce” is not used in the standard graph-theoretic sense here.) $G'_{K'}$ is G_K with edge e_i contracted, and $G''_{K''}$ is G_K with edge e_i deleted.

Equation (1) can be applied recursively on the induced graphs and simple reductions made where applicable within the recursion. After repeated applications of the formula, the induced graphs are either reduced to single edges for which the reliability is simply the probability that the edge works, or some K -vertices become disconnected, in which case the reliability of the induced graph is zero. In this way, the reliability of any general graph may be computed. This method of computing the reliability of a graph is known as “factoring” [10], [14] and is a special case of pivotal decomposition

TABLE 1
Polygon-to-chain reductions

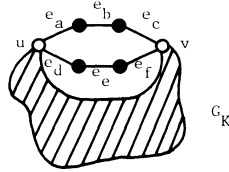
| Note: Darkened vertices represent K-vertices | | | |
|--|--|---|--|
| Polygon Type | Chain Type | Reduction Formulas | New Edge Reliabilities |
|  <p>(1)</p> |  | $\alpha = q_a p_b q_c$ $\beta = p_a q_b q_c$ $\delta = p_a p_b p_c \left(1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} \right)$ | $p_r = \frac{\delta}{\alpha + \delta}$ |
|  <p>(2)</p> |  | $\alpha = q_a p_b q_c$ $\beta = p_a q_b q_c$ $\delta = p_a p_b p_c \left(1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} \right)$ | $p_s = \frac{\delta}{\beta + \delta}$ |
|  <p>(3)</p> |  | $\alpha = p_a q_b q_c p_d + q_a p_b p_c q_d + q_a p_b q_c p_d$ $\beta = p_a q_b p_c q_d$ $\delta = p_a p_b p_c p_d \left(1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} + \frac{q_d}{p_d} \right)$ | $\Omega = \frac{(\alpha + \delta)(\beta + \delta)}{\delta}$ |
|  <p>(4)</p> |  | $\alpha = q_a p_b q_c p_d$ $\beta = p_a q_b q_c p_d + q_a p_b p_c q_d$ $\delta = p_a q_b p_c q_d$ $\gamma = p_a p_b p_c p_d \left(1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} + \frac{q_d}{p_d} \right)$ | |
|  <p>(5)</p> | <p>$K > 2$</p>  <p>See note</p> | $\alpha = q_a p_b p_c q_d$ $\beta = p_a q_b p_c q_d$ $\delta = p_a p_b p_c p_d \left(1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} + \frac{q_d}{p_d} \right)$ | $p_r = \frac{\gamma}{\alpha + \gamma}$ |
|  <p>(6)</p> |  | $\alpha = q_a p_b p_c q_d p_e$ $\beta = p_a q_b p_c (p_d q_e + q_d p_e) + p_b (q_a p_c p_d q_e + p_a q_c q_d p_e)$ $\delta = p_a p_b p_c p_d q_e$ $\gamma = p_a p_b p_c p_d p_e \left(1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} + \frac{q_d}{p_d} + \frac{q_e}{p_e} \right)$ | $p_t = \frac{\gamma}{\delta + \gamma}$ $\Omega = \frac{(\alpha + \gamma)(\beta + \gamma)(\delta + \gamma)}{\gamma^2}$ |
|  <p>(7)</p> |  | $\alpha = q_a p_b p_c q_d p_e p_f$ $\beta = p_a q_b p_c (q_d p_e p_f + p_d q_e p_f + p_d p_e q_f) + p_a p_b q_c p_f (p_d q_e + q_d p_e) + q_a p_b p_c p_d (q_e p_f + p_e q_f)$ $\delta = p_a p_b q_c p_d p_e q_f$ $\gamma = p_a p_b p_c p_d p_e p_f \left(1 + \frac{q_a}{p_a} + \frac{q_b}{p_b} + \frac{q_c}{p_c} + \frac{q_d}{p_d} + \frac{q_e}{p_e} + \frac{q_f}{p_f} \right)$ | <p>Note: For $K = 2$, new chain is</p>  $p_r = (p_b + p_a q_b p_c p_d) / \Omega$ $\Omega = p_b + p_a q_b p_c$ |

of a general binary coherent system [1]. For our purposes, factoring will only be applied to the edges of a single polygon or a chain.

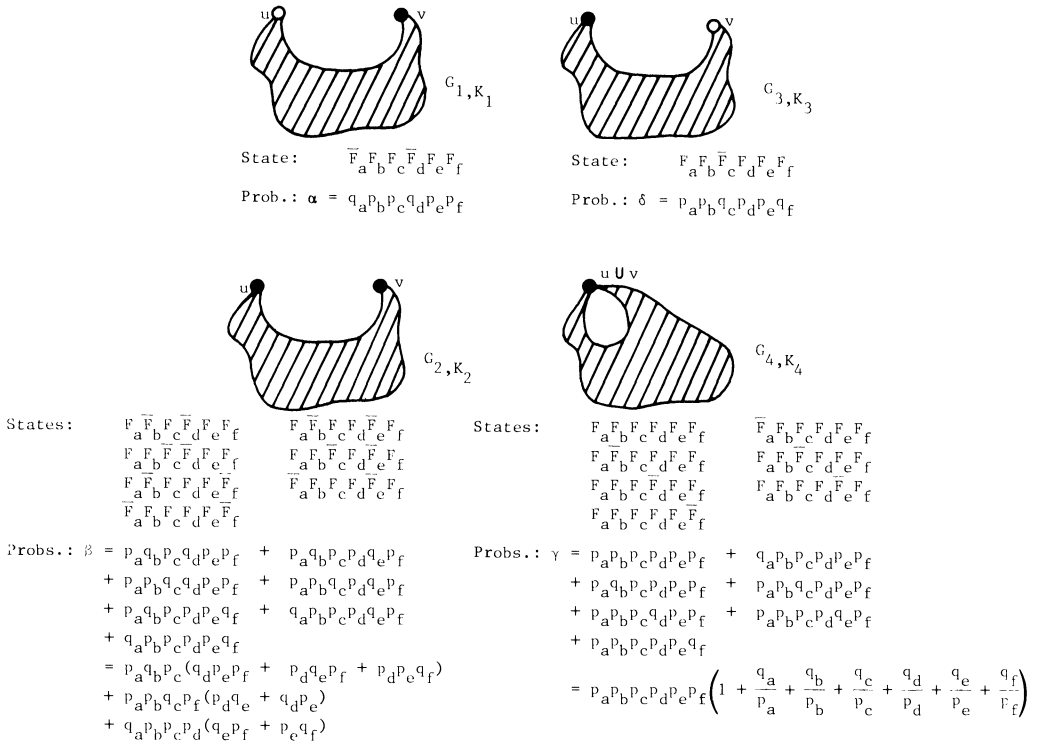
Polygon-to-chain reductions.

THEOREM 1. Suppose G_K contains a type j polygon. Let G'_K denote the graph obtained from G_K by replacing the polygon Δ_j with the chain χ_j having appropriately defined edge probabilities, and let Ω_j be the corresponding multiplication factor, all as in Table 1. Then, $R(G_K) = \Omega_j R(G'_K)$.

We prove the exactness of reduction 7 only, since reductions 1-6 may be shown in a similar fashion. Figs. 2 and 3 illustrate the proof of the theorem. To improve readability in the proof, we drop the subscript “7” on $\alpha, \beta, \delta, \gamma,$ and Ω even though, strictly speaking, these are functions of the type of reduction.



(a) Schematic of a graph with a type 7 polygon.



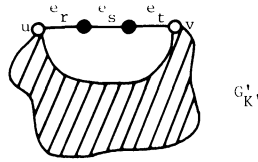
(b) Nonfailed induced graphs.

FIG. 2

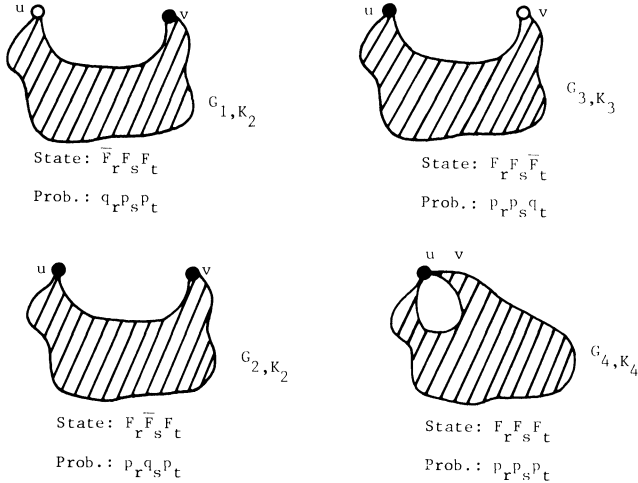
Proof of Theorem 1. Let F_i be the event that edge e_i in the polygon is working and let \bar{F}_i be the event that edge e_i has failed. \bar{F} denotes a compound event or state such as $F_a F_b \bar{F}_c F_d \bar{F}_e F_f$, and F denotes the set of all 2^6 such states. Also, $z_i = 1$ if F_i occurs and $z_i = 0$ if \bar{F}_i occurs. By conditional probability and extension of (1),

$$(2) \quad R(G_K) = \sum_{F \in \bar{F}} p_a^{z_a} q_a^{1-z_a} \cdots p_f^{z_f} q_f^{1-z_f} R(G_K | F).$$

Only sixteen of the possible sixty-four states are nonfailed states where $R(G_K | F) \neq 0$. Each nonfailed state will induce a new graph with a corresponding set



(a) Graph of Fig. 2 with polygon replaced by chain.



(b) Nonfailed induced graphs

FIG. 3

of K -vertices of which there only four different possibilities. Figure 2 gives these four graphs $G_{i,K}$, $i = 1, 2, 3, 4$, the states under which the graphs are induced, and the summed state probabilities in each case, α , β , δ , and γ . Thus, by grouping and eliminating terms, (2) is reduced to

$$(3) \quad R(G_K) = \alpha R(G_{1,K_1}) + \beta R(G_{2,K_2}) + \delta R(G_{3,K_3}) + \gamma R(G_{4,K_4}).$$

Now $G'_{K'}$ is obtained from G_K by replacing the polygon with a chain $u, e_r, v_1, e_s, v_2, e_t, w$, and redefining K as shown in Fig. 3. Using conditional probabilities again,

$$(4) \quad R(G'_{K'}) = p_r q_s p_t R(G'_{K'} | (F_r \bar{F}_s F_t)) + q_r p_s p_t R(G'_{K'} | (\bar{F}_r F_s F_t)) \\ + p_r p_s q_t R(G'_{K'} | (F_r F_s \bar{F}_t)) + p_r p_s p_t R(G'_{K'} | (F_r F_s F_t))$$

where only the nonfailed states have been written.

The four nonfailed states of $G'_{K'}$ induce the same four graphs which the nonfailed states of G_K induce. Multiplying (4) by a factor Ω , we thus have

$$(5) \quad \Omega R(G'_{K'}) = \Omega p_r q_s p_t R(G_{1,K_1}) + \Omega q_r p_s p_t R(G_{2,K_2}) + \Omega p_r p_s q_t R(G_{3,K_3}) + \Omega p_r p_s p_t R(G_{4,K_4}).$$

Equating, term by term, the coefficients in (3) and (5) gives

$$\alpha = \Omega q_r p_s p_t = \Omega (1 - p_r) p_s p_t, \quad \delta = \Omega p_r p_s q_t = \Omega p_r p_s (1 - p_t), \\ \beta = \Omega p_r q_s p_t = \Omega p_r (1 - p_s) p_t, \quad \gamma = \Omega p_r p_s p_t.$$

These four equations in the four unknowns $\Omega, p_r, p_s,$ and p_t may be easily solved to obtain

$$p_r = \frac{\gamma}{\alpha + \gamma}, \quad p_s = \frac{\gamma}{\beta + \gamma},$$

$$p_t = \frac{\gamma}{\delta + \gamma}, \quad \Omega = \frac{(\alpha + \gamma)(\beta + \gamma)(\delta + \gamma)}{\gamma^2},$$

which are the values given in Table 1 for a type 7 polygon. The reader may verify that when these values are substituted into (4), we obtain

$$\Omega R(G'_K) = \alpha R(G_{1,K_1}) + \beta R(G_{2,K_2}) + \delta R(G_{3,K_3}) + \gamma R(G_{4,K_4}) = R(G_K). \quad \square$$

It can be seen from Table 1 that polygon-to-chain reductions, like simple reductions, always reduce $|V| + |E|$ by at least 1.

Theorem 1 can be extended to give a result which can be useful for computing the reliability of a general graph. In a nonseparable graph, a *separating pair* is a pair of vertices whose deletion disconnects the graph. For example, vertices u and v in Fig. 2 are a separating pair. Using the same conditioning arguments as in the proof of Theorem 1, it can be shown that any subgraph between a separating pair can be replaced by a chain of 1, 2, or 3 edges to yield a reliability-preserving reduction. For two special cases, it has been shown that a subgraph between a separating pair can be replaced by a single edge [6]. The first case occurs when the subgraph including the separating pair has no K -vertices, and the second case occurs when the separating pair belongs to K . The fact that a chain can always be used to replace any subgraph, irrespective of the K -vertices, greatly increases the generality of any algorithm which uses this reduction.

4. Properties of series-parallel graphs. In this section we set down some properties of series-parallel graphs with respect to topology and reliability. We prove that a series-parallel graph must admit a polygon-to-chain reduction if all simple reductions have first been performed. Thus, every series-parallel graph is reducible irrespective of the vertices in K . Using this fact, we then outline a simple polynomial-time procedure for computing the reliability of such graphs.

The following property is a simple extension of the definition of a series-parallel graph.

Property 2. Let G' be the graph obtained from G by applying one or more of the following operations:

- a series replacement;
- a parallel replacement;
- an inverse series replacement (replace an edge by two edges in series);
- an inverse parallel replacement (replace an edge by two edges in parallel).

Then, G' is a series-parallel graph if and only if G is series-parallel.

Proof of Property 2 may be found in [3]. The next two properties show that the series-parallel structure of a graph is not altered by simple or polygon-to-chain reductions.

Property 3. Let G' be the graph obtained by a polygon-to-chain transformation on G . Then G' is a series-parallel graph if and only if G is series-parallel.

Proof. G' may be obtained from G by one or more series replacements, a parallel replacement, and one or more inverse series replacements, in that order. Thus, this property follows directly from Property 2. \square

Property 4. Let G'_K be the graph obtained from G_K by applying a simple reduction or a polygon-to-chain reduction on G_K . Then, G' is a series-parallel graph if and only if G is series-parallel.

Proof. A series or degree-2 reduction implements a series replacement, a parallel reduction implements a parallel replacement, and a polygon-to-chain reduction implements a polygon-to-chain transformation on G . Hence, by Properties 2 and 3, G' is a series-parallel graph if and only if G is a series-parallel. \square

By next proving that every series-parallel graph G_K admits a simple reduction or a polygon-to-chain reduction, it will be possible to show that $R(G_K)$ can be computed in polynomial time for such graphs.

Property 5. Let G_K be a series-parallel graph. Then, G_K must admit either a simple reduction or one of the seven types of polygon-to-chain reductions given in Table 1.

Proof. If G_K admits a simple reduction, then we are done. If G_K has no simple reductions, then by Property 1, any polygon of G_K must be one of the seven types given in Table 1. Hence, we need only show that G contains a polygon. Let G' be the graph obtained by replacing all chains in G with single edges. If G' contains a pair of parallel edges, then the two chains in G corresponding to this pair of edges constitute a polygon. We argue that G' must contain a pair of parallel edges. If G' has no parallel edges, no simple reductions are possible in G' since all vertices in G' have degree greater than 2. Thus, G' and hence G are not series-parallel graphs, which is a contradiction. \square

One simple procedure for computing $R(G_K)$ can now be outlined as follows: (1) Make all simple reductions; (2) find a polygon and make the corresponding reduction; and (3) repeat steps 1 and 2 until G_K is reduced to a single edge. If G_K is originally series-parallel, then Properties 4 and 5 guarantee that the above procedure eventually reduces G_K to a single edge. The reliability is calculated by initializing $M \leftarrow 1$, letting $M \leftarrow M\Omega_j$ whenever a polygon-to-chain reduction of type j is made, and letting $M \leftarrow M\Omega$, for $\Omega = 1 - q_a q_b$, whenever a degree-2 reduction is made on some edges e_a and e_b . At the end of the algorithm with a single remaining edge e_i , the reliability of the original graph is given by $R(G_K) = Mp_i$.

The total number of parallel and polygon-to-chain reductions executed by this procedure, before the graph is reduced to a single edge, is exactly $|E| - |V| + 1$. This is because the number of fundamental cycles in a connected graph is $|E| - |V| + 1$, and a parallel or polygon-to-chain reduction deletes exactly one such cycle [2]. The complexity of steps (1) and (2) above can be linear in the size of G , and thus, the running time of the whole procedure is at best quadratic in the size of G . In order to develop a linear-time algorithm, we have found it necessary to move the parallel reduction from the domain of simple reductions to the domain of polygon-to-chain reductions. Indeed, a parallel reduction is a trivial case of a polygon-to-chain reduction with a multiplier $\Omega = 1$. We will henceforth consider two parallel edges to be the type 8 polygon and the parallel reduction to be the type 8 polygon-to-chain reduction.

5. An $O(|E|)$ algorithm for computing the reliability of any series-parallel graph. The objective here is to develop an efficient, linear-time algorithm for computing the reliability of any series-parallel graph. All results needed to present this algorithm have been established; however, some additional notation and definitions must be given.

If u and v are the end vertices of a chain χ , then u and v are said to be *chain-adjacent*. When it is necessary to distinguish these vertices, we will use the notation $\chi(u, v)$. A subchain with end vertices u and v will also be denoted $\chi(u, v)$ but in this case u and

v cannot be said to be chain-adjacent. The algorithm is presented next, followed by a proof of its validity and linear complexity. The algorithm reduces G_K to two edges in parallel and prints $R(G_K)$ if G is initially series-parallel (We stop at two edges in parallel instead of a single edge because these edges do not form a polygon by our definition; their end vertices do not have degrees greater than 2.), or prints a message that G is not series-parallel. Comments are enclosed in square brackets.

ALGORITHM.

Input: A nonseparable graph G with vertex set V , $|V| \geq 2$, edge set E , $|E| \geq 2$, and set $K \subseteq V$, $|K| \geq 2$. Edge probabilities p_i for each edge $e_i \in E$.

Output: $R(G_K)$ if G is series-parallel or a message that G is not series-parallel.

Begin

$M \leftarrow 1$.

Perform all series reductions.

Perform all degree-2 reductions letting $M \leftarrow M\Omega$ for each such reduction.

Construct list, $T \leftarrow \{v \mid v \in V \text{ and } \deg(v) > 2\}$ marking all such v "onlist."

Mark all $v \notin T$ "offlist."

While $T \neq \emptyset$ and $|E| > 2$ **do**

Begin

Remove v from T .

$i \leftarrow 1$. [Index of the next chain out of v to be searched]

Until $i > 3$ or v is deleted or $\deg(v) = 2$ **do**

Begin

Search the i th chain out of v .

$i \leftarrow i + 1$.

If a polygon $\Delta(v, w)$ is found **then do**

Begin

Apply the appropriate type j polygon-to-chain reduction to $\Delta(v, w)$ to obtain $\chi(v, w)$, and let $M \leftarrow M\Omega_j$.

$i \leftarrow i - 1$.

If $\deg(v) = 2$ or $\deg(w) = 2$ **then do**

Begin

Apply all possible series and degree-2 reductions on the chain (or cycle) containing subchain $\chi(v, w)$ to obtain completely reduced chain $\chi(x, y)$ (or parallel edges (x, y) and (x, y)), letting $M \leftarrow M\Omega$ for each degree-2 reduction.

If $y \neq v$ and y is "offlist" **then** mark y "onlist" and add y to T .

If $x \neq v$ and x is "offlist" **then** mark x "onlist" and add x to T .

End

End

End

End

If $|E| = 2$ **then** print (" $R(G_K)$ is" $M(1 - q_a q_b)$) [for $E = \{e_a, e_b\}$]
else print (" G is not series-parallel").

End.

The key to the algorithm is the way in which the "until" loop operates. This loop says: "Sequentially search chains incident to v reducing any polygons which are found and making any subsequent series and degree-2 reductions until either (a) v is shown to be chain-adjacent to three distinct vertices, or (b) v is completely deleted from G through the reductions, or (c) v becomes a degree-2 vertex through the reductions.

No chain is ever searched more than once each time this loop is entered. The correctness of the algorithm is not hard to show. Arguments similar to those presented here may be found in [16] where the problem is the recognition of two-terminal series-parallel directed graphs.

Suppose firstly that G consists of a single cycle. The initial series and degree-2 reductions will reduce G_K to two edges in parallel, T will be empty, and the algorithm therefore gives $R(G_K)$ correctly at the final step of the algorithm. Next, suppose that G does not consist of a single cycle, in which case T will not be empty and an initial search for a polygon will begin. Since all initial series and degree-2 reductions were performed, by Property 5, any polygon found must be one of the eight specified types. If a polygon is found and reduced, the resulting chain may, in fact, be a subchain. If this happens, some new series and degree-2 reductions may be admitted on the chain (or cycle) containing that subchain but nowhere else. All such reductions are made when applicable. Thus, every time the "until" loop of the algorithm is entered or iterated, the graph admits no series or degree-2 reductions, and only polygons of the eight given types can exist.

Vertices are continually removed from the stack T and replaced, at most two at a time, only when polygon-to-chain reductions are made. At most $|E| - |V|$ polygon-to-chain reductions can ever be made since each polygon-to-chain reduction removes exactly one of the $|E| - |V| + 1$ fundamental cycles of G and the final reduced graph must retain at least one fundamental cycle. Therefore, at most $|V| + 2(|E| - |V|) = 2|E| - |V|$ vertices can ever pass through T before T becomes empty and the "while" loop must terminate. If $|E| = 2$ at that point, then $R(G_K)$ is correctly given at the last step of the algorithm since only reliability-preserving series, degree-2, and polygon-to-chain reductions are ever performed. Property 4 proves that the original graph must have been series-parallel.

If $|E| > 2$ when T becomes empty, then we must show that the reduced graph is not series-parallel and that the original graph was not series parallel. In this case, every vertex v with $\deg(v) > 2$ is chain-adjacent to at least three distinct vertices. This is true since (i) every vertex v with $\deg(v) > 2$ is initially put in the list T and its chain-adjacent vertices checked in the "until" loop and (ii) whenever the chain-adjacency of a vertex or vertices is altered (this can occur to at most two vertices at a time) after a polygon-to-chain reduction, then this vertex or vertices are returned to the list T if not already there. The following property proves that a graph with the given chain-adjacency structure is not series-parallel.

Property 6. Let G be a nonseparable graph such that all vertices v with $\deg(v) > 2$ are chain-adjacent to at least three distinct vertices. Then, G is not a series-parallel graph.

Proof. Let G' be the graph obtained from G by first replacing all chains with single edges in a sequence of series replacements and then removing any parallel edges in a sequence of parallel replacements. By Property 2, G is a series-parallel if and only if G' is a series-parallel. Now, every vertex $v \in V'$ has $\deg(v) > 2$ and there are no parallel edges in E' . Thus, G' admits no series or parallel replacements and cannot be series-parallel. Therefore, G cannot be series-parallel. \square

This proves that if the algorithm terminates with $|E| > 2$, the reduced graph is not series-parallel, and Property 4 proves that the original graph could not have been series-parallel either. This establishes the validity of the algorithm. We now turn our attention to its computational complexity.

In order to show that the algorithm is linear in the size of G , we use a multi-linked adjacency list to represent G . In this representation, for each vertex a doubly-linked

list of adjacent vertices corresponding to incident edges is kept together with the associated edge probabilities. Every edge is represented twice since we are dealing with an undirected graph, and additional links are kept between both representations of each edge. Such an adjacency list can be initialized in $O(|V|+|E|)$ time for any graph. Using the above representation, any series, degree-2, or polygon-to-chain reduction can be carried out in constant time. Also, none of the reductions ever require the use of more vertices or edges after the reduction than before. This means that if any new edges or vertices must be defined, old ones can be reused and the size of the graph representation is never increased.

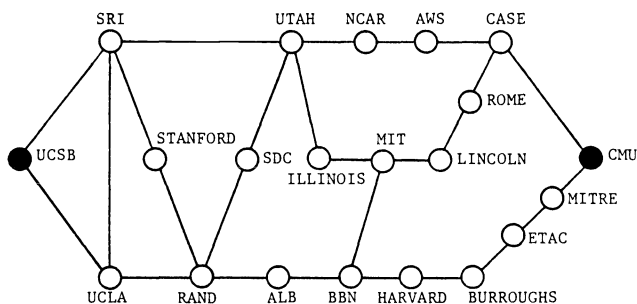
Now, initial series and degree-2 reductions are performed on $O(|V|)$ time only once and, consequently, may be ignored for purposes of complexity analysis. Consider the "until" loop of the algorithm. Each time chains emanating from the current vertex v are searched here, and l polygons are found and reduced, the maximum amount of work which can be performed is bounded by $C_1 + C_2l$, where C_1 is a constant bounding the amount of work required to find three chains with distinct end vertices, and C_2 is a constant bounding the amount of work required to perform a polygon-to-chain reduction and any subsequent series and degree-2 reductions. That C_1 is, in fact, a constant is obvious. C_2 is a constant because there are only eight types of polygons to recognize and reduce, and because after reduction of $\Delta(v, w)$ to $\chi(v, w)$, any chain $\chi(x, y)$ containing $\chi(v, w)$ can have length at most 9. Thus $\chi(x, y)$ would require at most 8 series and degree-2 reductions to be completely reduced. This worst case could occur if $\deg(v) = \deg(w) = 2$ after the polygon-to-chain reduction and the subchains $\chi(x, v)$, $\chi(v, w)$, and $\chi(w, y)$, which were proper chains before the reduction, are at their maximum possible lengths of 3. (In the case that G is a cycle after a polygon-to-chain reduction, the maximum length of such a cycle is 6, and reduction of the cycle to two edges in parallel requires at most 4 series and degree-2 reductions.) Since at most $2|E| - |V|$ vertices ever pass through T , and since at most $|E| - |V|$ polygon-to-chain reductions will ever be performed, the work performed by the algorithm is bounded by $C_1(2|E| - |V|) + C_2(|E| + |V|)$. Under the connectivity assumptions $|E| \geq |V|$, and we have therefore proven the following theorem:

THEOREM 2. *Let G be a nonseparable series-parallel graph. Then, for any K , $R(G_K)$ can be computed in $O(|E|)$ time.*

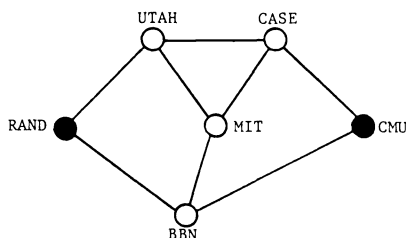
6. Extension to the algorithm. The algorithm of § 5 can be extended to make all possible simple and polygon-to-chain reductions in a nonseries-parallel graph. In this way, the extended algorithm can be used as a subroutine in a more general network reliability algorithm for computing $R(G_K)$ when G is not series-parallel. The complexity of computing $R(G_K)$ can often be reduced to some degree by this device.

Suppose the reduction algorithm of § 5 starts with a nonseries-parallel graph G . After termination of the algorithm, G_K may or may not have been partially reduced. From the proof of Property 6, the only possible remaining reductions are polygon-to-chain reductions. Each such polygon-to-chain reduction would correspond to a parallel edge replacement used to obtain the graph G' of that proof. Therefore, G_K can be totally reduced by first applying the algorithm and then finding and reducing any remaining polygons. This can easily be done by searching all chains emanating from all vertices v with $\deg(v) > 2$. In the worst case, each chain, and thus each edge, must be searched twice. Parallel chains can be recognized in constant time, and therefore, the added computation is $O(|E|)$ and the algorithm with the extension remains $O(|E|)$.

To illustrate the usefulness of the extended algorithm for a general graph, let us consider the ARPA computer network configuration as shown in Fig. 4a [4]. Suppose



(a) ARPA computer network.



(b) Reduced network.

FIG. 4

we are interested in the terminal-pair reliability between UCSB and CMU. Application of the extended algorithm yields a reduced network as shown in Fig. 4b with redefined edge reliabilities and an associated multiplier. The original reliability problem is now equivalent to computing the terminal-pair reliability between RAND and CMU in the reduced network. In linear time the size of the network has been reduced considerably and, because computing the reliability of a general network is exponential in its size, a significant computational advantage should be gained.

REFERENCES

- [1] R. E. BARLOW AND F. PROSCHAN, *Statistical Theory of Reliability*, Holt, Rinehart and Winston, New York, 1975.
- [2] N. DEO, *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall, Englewood Cliffs, NJ, 1974.
- [3] R. J. DUFFIN, *Topology of series-parallel networks*, J. Math. Analysis Appl., 10 (1965), pp. 303-318.
- [4] L. FRATTA AND U. MONTANARI, *A Boolean algebra method for computing the terminal reliability in a communication network*, IEEE Trans. Circuit Theory, CT-20 (1973), pp. 203-211.
- [5] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.
- [6] J. N. HAGSTROM, *Combinatoric tools for computing network reliability*, Ph.D. Thesis, Dept. of IEOR, Univ. California, Berkeley, 1980.
- [7] E. HANSLER, G. K. MCAULIFE, R. S. WILKOV, *Exact calculation of computer network reliability*, Networks, 4 (1974), pp. 95-112.
- [8] P. M. LIN, B. J. LEON, T. C. HUANG, *A new algorithm for symbolic system reliability analysis*, IEEE Trans. Reliability, R-25 (1976), pp. 2-15.
- [9] K. B. MISRA, *An algorithm for the reliability evaluation of redundant networks*, IEEE Trans. Reliability, R-19 (1970), pp. 146-151.

- [10] F. MOSKOWITZ, *The analysis of redundancy networks*, AIEE Trans. (Commun. Electron.), 77 (1958), pp. 627-632.
- [11] J. S. PROVAN AND M. O. BALL, *The complexity of counting cuts and of computing the probability that a graph is connected*, working Paper MS/S 81-002, College of Business and Management, Univ. Maryland, College Park, 1981.
- [12] A. ROSENTHAL, *Computing reliability of complex systems*, Ph.D. Thesis, Dept. of EECS, Univ. California, Berkeley, 1974.
- [13] ———, *Series and parallel reductions for complex measures of network reliability*, Networks, 11 (1981), pp. 323-334.
- [14] A. SATYANARAYANA AND M. K. CHANG, *Network reliability and the factoring theorem*, Networks, to appear; Also, ORC 81-12, Operations Research Center, Univ. California, Berkeley, 1981.
- [15] J. SHARMA, *Algorithm for reliability evaluation of a reducible network*, IEEE Trans. Reliability, R-25 (1976), pp. 337-339.
- [16] J. VALDES, R. E. TARJAN AND E. L. LAWLER, *The recognition of series parallel digraphs*, this Journal, 11 (1982), pp. 297-313.
- [17] L. G. VALIANT, *The complexity of enumeration and reliability problems*, this Journal, 8 (1979), pp. 410-421.

A DEPTH-UNIVERSAL CIRCUIT*

STEPHEN A. COOK†‡ AND H. JAMES HOOVER†

Abstract. This paper describes a family of depth-universal circuits. For any n, c, d there is a universal circuit $U(n, c, d)$ that can simulate any circuit α having n inputs, of size c and depth d , and U has depth $O(d)$ and size $O(c^3 d / \log c)$. The construction is used to give an alternative proof of a theorem of Ruzzo showing the invariance under different uniformity conditions of complexity classes defined by uniform circuit families.

Key words. Boolean circuits, parallel processing, computational complexity

1. Introduction. Boolean circuits are one of the more general models of parallel computation. Within this model, the fast circuits are particularly interesting, especially those in NC, that is the circuits of polynomial size and poly log depth. A natural question to ask is: Are there general purpose circuits for performing fast parallel computations?

More precisely, given n, c, d is there a circuit U of size $c^{O(1)}$ and depth $O(d)$ that can simulate any n -input circuit of size c and depth d ? In this paper we present a universal circuit of size $O(c^3 d / \log c)$ and depth $O(d)$. We call this circuit depth-universal because of the only constant factor increase in depth.

First, some background. Valiant [Va76] examines the question of universal circuits and presents one of size $O(c \log c)$ and depth $O(c)$. Using an information-theoretic argument, Valiant shows that such circuits are optimal with respect to size, and thus there are no size-universal circuits. Although this circuit has very poor depth performance, Valiant also mentions that permutation networks can be used to construct a universal circuit of size $O(dc \log c)$ and depth $O(d \log c)$.

In [Ho78], using an infinite-circuit model, Hoover constructs a depth universal circuit with polynomial activation size. In addition, this circuit also addresses the problem of translating the encoding of the circuit to be simulated into the appropriate control signal settings for the universal circuit. Our construction uses a number of the ideas in [Ho78].

In a noncircuit vein, Goldschlager [Go78] presents a time-universal parallel computer. His conglomerate can simulate any p processor, t time conglomerate in time $O(t)$ using $O(2^t)$ processors. It has the drawback of requiring an exponential number of processors.

Galil and Paul [GP83] also look at universal parallel computers. Their efficient general purpose parallel machine can simulate any p processor, t time parallel computation using $O(p)$ processors and $O(t \log^2 p)$ time. Thus their machine is processor-universal. Using sorting and permutation networks, Galil and Paul improve the time performance and construct two other universal machines with $O(t \log p)$ time and $O(tp \log p)$ size.

In [adH83], auf der Heide constructs a time-universal parallel computer with size $O(p^{1+\epsilon})$. It can simulate any parallel computer that has predictable communication (i.e., each processor can precompute the address of the processors it wants to communicate with in the next t steps in time $O(t)$). The parallel computer in [adH83] consists

* Received by the editors October 1, 1983, and in revised form February 20, 1984. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada.

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

‡ The research of this author was supported in part by the Killam Foundation of Canada.

of a network whose nodes are processors with an unbounded number of states. The simulation uses ideas similar to ours but in a different setting and was done independently.

We begin by describing a depth-universal circuit for circuits of depth $\Omega(\log n)$. This circuit is uniform and of polynomial size. However, it takes as input a nonstandard encoding of the circuit to be simulated. Next we describe a circuit that converts the more common standard circuit encoding into the nonstandard encoding. The conversion circuit is also uniform and polynomial size, but requires depth $O(\log n \log \log n)$. Thus we obtain a depth-universal circuit accepting the standard circuit encoding for circuits of depth $\Omega(\log n \log \log n)$. Finally, we focus on the problem of generating circuit encodings in the first place, and this leads to the alternative proof of Ruzzo's uniformity result.

2. Definitions. We use the following accepted definitions [Bo77], [Co81].

Let $B_n = \{f | f: \{0, 1\}^n \rightarrow \{0, 1\}\}$ denote the set of all Boolean functions of rank n . A circuit α with n inputs is a finite directed acyclic graph such that each node has a label from $\{x_1, \dots, x_n\} \cup B_0 \cup B_1 \cup B_2$. A node labelled x_i must have indegree zero and is called an *input node*. A node ν with label $g \in B_i$ must have indegree i , and one edge into ν is associated with each argument of g . The circuit has a sequence of $m \geq 1$ nodes designated *output nodes* and labelled y_1, \dots, y_m . When the variables x_i are assigned values from $\{0, 1\}$ every output node y_i assumes a unique value in $\{0, 1\}$. Thus the circuit α computes a function $f: \{0, 1\}^n \rightarrow \{0, 1\}^m$ in the obvious way.

We are interested in computing functions f from $\{0, 1\}^* \rightarrow \{0, 1\}^*$. Let f^n be f restricted to inputs of length n . A family $\langle \alpha_n \rangle$ of circuits computes f if and only if α_n computes f^n for all n .

Let $c(\alpha)$, the complexity of α , be the number of nodes in α , and let $d(\alpha)$, the depth of α , be the length of the longest path in α . For any nontrivial single output circuit $c(\alpha) = \Omega(n)$ and $d(\alpha) = \Omega(\log n)$.

We define two types of encodings for the circuit α . The standard encoding, denoted $\bar{\alpha}$, is the usual encoding for describing circuits. The extended encoding, denoted $\hat{\alpha}$, is derived from Ruzzo's [Ru81] extended connection language. Our universal circuit accepts the extended encoding directly. It can be modified to accept the standard encoding by adding the conversion circuit described in § 4. Unfortunately, the conversion circuit is not depth preserving for very shallow circuits.

DEFINITION. The *standard encoding* $\bar{\alpha}$ of a circuit α is a string $\in \{0, 1\}^*$ grouped into 4-tuples (ν, g, l, r) , one tuple for each node of α , interpreted as follows. Node number ν is a g -gate, where $g \in \{null, input\} \cup B_0 \cup B_1 \cup B_2$. The left (right) input to ν is numbered l (r). The nodes of α are numbered $1, \dots, c$ with nodes $1, \dots, n$ corresponding to inputs x_1, \dots, x_n . The gate type, g , is encoded using 5 bits, while ν, l and r are encoded in binary as strings of $\lceil \log_2 c \rceil$ bits. Missing inputs to ν are numbered 0. Tuples appear in $\bar{\alpha}$ ordered by increasing ν .

DEFINITION. The *extended encoding* $\hat{\alpha}$ of a circuit α is a string $\in \{0, 1\}^*$ of 5-tuples (ν, π, g, l, r) . Values of π are strings from $\{L, R\}^*$ representing backwards paths, via left and right inputs, from node number ν . The node resulting from following path π from node ν is denoted $\pi(\nu)$. If π is empty, that is $\pi = \epsilon$, $\pi(\nu) = \nu$. Node $\pi(\nu)$ is a g -gate with left (right) input numbered l (r). The nodes of α are numbered $1, \dots, c$ with nodes $1, \dots, n$ corresponding to inputs x_1, \dots, x_n . Furthermore, paths π are restricted to have length $\leq \lceil \log_2 c(\alpha) \rceil$. The values of g, ν, l and r are encoded as for the standard encoding. There is one tuple in $\hat{\alpha}$ for each possible combination of ν and π . If $\pi(\nu)$ is not a legitimate node then the gate type is *null* and l, r are 0. The tuples are ordered in $\hat{\alpha}$ by increasing ν and π .

DEFINITION. A family $\langle \alpha_n \rangle$ of circuits is U_{BC} uniform provided some deterministic Turing machine can compute the transformation $1^n \rightarrow \bar{\alpha}_n$ in space $O(\log c(\alpha_n))$.

3. The depth-universal circuit. We now describe the construction of a family of depth-universal circuits. Each member U of the family can simulate circuits of a particular size, depth and input length. When supplied with the extended encoding $\hat{\alpha}$ of the circuit α , and with the values of the inputs, U computes the same function as α . U will have depth the same order as α and have only polynomially greater size.

THEOREM 1. For all n, c, d where $c \geq d \geq \lceil \log_2 c \rceil, c \geq n$ there exists a depth-universal circuit $U = U(n, c, d)$ with n regular inputs $x_1, \dots, x_n, O(c^2 \log c)$ encoding inputs and c outputs y_1, \dots, y_c , such that: When supplied with the extended encoding, $\hat{\alpha}$, of any n -input circuit α with $d(\alpha) \leq d$ and $c(\alpha) \leq c$ the circuit U simulates α on x_1, \dots, x_n . The outputs y_1, \dots, y_c of U correspond to the nodes $1, \dots, c$ of α . Furthermore, $d(U) = O(d)$ and $c(U) = O(c^3 d / \log c)$, and U is U_{BC} uniform in the sense that the transformation $1^{n01^c01^d} \rightarrow \overline{U(n, c, d)}$ can be computed by a deterministic Turing machine in space $O(\log c)$.

Construction of U . The circuit α will be simulated by U in stages. Each stage will simulate a slice of α of thickness h . Slice i contains all nodes ν with $(i - 1)h < \text{depth}(\nu) \leq ih$, where $\text{depth}(\nu)$ denotes the length of the longest path from an input to ν . There are $\lceil d/h \rceil + 1$ slices in α . Slice 0 consists of just the inputs x_1, \dots, x_n .

U is constructed as a multi-layer sandwich of switching layers separated by simulation layers. Simulation layer i computes the values of all nodes in slice i . In addition it recomputes the values of all nodes simulated in earlier slices. Switch layer i connects the outputs from simulation layer $i - 1$ to simulation layer i .

A simulation layer is constructed with trees of universal gates. Each universal gate u can compute all 22 of the functions in $B_0 \cup B_1 \cup B_2$. It has two regular inputs labelled L and R and control inputs s_1, \dots, s_5 . The control inputs specify which function u is to compute and have the same encoding as g , the gate-type, in the extended circuit encoding.

The universal tree of depth h , denoted T^h , is a complete binary tree constructed by connecting the output of each universal gate to the L or R input of its parent. Each universal gate of T^h is naturally identified by a path $\pi \in \{L, R\}^*$ of length $\leq h$ from the root. The peripheral universal gates of T^h each have two inputs, L and R , resulting in 2^{h+1} input leaves. The input leaves of T^h are identified by paths of length $h + 1$, and this induces a natural numbering of the leaves. T^h has size $O(2^h)$ and depth $O(h)$. By appropriate duplication of inputs at the leaves, and setting of control inputs to each universal gate, each universal tree can simulate any one-output circuit of depth h .

A switching layer is constructed with selector trees. The selector tree for c inputs, S^c , has c regular inputs w_1, \dots, w_c and $k = \lceil \log_2 c \rceil$ control inputs b_1, \dots, b_k . Setting the control inputs to represent i in binary selects input w_i to the output. If i is not in $1, \dots, c$ the output of S^c is 0. S^c has size $O(c)$ and depth $O(\log c)$.

The switching layers of U have depth $O(\log c)$. This gives U a depth of $O((h + \log c) \lceil d/h \rceil)$. In order for U to be depth-universal, that is have depth $O(d)$, h must be chosen to be $\Omega(\log c)$. We choose $h = \lceil \log_2 c \rceil$. Letting $q = \lceil d/h \rceil$, U has levels numbered $0, \dots, q$.

Note that the size of this circuit can be improved to $O(c^{2+\epsilon}d)$ by choosing $h = \epsilon \log c, \epsilon < 1$. Since our main interest is in keeping the size polynomial, h is chosen for simplicity.

The i th level is constructed as shown in Fig. 1. At the bottom of level i are c universal trees $T_{i,1}^h, \dots, T_{i,c}^h$. The output of $T_{i,j}^h$ is denoted $z_{i,j}$. The universal tree $T_{i,j}^h$ is responsible for simulating node j of α . Attached to input leaf k of $T_{i,j}^h$ is a selector

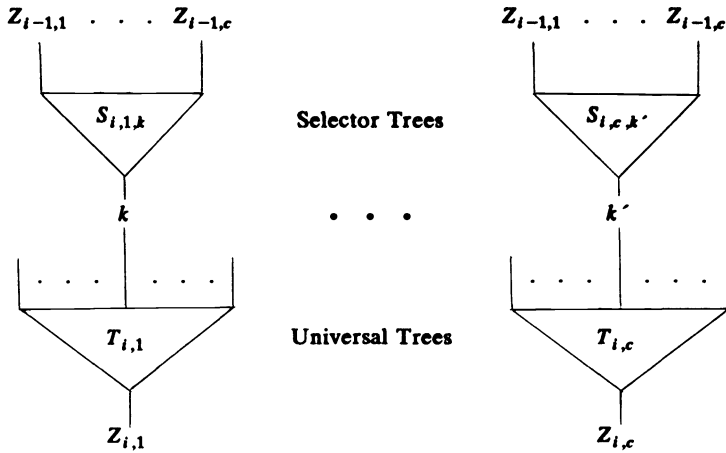


FIG. 1. Level i of the universal circuit.

tree, $S_{i,j,k}^c$, for c inputs. For $i \geq 1$, input y_m of $S_{i,j,k}^c$ is connected to $z_{i-1,m}$, namely the output from the universal tree simulating node m in the previous level. For $i = 0$ (level 0), input y_m of $S_{i,j,k}^c$ is connected to regular input x_m if $1 \leq m \leq n$, and the constant 0 if $m > n$. At level q the output of $T_{q,j}^h$ is designated y_j .

Each of the $O(d/\log c)$ levels of U has c^2 selector trees of size $O(c)$ for a total size for U of $O(c^3 d/\log c)$.

Operation of U . The extended encoding $\hat{\alpha}$ is used directly, without any additional circuitry, to set up U for simulating α . Signals from tuple (ν, π, g, l, r) are connected into U as follows. For all universal trees $T_{i,\nu}^h$ simulating node ν , the bits of g are attached to the control inputs of universal gate π in the tree. The selector tree attached to input leaf m (m encoded as path λ) of $T_{i,\nu}^h$, namely $S_{i,\nu,m}^c$, must select the appropriate L or R input from the previous slice. This is achieved by connecting the control inputs of the selector tree to l or r depending on whether λ ends in L or R .

Thus node ν in slice i is computed in terms of the values of nodes in slice $i - 1$.

Correctness of U . A straightforward induction on the level number, i , shows that level i correctly computes the value of all nodes ν with $\text{depth}(\nu) \leq ih$. Since α has $\text{depth} \leq qh$, by level q all nodes have been correctly simulated, and the output of $T_{i,\nu}^c$, namely y_ν , is the value of node ν . Since U is nothing but a very regular arrangement of trees it is easy to verify that U is U_{BC} uniform.

4. Converting circuit encodings. The standard encoding $\bar{\alpha}$ of a circuit α is minimal in the sense that each node and edge of α appears only once in $\bar{\alpha}$. The extended encoding $\hat{\alpha}$ used by U contains redundant edge information in the form of short, $\lceil \log_2 c(\alpha) \rceil$, interconnection paths. This makes its length about the square of the standard encoding length. It is this redundancy that allows the direct setting of the selectors and universal trees and so permits the simple structure and small depth of U .

As observed by Ruzzo [Ru81] both the standard and extended encodings can be converted to each other in $DSPACE(\log c(\alpha))$. To convert from extended to standard is immediate and just involves discarding unnecessary tuples. To convert from standard to extended requires the traversing of paths of length $O(\log c(\alpha))$ back from a node to its predecessors. The space is kept small since each node has fan-in ≤ 2 and so the paths can be represented as $O(\log c(\alpha))$ length strings from $\{L, R\}^*$.

Under the U_{BC} definition of uniformity all reasonable encodings are $DSPACE(\log c(\alpha))$ equivalent, and whenever $\langle \alpha_n \rangle$ is U_{BC} uniform we can reasonably expect to have $\hat{\alpha}$ available as input to U .

However it can still be argued that $\hat{\alpha}$ contains a lot of precomputed information whose explicit computation by a circuit would increase the depth of U . The next result shows that $\hat{\alpha}$ does not contain that much more easily accessible information than $\bar{\alpha}$. If we restrict U to accepting $\bar{\alpha}$ we pay a depth penalty only for circuits of depth $d = o(\log c \log \log c)$.

THEOREM 2. *There exists a conversion circuit CU that when supplied with the standard encoding $\bar{\alpha}$ for any n -input circuit α of size c and depth d will output the extended encoding $\hat{\alpha}$. Furthermore CU is U_{BC} uniform and $c(CU) = O(c^4)$ and $d(CU) = O(\log c \log \log c)$.*

Proof. Let $\nu \in \{1, \dots, c\}$ be a node of α and $\pi \in \{L, R\}^*$ be a path with $|\pi| \leq \lceil \log_2 c \rceil$. The circuit CU has two main stages.

(1) Compute all possible $\nu(\pi)$.

(2) For each valid $\nu(\pi)$ produce the tuple (ν, π, g, l, r) of the extended encoding directly from the tuple $(\nu(\pi), g, l, r)$ of the standard encoding.

Step (1) is the most difficult step. In the subcircuit for (1) there are nodes labelled $p_{ij}(\pi)$ and defined by

$$p_{ij}(\pi) = 1 \quad \text{iff} \quad i(\pi) = j.$$

That is, $p_{ij}(\pi) = 1$ if and only if node j is reached by following path π back from node i .

The $p_{ij}(\pi)$ are computed by a subcircuit as follows:

- (a) $p_{ij}(\varepsilon), p_{ij}(L), p_{ij}(R)$ are set directly from the standard encoding.
- (b) The remaining $p_{ij}(\pi)$ are computed from the equation:

$$p_{ij}(\pi_a \pi_b) = \bigvee_{k=1}^c p_{ik}(\pi_a) \wedge p_{kj}(\pi_b).$$

Note that there are $O(c^2 c)$ of the $p_{ij}(\pi)$, and that for given i and π at most one of $p_{i,1}(\pi), \dots, p_{i,c}(\pi)$ has value 1.

For (1a) setting $p_{ij}(L)$ and $p_{ij}(R)$ requires checking that $l = j$ and $r = j$ in (i, g, l, r) . This can be done with a circuit of size $O(\log c)$ and depth $O(\log \log c)$. Thus (1a) has size $O(c^2 \log c)$ and depth $O(\log \log c)$.

For step (1b) choosing π_a, π_b to be about the same length keeps the number of compositions to $\log |\pi| = O(\log \log c)$. Each evaluation of the equation requires depth $O(\log c)$ and so computing $p_{ij}(\pi)$ has depth $O(\log c \log \log c)$. Each evaluation has size $O(c)$ for a total size of $O(c^4)$ for step (1b).

Step (2) is a straightforward selection. The tuple (i, π, g, l, r) is produced by using $p_{i,1}(\pi), \dots, p_{i,c}(\pi)$ to select g, l, r from the c tuples $(1, g, l, r), \dots, (c, g, l, r)$. Producing each tuple of the extended encoding requires a selector of size $O(c \log c)$ and depth $O(\log c)$. Thus step (2) has size $O(c^3 \log c)$ and depth $O(\log c)$.

So the circuit CU has size $O(c^4)$ and depth $O(\log c \log \log c)$.

COROLLARY 1. *For all n, c, d where $c \geq d \geq \log c \log \log c$ and $c \geq n$ there exists a depth-universal circuit $U = U(n, c, d)$ with n regular inputs $x_1, \dots, x_n, O(c \log c)$ encoding inputs and c outputs y_1, \dots, y_c such that: When supplied with the standard encoding, $\bar{\alpha}$ of any n -input circuit α with $d(\alpha) \leq d$ and $c(\alpha) \leq c$ the circuit U simulates α on x_1, \dots, x_n . The outputs y_1, \dots, y_c of U correspond to the nodes $1, \dots, c$ of α . Furthermore, U is U_{BC} uniform and $d(U) = O(d)$ and $c(U) = O(c^4)$.*

5. Ruzzo's theorem. The constructions in Theorems 1 and 2 can be used to give an alternative proof of a theorem of Ruzzo [Ru81] which shows the invariance under different uniformity conditions of complexity classes defined by uniform circuit families. Slightly altering Ruzzo, let us define the *extended connection language* L_{EC} of a circuit family $\langle \alpha_n \rangle$ to be the set of (codes for) tuples (n, ν, π, g, l, r) , where n is in binary and (ν, π, g, l, r) is a tuple in our extended encoding $\hat{\alpha}_n$ of α_n . Then the family $\langle \alpha_n \rangle$ is U_E uniform if and only if some DTM can determine whether an arbitrary tuple (n, ν, π, g, l, r) is in L_{EC} in time $O(\log c(\alpha_n))$.

Every U_E uniform family is certainly U_{BC} uniform, and in fact U_E appears to be a much stronger condition than U_{BC} . Nevertheless, in [Ru81] it is proved that:

THEOREM 3 (Ruzzo). *Let $Z(n)$ and $T(n)$ be any functions such that $Z(n) \geq n$, $T(n) \geq \log^2 Z(n)$, and $\lceil \log_2 Z(n) \rceil, T(n)$ are computable by deterministic Turing machines given input n (in binary) in time $O(\log Z(n))$. If a language A is recognized by a U_{BC} uniform circuit family of size $Z(n)^{O(1)}$ and depth $O(T(n))$, then A is recognized by a U_E uniform circuit family with the same size and depth bounds.*

The main application of the above result is to the complexity classes NC^k , defined by the size and depth bounds $n^{O(1)}$ and $O(\log^k n)$ respectively. The theorem implies that for $k \geq 2$, NC^k remains the same whether the circuit families are required to be U_E or U_B uniform.

The notion of U_E uniform is interesting partly because alternating Turing machine space-time complexity classes have precise characterizations in terms of U_E uniform circuit size-depth complexity classes (see [Ru81, Thms. 3, 4]). In fact, Ruzzo uses alternating Turing machines to prove Theorem 3 above. We can use our universal circuit construction from Theorem 1 to give:

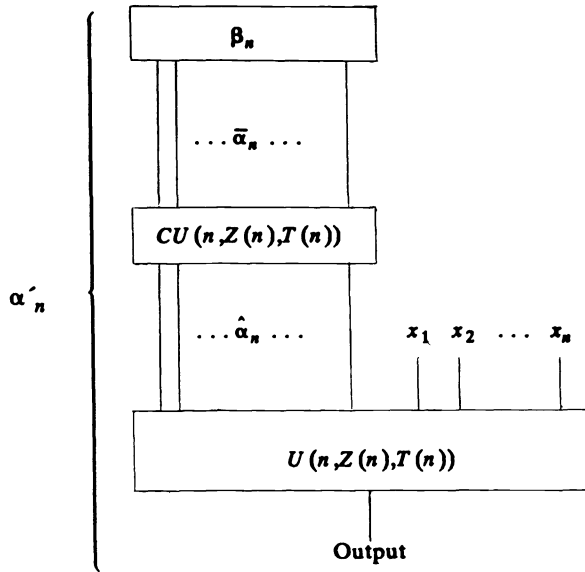
Proof of Theorem 3. First note that we can make U to be U_E uniform instead of just U_{BC} uniform. In this case, U_E uniform means that some DTM can determine in time $O(\log c)$ given a tuple $(n, c, d, \nu, \pi, g, l, r)$ with $c \geq n$ whether (ν, π, g, l, r) is in the extended encoding \hat{U} for $U(n, c, d)$. Thus h can be computed in time $O(\log c)$ from the binary length of $c+1$, and instead of taking $q = \lceil d/h \rceil$, it is sufficient to use the easily computed over-approximation $q = 2^e$, with $e = 1 + (\text{binary length of } d) - (\text{binary length of } h)$. The codes for the gates in U can be chosen to explicitly incorporate their position. For example, the binary number for a gate in the selector tree $S_{i,j,k}$ would include the indices i, j, k in binary and a bit string indicating the path from the root of the tree to the gate.

With similar care in numbering the gates, the conversion circuit CU in Theorem 2 can be made U_E uniform.

Now suppose $Z(n)$ and $T(n)$ are as in Theorem 3, and $\langle \alpha_n \rangle$ is a U_{BC} uniform circuit family with size and depth bounds $Z(n)^{O(1)}$ and $O(T(n))$ which recognizes language A . We construct a U_E uniform family $\langle \alpha'_n \rangle$ to recognize A as shown in Fig. 2.

The circuit α'_n has three stages. First the stage β_n computes the description $\tilde{\alpha}_n$, next the circuit $CU(n, Z(n), T(n))$ converts that description to $\hat{\alpha}_n$, and finally the universal circuit $U(n, Z(n), T(n))$ takes the description $\hat{\alpha}_n$ and x_1, \dots, x_n as inputs and simulates α_n . The circuit β_n has no inputs, but simulates the DTM which transforms 1^n to $\tilde{\alpha}_n$. The simulation applies a transitive closure circuit to the configuration-transition matrix of the DTM, as described in Theorem 2 of [Bo77]. Thus $c(\beta_n) = Z(n)^{O(1)}$ and $d(\beta_n) = O(\log^2 Z(n))$.

The conditions on $Z(n)$ and $T(n)$ ensure that all three stages can be made U_E uniform and that the bounds for the size and depth of α'_n are satisfied. \square

FIG. 2. Circuit α'_n recognizing language A.

Acknowledgments. We wish to thank the referees for careful comments and for pointing out additional references.

REFERENCES

- [Bo77] A. BORODIN, *On relating time and space to size and depth*, this Journal, 6 (1977), pp. 733-744.
- [Co81] S. A. COOK, *Towards a complexity theory of synchronous parallel computation*, L'Enseignement Mathématique, 1-2 (1981), pp. 99-124.
- [GP83] Z. GALIL AND W. PAUL, *An efficient general purpose parallel computer*, J. Assoc. Comput. Mach., 2 (1983), pp. 360-387.
- [Go78] L. GOLDSCHLAGER, *A unified approach to models of synchronous parallel machines*, Proc. 11th Annual ACM Symposium on Theory of Computing, 1978, pp. 89-94.
- [adH83] F. M. AUF DER HEIDE, *Efficient simulations among several models of parallel computers*, Manuscript, 1983.
- [Ho78] H. J. HOOVER, *Some topics in circuit complexity*, M.Sc. thesis and TR-139/80, Dept. Computer Science, Univ. Toronto, Toronto, Ontario, Canada, Dec. 1979.
- [Ru81] W. L. RUZZO, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365-383.
- [Va76] L. G. VALIANT, *Universal circuits*, Proc. 8th Annual ACM Symposium on Theory of Computing, 1976, pp. 196-293.

SEARCHING SEMISORTED TABLES*

HELMUT ALT† AND KURT MEHLHORN‡

Abstract. A “semisorted table” is a one-dimensional array containing n data, which are not necessarily sorted, but can appear in p different permutations of the ascending order. We consider the problem of searching in such a table without knowing, in which one of the p permutations the data are stored (SST). It is shown that any deterministic search algorithm for SST needs at least $\sqrt[p]{p}$ comparisons in the worst case. This lower bound is generalized to average case performance even for nondeterministic algorithms. Some examples are given where the lower bound is tight.

Key words. searching, implicit data structures, comparison trees

1. Introduction. This paper deals with searching semisorted tables (SST), by which we mean the following problem:

Let U be an infinite, totally ordered universe. Let Π be a set of permutations of $\{1, \dots, n\}$, and $p = |\Pi|$. Assume that n elements of U may be stored in an array A according to any permutation $\pi \in \Pi$ of their ascending order. How many comparisons are necessary to search for a given $x \in U$?

Note, that the restriction of our search problem to data stored in an array is not essential. In fact, even if they are stored in some arbitrary data structure, for a given problem instance the assignment of elements of the universe U to the memory locations of that structure will be unique and never will be changed. So we can give these locations a fixed numbering $1, \dots, n$, such that the statement that data are “stored according to the permutation π of their ascending order” is well defined.

Lower and upper bounds for the degenerate cases of the problem are well known:

If $p = 1$, i.e., the data can appear in only one fixed permutation, a (modified) binary search can be done and $\lceil \log n \rceil + 1$ comparisons are necessary and sufficient. If $p = n!$, i.e., data can appear in any permutation, linear search has to be done with n comparisons being necessary and sufficient. Our problem is to find lower bounds for any p somewhere between 1 and $n!$.

In § 2 a comparison tree argument will show that $\Omega(\sqrt[p]{p})$ is a lower bound in the worst case. This argument and, thus, the lower bound are then extended to *nondeterministic* algorithms, and finally to the *average* complexity of SST. The final section of the paper shows that these lower bounds are tight for some, but not for all cases.

Note, that our lower bound only depends on the number p of possible arrangements of data, and does not restrict the way to arrange them. One possible restriction is, for example, to consider only data structures, where all possible arrangements of data are compatible with a given partial order. Most of the standard comparison based data structures like all versions of binary search trees, sorted arrays etc. have that property, but others, like, e.g., rotated lists [MS], do not. For data structures satisfying the partial order restriction an exact lower bound for searching has been found in a recent paper by Linial and Saks [LS]. They show that any search algorithm for a given data structure has to make at least $\log N$ comparisons, where N is the number of ideals of the underlying partial order. That method can be applied to instances of SST which satisfy a given partial order and gives better lower bounds in some of these cases (cf. second example in § 4).

* Received by the editors July 5, 1983, and in revised form May 30, 1984.

† Department of Computer Science, The Pennsylvania State University, University Park, Pennsylvania 16802.

‡ Fachbereich 10, Universität des Saarlandes, 6600 Saarbrücken, West Germany.

The SST problem by itself seems interesting to us, but there also exists one seemingly important application: Finding lower bounds for a case of partial match retrieval in implicit data structures (cf. [MS], [R]), as has been done in [AMM].

The deterministic worst case lower bound for the SST problem was investigated concurrently and independently by S. Cook ([B]).

The model of computation used here requires that comparisons are done only between the element x searched for and some array element. It is an interesting open question, if the lower bounds still hold, if comparisons between different array elements are allowed, as well.

2. A worst case lower bound for SST. Let us now define the SST problem more formally:

Let U be an infinite, totally ordered universe, let $S = \{x_i | 1 \leq i \leq n\} \subseteq U$ where $x_1 < x_2 < \dots < x_n$. An array $A[1:n]$ is used to store S . If π is a permutation of $\{1, \dots, n\}$ then we say S is stored according to π , if and only if $A[\pi(i)] = x_i$.

Let Π be a set of permutations of $\{1, \dots, n\}$. Then SST(Π) is the following problem:

input: some $x \in U$

problem: decide, if $x \in S$ under the precondition that S is stored according to some $\pi \in \Pi$ (which one is not known though).

The following *model of computation* will be used:

Algorithms are based on comparisons of the form $x ? A[i]$ where x is the element searched for and $i \in \{1, \dots, n\}$. So any such algorithm can be illustrated by a comparison tree T , each node of which is labelled by some $i \in \{1, \dots, n\}$, meaning that at this point x is compared to $A[i]$. If the outcome of that comparison is " $<$ " (" $>$ ") the algorithm proceeds using the left (right) subtree, if it is " $=$ " the algorithm halts giving a positive answer.

Certainly a computation (= path starting from the root) in T only depends on the permutation of Π in which the data are stored and the relative position of x within the data but not on the particular choice of data. (For example, a search for 3 in a table containing 4 3 5 1 2 in that order would take the same path in the comparison tree as searching for 5 in the table 7 5 9 2 3.)

Now a lower bound for SST(Π) will be shown, depending on the number p of elements of Π and on the number n of data.

THEOREM 1. *For all $n \in \mathbb{N}$, Π any set of p permutations of $\{1, \dots, n\}$, any algorithm solving SST(Π) makes at least $\sqrt[p]{p}$ comparisons in the worst case.*

Proof of Theorem 1. Call a sequence i_1, \dots, i_n ($1 \leq k \leq n$) *valid* if there exists a $\pi \in \Pi$ such that $\pi(j) = i_j$ ($1 \leq j \leq k$), i.e., if the k smallest elements may be stored in table positions i_1, \dots, i_k . Denote by ε the empty sequence over $\{1, \dots, n\}$ and let it be valid by definition. Clearly a permutation π is in Π exactly if $\pi(1), \dots, \pi(n)$ is valid. Let T be a comparison tree solving SST(Π) and let s be its depth, i.e., $s+1$ is the maximum number of comparisons for a search.

We will show that the number of permutations for which T works correctly is at most $(s+1)^n$.

LEMMA 1. *Any valid sequence of length k ($0 \leq k < n$) can be extended in at most $s+1$ ways to a valid sequence of length $k+1$.*

Proof. Let i_1, \dots, i_k (the empty sequence ε if $k=0$) be a valid sequence and $\pi \in \Pi$ some permutation which makes it valid (i.e., $\pi(j) = i_j$, $1 \leq j \leq k$; any $\pi \in \Pi$ will do in the case $k=0$). Assume that data are stored in the table according to π , i.e., $A[\pi(i)] = x_i$. In particular the k smallest elements are stored in ascending order in $A[i_1], \dots, A[i_k]$.

Consider the path in T which is taken by a search for some $x \in U$ with $x_k < x < x_{k+1}$, i.e., $x \notin S$, $x > A[i]$ if $i \in \{i_1, \dots, i_k\}$, $x < A[i]$ otherwise ($x < A[i]$ for all i if $k=0$). Such an x exists w.l.o.g. since U is infinite and the algorithm only depends on the permutation in which data are stored (i.e., we can assume that there are "gaps" between the data stored in the table.) This path now depends only on i_1, \dots, i_k (is unique if $k=0$). In fact it can be described by the rule: start in the root, take the $>$ -branch if the current node is labelled by some i_j ($1 \leq j \leq k$), take the $<$ -branch otherwise (in the case $k=0$ always take the $<$ -branch).

Let $L = \{l_1, \dots, l_t\}$ be the set of labels encountered on the above path which are not in $\{i_1, \dots, i_k\}$, i.e., on the path they are the ones, whose left child is visited next (see Fig. 1).

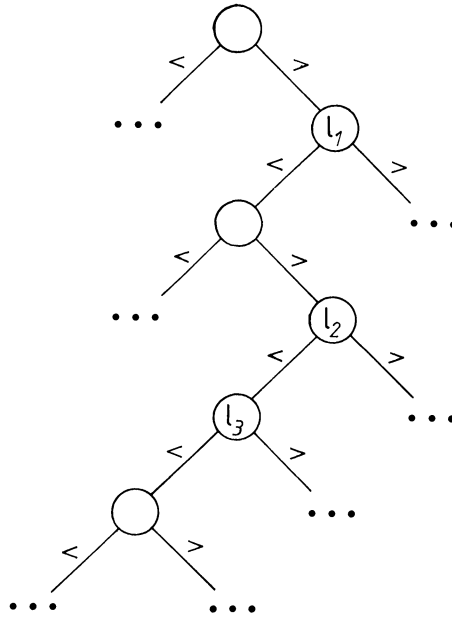


FIG. 1

Let $i_{k+1} = \pi(k+1)$, i.e., $A[i_{k+1}]$ contains x_{k+1} .

We claim: $i_{k+1} \in L$. Assume otherwise. Then x would not be compared to x_{k+1} and the outcome of all comparisons would be the same if we traverse T with x_{k+1} instead of x . So the same path would be taken and T would give the same answer for x_{k+1} as for x , namely that it is not in S , a contradiction. Now L depends on i_1, \dots, i_k only and it has $\leq s+1$ elements, since they are some of the labels of one particular path in T . So the valid sequence i_1, \dots, i_k can be extended in at most $s+1$ ways to a valid sequence i_1, \dots, i_k, i_{k+1} , which proves Lemma 1.

Now, applying Lemma 1 repeatedly, we have that there are at most $(s+1)^k$ valid sequences of length k ($1 \leq k \leq n$). Since all permutations in Π give different valid sequences of length n we have $P \leq (s+1)^n$, i.e., the number of comparisons $s+1 \geq \sqrt[n]{P}$, which proves Theorem 1.

3. Nondeterministic and average case lower bounds. In this section we want to show that the lower bound of § 2 also holds for nondeterministic algorithms and for the average search time.

A nondeterministic comparison based search, in one step of the computation compares, just like a deterministic one, the element z searched for to some table entry $A[i]$. But for any outcome “ $<$ ” or “ $>$ ” it may have several choices to which entry to compare x next. So the comparison tree is not necessarily binary any more. The answers the algorithm gives at the end of a computation may be “yes” ($x \in S$) “no”, or “undetermined”. We require the algorithm to be complete and consistent: If $x \in S$ there must not be any computation for x answering “no” and there must be at least one computation answering “yes”. If $x \notin S$ there must not be any computation answering “yes” but at least one answering “no”.

Of course, if $x \in S$, this can be determined by a nondeterministic algorithm in constant time: Choose nondeterministically any table position i , answer “yes” if $x = A[i]$. We show that the same lower bound as in § 2 holds in the nondeterministic case for $x \notin S$.

THEOREM 2. *For all $n \in \mathbb{N}$, Π any set of p permutations of $\{1, \dots, n\}$, any algorithm solving SST (Π) makes at least $\sqrt[p]{p}$ comparisons in the worst case, i.e., for any set $S \subset U$, $|S| = n$ there exists an $x \in U$ such that the shortest computation searching for x in S and leading to a “yes” or “no”-answer has length $\geq \sqrt[p]{p}$.*

Proof. The proof is essentially the same as the one for Theorem 1.

Assume $0 \leq k < n$, i_1, \dots, i_k is a valid sequence, $\pi \in \Pi$ a permutation making it valid, and data are stored according to π . Now again any search for $x \in U$ with $x_k < x < x_{k+1}$ leading to a “no” answer has to compare x with x_{k+1} . So only the positions appearing on every path in the comparison tree leading to a “no” answer for x can be used to store x_{k+1} . So if s_{k+1} is the shortest length of such a path, there are at most s_{k+1} possibilities to extend the above valid sequence to one of length $k+1$. So there are at most s_1, s_2, \dots, s_n valid sequences of length n , we need at least p to make the algorithm work for all permutations in Π . So there exists and $i \in \{1, \dots, n\}$ with $s_i \geq \sqrt[p]{p}$. So the shortest search for an x with $x_{i-1} < x < x_i$ makes at least $\sqrt[p]{p}$ comparisons.

Next it will be shown that the lower bound of Theorems 1 and 2 even holds for the average case of unsuccessful searches, even for nondeterministic algorithms. Any unsuccessful search can be associated with an interval $(x_i, x_{i+1}) = \{y \in U \mid x_i < y < x_{i+1}\}$, namely the one containing the element x searched for.

We assume that all these intervals have the same access-probability. Let Π be a set of permutations of $\{1, \dots, n\}$, for which we consider SST (Π). To Π a tree T_Π is associated in the following way:

The nodes of T_Π except the root, are labelled with numbers of $\{1, \dots, n\}$. There are exactly $p = |\Pi|$ leaves representing the permutations in Π . Call the leaf associated with $\pi \in \Pi$, l_π .

For any $\pi \in \Pi$ the sequence of labels on the path from the root to l_π exactly corresponds to the permutation π . (So T_Π has height n .)

As an example consider: $n = 7$, Π the set of all permutations of $\{1, \dots, 7\}$ not displacing 1, \dots , 4. So $p = 3! = 6$. A suitable search strategy is to do binary search on the first 4 positions and linear search on the others. The corresponding search tree (a square denotes an unsuccessful search), is shown in Fig. 2.

The tree T_Π , in this case, is shown in Fig. 3.

Furthermore, by definition, the labelling i_1, \dots, i_k ($k \geq 0$) of any path in T_Π starting in the root, is a valid sequence. By the proofs of Theorems 1 and 2 a valid sequence of length k ($0 \leq k < n$) can only be extended to one of length $k+1$ by using the nodes on a certain path in the comparison tree. This path corresponds to an unsuccessful search for an element $x \in U$ with $x_k < x < x_{k+1}$. So the outdegree of the node in T_Π corresponding to the valid sequence i_1, \dots, i_k is a lower bound on the

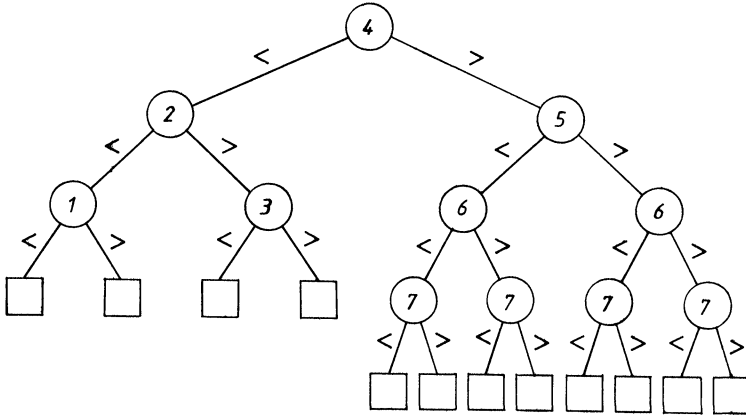


FIG. 2

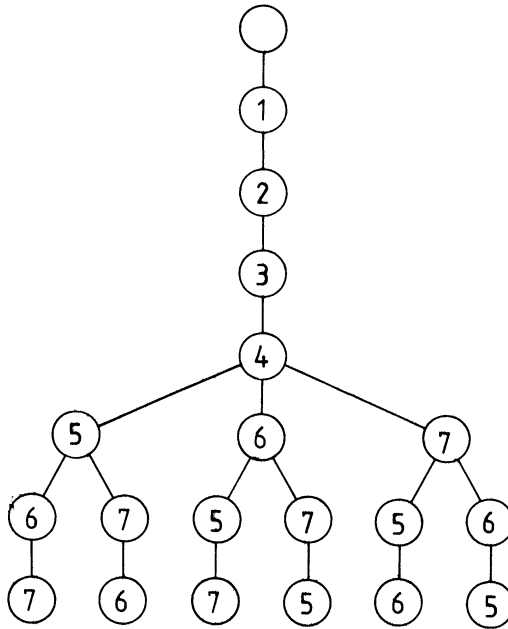


FIG. 3

number of comparisons for the above unsuccessful search, assuming that the elements x_1, \dots, x_k are stored in positions i_1, \dots, i_k .

For a permutation $\pi \in \Pi$ let $c(\pi)$ be the average number of comparisons for searches for elements $x \in U$ with $x_k < x < x_{k+1}$ ($0 \leq k < n$) assuming that all "gaps" are equally likely.

For a tree T and any leaf l of T let $P_T(l)$ be the set of ancestors of l . Then by the considerations above

$$(1) \quad c(\pi) \cong \frac{1}{n} \sum_{v \in P_{T_\Pi}(l_\pi)} \text{deg}(v).$$

So, additionally assuming that all permutations are equally likely, the average number of comparisons for an unsuccessful search is

$$\frac{1}{p} \sum_{\pi \in \Pi} c(\pi).$$

This expression is because of (1) greater than or equal $s(T_\Pi)$ where for any tree T with p leaves we define

$$s(T) = \frac{1}{p} \sum_{l \text{ leaf of } T} \frac{1}{\text{depth}(l)} \sum_{v \in P_T(l)} \text{deg}(v)$$

which is the average outdegree of T 's nodes weighted according to their number of descendants.

CLAIM. For any tree T with p leaves all having depth $h \geq 1$:

$$p \leq s(T)^h.$$

Proof. (by induction on h). If $h = 1$, the tree has the form of Fig. 4.

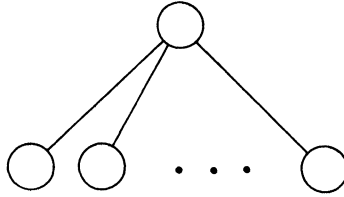


FIG. 4

Assume that the root has outdegree k . Then $p = s(T) = k$ and thus, the claim is true.

For the *inductive step* assume that we have a tree T of height $h + 1$ and that its root v_0 has outdegree k . Let T_1, \dots, T_k be the subtrees of the root and p_1, \dots, p_k their numbers of leaves respectively (see Fig. 5).

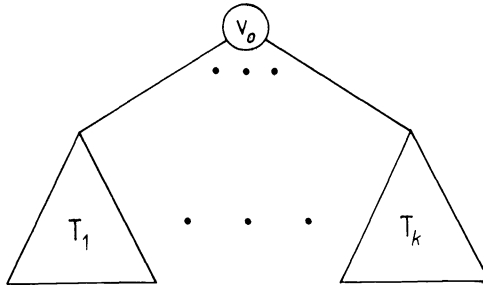


FIG. 5

Then

$$\begin{aligned} s(T) &= \frac{1}{p(h+1)} \sum_{l \text{ leaf of } T} \sum_{v \in P_T(l)} \text{deg}(v) \\ (2) \quad &= \frac{1}{p(h+1)} \sum_{i=1}^k \sum_{l \text{ leaf of } T_i} \left[\sum_{v \in P_{T_i}(l)} \text{deg}(v) + \text{deg}(v_0) \right] \\ &= \frac{1}{p(h+1)} \sum_{i=1}^k [p_i k + p_i h s(T_i)] \end{aligned}$$

since $\text{deg}(v_0) = k$ and by definition of $s(T_i)$.

We have to show

$$p \leq s(T)^{h+1}$$

or, because of (2)

$$p^{h+2} \leq \left[\frac{1}{h+1} \left(kp + h \sum_{i=1}^k p_i s(T_i) \right) \right]^{h+1}.$$

Since by inductive hypothesis $p_i \leq s(T_i)^h$ and hence $s(T_i) \geq (p_i)^{1/h}$ it suffices to show

$$p^{h+2} \leq \left[\frac{1}{h+1} \left(kp + h \sum_{i=1}^k p_i^{1+1/h} \right) \right]^{h+1}.$$

Now note that

$$\sum_{i=1}^k p_i^{1+1/h}$$

is minimal (subject to the constraint $\sum_{i=1}^k p_i = p$) if $p_i = p/k$ for all i .

Therefore, it suffices to show:

$$p^{h+2} \leq \left[\frac{1}{h+1} (kp + hk(p/k)^{1+1/h}) \right]^{h+1}$$

or

$$(3) \quad p \leq \left[\frac{1}{h+1} (k + h(p/k)^{1/h}) \right]^{h+1}.$$

The left-hand side of this inequation does not depend on k . Therefore, the proof is finished, if it is possible to prove inequation (3) for that k , for which the right-hand side is minimal. So, consider

$$f(k) = k + h(p/k)^{1/h}.$$

Then, the derivative

$$f'(k) = 1 - p^{1/h} \cdot k^{-1/(h+1)}$$

equals 0, if and only if

$$k^{h+1} = p,$$

i.e.,

$$k = p^{1/(h+1)}.$$

So, by (3), it suffices to show

$$p \leq \left[\frac{1}{h+1} (p^{1/(h+1)} + h(p^{1-1/(h+1)})^{1/h}) \right]^{h+1}.$$

The right-hand side equals

$$\left[\frac{1}{h+1} (p^{1/(h+1)} + hp^{1/(h+1)}) \right]^{h+1} = [p^{1/(h+1)}]^{h+1} = p,$$

which finishes the proof of the claim.

Applying the claim to the tree T_{Π} gives

$$p^{1/n} \leq s(T_{\Pi}).$$

So, since $s(T_\Pi)$ is a lower bound on the average number of comparisons for unsuccessful searches in SST (Π), we have the following result:

THEOREM 3. *For all $n \in N$, Π any set of p permutations of $\{1, \dots, n\}$, any nondeterministic algorithm solving SST (Π) makes at least $\sqrt[p]{p}$ comparisons on the average.*

4. Tightness of the lower bounds. Theorems 1 through 3, of course, only give nontrivial lower bounds, if the number p of possible permutations is "sufficiently high". In fact, p needs to grow more than exponentially in n , in order to have $\sqrt[p]{p}$ not bounded above by a constant.

One case in which the lower bounds are tight, is $p = n!$. So every possible permutation is allowed and by linear search we have $O(n)$ worst case and average algorithms. On the other hand Theorems 1 and 3 give an $\Omega(\sqrt[p]{n!}) = \Omega(n)$ lower bound, showing that linear search is asymptotically optimal. (Of course, this lower bound can be shown by a much easier argument.) Linear search can be generalized in the following way:

Let $s = \Omega(\log n)$ and assume w.l.o.g. that s divides n . Let Π be the set of all permutations of $\{1, \dots, n\}$ obtained in the following way:

$\{1, \dots, n\}$ is broken up into n/s blocks of size s . The first block contains $\{1, \dots, s\}$, the second one contains $\{s+1, \dots, 2s\}$ etc. Within a block the order of elements is arbitrary.

So $p = (s!)^{n/s}$ and the lower bounds from Theorems 1 and 3 are

$$\sqrt[p]{p} = (s!)^{1/s} = \Theta(s).$$

On the other hand, we have the following algorithm to solve SST (Π):

Do a binary search on positions $1, s+1, 2s+1$, etc., i.e., the first positions of the blocks in order to find the only two blocks which possibly may contain the element searched for. Then do a linear search in these blocks.

This algorithm performs in the worst case and on the average

$$\begin{aligned} O(\log(n/s) + s) &= O(\log n - \log s + s) \\ &= O(s) \end{aligned}$$

comparisons since $s = \Omega(\log n)$. So the lower bound given by Theorems 1 through 3 are tight in all these cases. Note, that while s ranges from $\log n$ to n , p ranges from $\Theta(2^{n \log \log n})$ to $\Theta(n!)$.

But not in every case are the lower bounds of this paper tight. As a counter example let Π be the set of all permutations, where the elements $\{1, \dots, n/2\}$ are in their original positions, the other ones are permuted arbitrarily. Clearly

$$p = (n/2)!$$

and

$$\sqrt[p]{p} = \sqrt[p]{(n/2)!} = \Theta[(n/2)^{1/2}] = \Theta(\sqrt{n}).$$

On the other hand it is clear, that no algorithm can do better than $\Theta(n)$ even on the average, because on the second half of the data a linear search has to be performed.

REFERENCES

- [AMM] H. ALT, K. MEHLHORN AND J. I. MUNRO, *Partial match retrieval in implicit data structures*, Inform. Proc. Lett., 19 (1984), pp. 61-65.
 [B] A. BORODIN, Oral communication.

- [LS] N. LINIAL AND M. E. SAKS, *Information bounds are good for search problems on ordered data structures*, Proc. 24th IEEE Symposium on Foundations of Computer Science, Tucson, AZ, 1983, pp. 473–475.
- [M] J. I. MUNRO, *A multikey search problem*, Proc. 17th Annual Allerton Conference, October 1979.
- [MS] J. I. MUNRO AND H. SUWANDA, *Implicit data structures*, Proc. 11th Annual ACM Symposium on Theory of Computing, May 1979, pp. 108–117.
- [R] R. L. RIVEST, *Partial match retrieval algorithms*, this Journal, 5 (1976), pp. 19–50.
- [Y] A. C. YAO, *Should tables be sorted?*, Proc. IEEE Symposium on Foundations of Computer Science, 1978, pp. 22–27.

ON APPROXIMATION ALGORITHMS FOR #P*

LARRY STOCKMEYER†

Abstract. The theme of this paper is to investigate to what extent approximation, possibly together with randomization, can reduce the complexity of problems in Valiant's class #P. In general, any function in #P can be approximated to within any constant factor by a function in the class Δ_2^P of the polynomial-time hierarchy. Relative to a particular oracle, Δ_2^P cannot be replaced by Δ_1^P in this result. Another part of the paper introduces a model of random sampling where the size of a set X is estimated by checking, for various "sample sets" S , whether or not S intersects X . For various classes of sample sets, upper and lower bounds on the number of samples required to estimate the size of X are discussed. This type of sampling is motivated by particular problems in #P such as computing the size of a backtrack search tree. In the case of backtrack search trees, a sample amounts to checking whether a certain path exists in the tree. One of the lower bounds suggests that such tests alone are not sufficient to give a polynomial-time approximation algorithm for this problem, even if the algorithm can randomize.

Key words. #P, approximation algorithms, probabilistic algorithms, computational complexity, relativization

1. Introduction. There are several computational problems which can be formulated as problems of counting the number of objects having a certain property. Valiant [17] has introduced the class #P which includes a variety of counting problems such as counting the number of perfect matchings in a graph, computing the permanent of a matrix [17], finding the size of a backtrack search tree [8], and computing the probability that a network remains connected when links can fail with a certain probability [18]. For many problems in #P, including those just mentioned, no polynomial-time algorithms are known. Indeed, it is not known whether these problems can be solved at any fixed level of the polynomial-time hierarchy [1], [15]. The obvious algorithm of explicitly counting the number of objects is not efficient since the number of objects grows exponentially in the size of the input. For example, in computing the permanent of an $n \times n$ matrix with 0-1 entries a_{ij} , the objects are the $n!$ permutations on $\{1, \dots, n\}$, and the permanent is equal to the number of permutations π with $\prod_{i=1}^n a_{i,\pi(i)} = 1$. The best known general upper bound is that any #P problem can be solved within polynomial space.

Two known approaches for reducing the computational complexity of problems are approximation and randomization. Several NP-complete optimization problems, which are apparently intractable to solve exactly, can be solved in polynomial time if one is willing to settle for a solution within a constant factor of the optimum [3, Chapter 6]. Randomization, that is, allowing algorithms to make decisions based on the outcomes of random coin tosses but requiring that the probability of error be small, has been useful in solving certain number theoretic problems faster than the best known deterministic algorithms [12], [14]. The purpose of this paper is to investigate to what extent approximation, or approximation together with randomization, can reduce the complexity of counting problems.

Before proceeding to outline the remainder of the paper, we should explain informally our definitions of "approximation" and "randomization." If $f(x)$ is an

* Received by the editors October 25, 1983, and in revised form July 5, 1984. Portions of this paper have been reprinted, with permission, from *The complexity of approximate counting* by L. Stockmeyer, appearing in the Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 1983, pp. 118-126, © 1983, Association for Computing Machinery, Inc.

† Computer Science Department, IBM Research Laboratory K51-281, 5600 Cottle Road, San Jose, California 95193.

integer valued function and r is a constant with $r > 1$, then a probabilistic algorithm approximates $f(x)$ to within the factor r if

- (1) The value produced by the algorithm is between $f(x)/r$ and $r \cdot f(x)$ with probability $\geq \frac{1}{2} + \epsilon$ for some fixed $\epsilon > 0$.

(In the case of #P problems, each input x determines a set X_x such that membership in X_x can be checked in polynomial time, and $f(x) = |X_x|$.)

Regarding previous work, Knuth [8] gives a polynomial-time probabilistic algorithm for estimating the size of a backtrack search tree where the expected value of the output of the algorithm is exactly the size of the tree. Lovasz does the same for computing the permanent of a 0-1 matrix. However, for both of these algorithms the variance is very large and the algorithms do not satisfy (1). By analyzing the variance of Lovasz's algorithm and related algorithms, Karmarkar, Karp, Lipton and Luby [10] give probabilistic algorithms for the 0-1 permanent which satisfy (1) and whose running times, although not polynomial in n , are better than the best known deterministic algorithm for computing the permanent exactly.

One well-known approach for estimating the size of a set X is random sampling. Assuming that X is a subset of a finite universe U of known size, one would randomly choose elements of U , compute the fraction γ of chosen elements which belong to X , and give $\gamma|U|$ as an estimate for the size of X . Goldschlager [5] suggests this as an approximation method for computing the permanent. Since $|U|$ is typically exponential in n , this "singleton sampling" runs in time polynomial in n and satisfies (1) only if $|X| \geq |U|/n^k$ for some constant k . For example, if $|X|$ is small, say $|X| \approx |U|^{1/2}$, then singleton sampling would require more than $|U|^{1/2}$ samples to reach a good estimate.

In § 2 we define and study a class of restricted, but very natural, probabilistic sampling methods motivated by the particular counting problems mentioned above. Instead of "singleton sampling" the algorithm is allowed to sample a large set $S \subseteq U$ in one step; the information returned from the sample is whether $S \cap X = \emptyset$. Depending on the application, there might be restrictions on the types of subsets S which can be sampled (some such restrictions are detailed in § 2), so we are interested in how the complexity, measured as the number of samples, depends on the types of samples which are allowed. Two motivating examples are given in § 2. The main technical results of § 2 establish, for two particular classes of sample sets, lower bounds on the number of samples required to approximate $|X|$. One of these lower bounds suggests that this type of sampling will not give a polynomial-time algorithm for approximating the size of a backtrack search tree.

In § 3 we attempt to classify the complexity of approximately computing functions in #P. The classification is done in terms of the polynomial-time hierarchy (for short, P-hierarchy) [15]. For any function in #P, it is shown that a machine at the Δ_3^P level of the P-hierarchy can compute approximate solutions that are accurate to within the factor $(1 + \epsilon n^{-d})$ for any fixed constants d and $\epsilon > 0$. (As noted above, computing #P functions exactly is not known to be possible at any finite level of the P-hierarchy.) The proof is based on the technique, introduced recently by Sipser [13], of estimating the size of a set by hashing the set into a second set of known size. In the relativized world, cf. [2], we also can give a corresponding lower bound to the general problem of approximately computing #P functions: there is an oracle A and a function in #P^A (#P relativized to A) such that the function cannot be approximately computed to within any constant factor by a machine at the $\Delta_2^{P,A}$ level of the A -relativized P-hierarchy. The Δ_3^P upper bound relativizes to an arbitrary oracle. (See [2], [15] for definitions of oracle machines and the relativized polynomial-time hierarchy.)

In § 4 we give a relativization result that complements a recent result of Sipser and Gács [13] that BPP is contained in the second level of the P-hierarchy. Recall that BPP is the class, as defined by Gill [4], of languages accepted by probabilistic polynomial-time Turing machines with error probability $\leq \frac{1}{2} - \epsilon$ for some fixed $\epsilon > 0$. Sipser and Gács show that

$$\text{BPP} \subseteq \Sigma_2^P \cap \Pi_2^P$$

and this result relativizes to an arbitrary oracle. Given this inclusion, it is natural to ask whether BPP can be placed even lower in the P-hierarchy. The next lowest class below $\Sigma_2^P \cap \Pi_2^P$ is Δ_2^P , the class of languages accepted by deterministic polynomial-time Turing machines using oracles in NP. We show that there is an oracle A such that

$$\text{BPP}^A \not\subseteq \Delta_2^{P,A}.$$

Thus, any attempt to prove $\text{BPP} \subseteq \Delta_2^P$ cannot proceed by a proof that relativizes to an arbitrary oracle. The connection between this result and the rest of the paper is that there is an oracle such that a BPP machine can approximately count the number of strings of a given length that are in the oracle whereas Δ_2^P machines cannot.

2. Intersection-samples.

2.1. Definitions and motivation. If U is a set, 2^U denotes the set of subsets of U , and $|U|$ denotes the cardinality of U . For integer $n \geq 0$, $\{0, 1\}^n$ denotes the set of binary words of length n and $\{0, 1\}^{\leq n}$ denotes the set of binary words of length n or less. If x is a word, $|x|$ denotes the length of x .

It will be convenient first to define a framework which captures several counting problems. In this framework, a problem is specified by a finite set (*universe*) U and a collection of *inputs* $\mathcal{X} \subseteq 2^U$. Given an arbitrary input $X \in \mathcal{X}$, we want to approximate $|X|$ to within a constant factor. Let $N = |U|$. For the situations which motivate this question, N is so large that the count cannot be done explicitly. For example, $N = 2^n$ where n is the natural measure of the “size” of the problem. We are also given a class of sets, the *samples*,

$$\mathcal{C} \subseteq 2^U.$$

For an arbitrary $S \in \mathcal{C}$, a unit cost operation is computing the predicate

$$\text{INT}(X, S) = \begin{cases} 1 & \text{if } S \cap X \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases}$$

For what types of sample classes \mathcal{C} can a probabilistic algorithm with running time polynomial in n ($= \log N$) compute an estimate to $|X|$ accurate to within a constant factor? Before giving some answers, we should first give some motivation. In particular, we must justify why $\text{INT}(X, S)$ can be viewed as a unit cost operation even when S is a large set. (Since we are only interested in distinguishing polynomial from nonpolynomial running times, any operation which can be done within time polynomial in n is viewed as a “unit cost” operation.)

1. *Estimating the size of a backtrack search tree.* Knuth [8] has considered the problem of estimating the size of a tree X where X is described implicitly by a polynomial-time backtrack search procedure. That is, given the name u of a node of X , the search procedure tells us which sons of u , if any, are in X . Let n be a known upper bound on the height of X , and to simplify the notation assume that X is binary. Imagine that X is embedded in F , the full binary tree of height n . The nodes of F are labeled by strings in $\{0, 1\}^{\leq n}$: the root is labeled λ , and the sons of u are labeled $u0$

and $u1$. Thus, the universe is $U_{tree} = \{0, 1\}^{\leq n}$. In this example, since the input $X \subseteq U$ is a tree with the same root as F , X must satisfy the following *tree property*:

if $u \in X$ then $w \in X$ for all prefixes w of u .

Let \mathcal{X}_{tree} be the class of sets X satisfying the tree property. The *tree size problem* is

$$\mathcal{P}_{tree} = (U_{tree}, \mathcal{X}_{tree}).$$

(We have actually defined a family of problems for $n \geq 1$. For simplicity in this example and the next, the index n has been suppressed.)

For w an arbitrary node of F , we can determine in time polynomial in n whether the full subtree of F rooted at w contains a node of X simply by attempting to follow the search procedure from the root of F to w . Thus the relevant class of samples is the class of *subtree samples*:

$$\mathcal{C}_{subtree} = \{S_w \mid w \in \{0, 1\}^{\leq n}\}$$

where

$$S_w = \{u \in \{0, 1\}^{\leq n} \mid w \text{ is a prefix of } u\}$$

is the full subtree of F rooted at w . If we view the backtrack search procedure as a “black box,” i.e., if the only way we can access the procedure is by giving it inputs and looking at its outputs, then subtree samples are, intuitively, the most general way of accessing the tree X .

2. *Counting perfect matchings.* For some #P-complete problems such as counting the number of perfect matchings in a graph (equivalently, for bipartite graphs, computing the permanent of a 0–1 matrix), the corresponding existential question, does there exist a perfect matching, can be solved in polynomial time [9]. Can this fact be used to estimate the number of perfect matchings in polynomial time? We do not have a definitive answer to this question, but we have some preliminary evidence on the pessimistic side. The problem fits the general framework as follows. Suppose we want to solve this problem for graphs with n vertices. Let $m = n(n-1)/2$ and let $E = \{e_1, \dots, e_m\}$ be the edges of K_n , the complete graph on n vertices. In this case the universe is $U_{word} = \{0, 1\}^m$. A particular graph G (viewed as a subgraph of K_n) defines a subset X_G of U_{word} as follows:

if $u = u_1 u_2 \dots u_m \in \{0, 1\}^m$, then

$$u \in X_G \text{ iff } \{e_j \mid u_j = 1\} \text{ is a perfect matching in } G.$$

The ability to check the existence of perfect matchings in subgraphs of G allows us to compute a variety of intersection samples as follows. Given $E_{in}, E_{out} \subseteq E$ with $E_{in} \cap E_{out} = \emptyset$, in polynomial time we can compute the predicate $P(G, E_{in}, E_{out})$ defined to be *true* iff G has a perfect matching $M \subseteq E$ with $E_{in} \subseteq M$ and $E_{out} \cap M = \emptyset$. Specifically, if E_{in} contains an edge not in G or if E_{in} contains two edges with a common endpoint then $P(G, E_{in}, E_{out})$ is false. Otherwise, remove from G all edges in E_{in} and E_{out} and all edges incident on an endpoint of an edge in E_{in} . Then $P(G, E_{in}, E_{out})$ iff the resulting subgraph has a perfect matching. Note that $P(G, E_{in}, E_{out})$ is really an intersection sample in disguise. Specifically, if we define $w = w_1 \dots w_m$ by setting $w_j = 1$ for all $e_j \in E_{in}$, $w_j = 0$ for all $e_j \in E_{out}$, and $w_j = *$ otherwise, then

$$P(G, E_{in}, E_{out}) = \text{INT}(X_G, Q_w)$$

where

$$Q_w = \{u \in \{0, 1\}^m \mid u \text{ matches } w\}.$$

Here, * is a “don’t care” symbol that matches 0 and 1. Thus the relevant class of samples, the *partial match samples*, is

$$\mathcal{C}_{pm} = \{Q_w \mid w \in \{0, 1, *\}^m\}.$$

Let $\mathcal{X}_{\text{all}} = \{X \mid X \subseteq U_{\text{word}}\}$. The *word counting problem* is

$$\mathcal{P}_{\text{word}} = (U_{\text{word}}, \mathcal{X}_{\text{all}}).$$

An obvious weakness in this definition is the choice of the class \mathcal{X} of inputs as all subsets of U_{word} rather than all X of the form X_G for some graph G . The reason for this choice is discussed below.

Another #P-complete problem with a polynomial-time existential question is: given a graph G and vertices s and t , count the number of subgraphs H of G for which there is a path from s to t in H [18] (this problem arises in network reliability). In this case, a partial match sample asks whether there is a path from s to t in a certain subgraph of G .

Our formal model of computation in this section is a variation of the probabilistic decision tree studied previously by Manber and Tompa [11]. Fix some counting problem \mathcal{P} consisting of a universe U and a class of inputs \mathcal{X} , and fix a class of samples \mathcal{C} . Each internal node of the decision tree is either a *sample node* or a *randomizing node*. A sample node is labeled by a set $S \in \mathcal{C}$ and has two branches that are taken depending on the outcome of $\text{INT}(X, S)$ where X is the particular input. A randomizing node has any number d of branches; each branch is taken with probability $1/d$. Each leaf is labeled by an integer between 0 and N , where $N = |U|$. A decision tree *solves* \mathcal{P} to within the factor r with error probability δ if, for any input $X \in \mathcal{X}$, the tree reaches a leaf with label between $|X|/r$ and $r|X|$ with probability $\geq 1 - \delta$. The *sample height* of the tree is the maximum number of sample nodes along any path from the root to a leaf. As Manber and Tompa point out, a lower bound on sample height implies, to within a constant factor, a lower bound on the expected number of samples (expectation with respect to the choices made at randomizing nodes), since branches which are much longer than the expected value can be pruned while increasing the error probability only slightly.

DEFINITION. $H(N, \mathcal{P}, \mathcal{C}, r, \delta)$ is the minimum sample height of a probabilistic decision tree, using only samples from \mathcal{C} , which solves \mathcal{P} to within the factor r with error probability δ , where N is the size of the universe of \mathcal{P} .

Throughout the rest of § 2, r and δ are fixed constants with $r > 1$ and $0 < \delta < \frac{1}{2}$.

2.2. Statements of results and discussion. In the case of subtree samples, the following lower and upper bounds are proved:

$$H(N, \mathcal{P}_{\text{trees}}, \mathcal{C}_{\text{subtree}}, r, \delta) = \Omega(N^{1/2}),$$

$$H(N, \mathcal{P}_{\text{trees}}, \mathcal{C}_{\text{subtree}}, r, \delta) = O((N \log N)^{1/2}).$$

(Here and subsequently, the constants implicit in the O - and Ω -notations depend on r and δ .) Knuth [8] has reported success in practice with a simple $O(\log N)$ time probabilistic procedure for estimating the size of a backtrack search tree. The $\Omega(N^{1/2})$ lower bound is not meant to contradict this, but it does suggest that the success must be linked to the properties of trees generated by particular backtrack search procedures. It should also be noted that the lower bound $\Omega(N^{1/2})$ is proved when the input X is

further restricted to a subset \mathcal{X}' of $\mathcal{X}_{\text{tree}}$. For $1 \leq d \leq n$, let $X(d, 0)$ be the tree whose nodes are $\{0, 1\}^{\leq d}$. For $1 \leq k \leq 2^d$, let $X(d, k)$ be the tree $X(d, 0)$ together with the full subtree of height $n - d$ attached to the k th leaf of $X(d, 0)$. Then \mathcal{X}' is the set of trees $X(d, k)$ for $1 \leq d \leq n$ and $0 \leq k \leq 2^d$. It is relevant that the input can be restricted to \mathcal{X}' because the trees in \mathcal{X}' can actually be generated by backtrack search procedures of size $O(n)$, whereas just naming trees in $\mathcal{X}_{\text{tree}}$ requires names of length 2^n .

In the case of partial match samples we show that

$$H(N, \mathcal{P}_{\text{word}}, \mathcal{C}_{\text{pm}}, r, \delta) = \Omega(N^{1/5}).$$

Since N is exponential in the size of the graph, the $\Omega(N^{1/5})$ lower bound suggests that this type of sampling does not give an efficient way of estimating the number of perfect matchings. However, there are two loopholes in this interpretation. First, as noted above, we do not restrict the input X to specify the set of perfect matchings of some graph. Unfortunately, with this restriction an interesting lower bound cannot be proved for the (nonuniform) decision tree model. To see this, let $w_i \in \{0, 1, *\}^m$ have 1 in the i th position and $*$'s elsewhere. If $\text{INT}(X_G, Q_{w_i}) = 1$ then e_i is an edge of the graph G . If $\text{INT}(X_G, Q_{w_i}) = 0$ then we can assume that e_i is not an edge of the graph since it is not used in any perfect matching. Thus, by making m samples, the decision tree can determine the graph. Since the graph determines the number of its perfect matchings, a nonuniform decision tree, using the samples w_i , can find the number of perfect matchings within time polynomial in the size of the graph. The second loophole is that it is conceivable that a different class of samples could lead to an efficient estimation procedure. There is room for more work on this problem.

Remark. It is natural to ask whether the structure of the subtree and partial match samples is being used to prove these lower bounds. Do lower bounds of the form N^ϵ hold for arbitrary classes of samples? It is not difficult to see that a much smaller sample height suffices if any subset can be a sample. Letting

$$\mathcal{P}_{\text{all}} = (U, \mathcal{X}_{\text{all}}) \quad \text{and} \quad \mathcal{X}_{\text{all}} = \mathcal{C}_{\text{all}} = 2^U,$$

it can be shown that

$$H(N, \mathcal{P}_{\text{all}}, \mathcal{C}_{\text{all}}, r, \delta) = O(\log \log N \log \log \log N),$$

$$H(N, \mathcal{P}_{\text{all}}, \mathcal{C}_{\text{all}}, r, \delta) = \Omega(\log \log N).$$

The simple proofs are sketched in [16]. Of course, this upper bound is useless in cases where N is exponentially large since time N is needed just to *name* a member of \mathcal{C}_{all} . It is also noted in [16] that without approximation (i.e. $r = 1$) or without randomization (i.e. $\delta = 0$), the obvious upper bound of N samples cannot be significantly improved, even when any subset of U can be a sample:

$$H(N, \mathcal{P}_{\text{all}}, \mathcal{C}_{\text{all}}, 1, \delta) = \Omega(N),$$

$$H(N, \mathcal{P}_{\text{all}}, \mathcal{C}_{\text{all}}, r, 0) = \Omega(N).$$

2.3. Proofs. For the lower bound proofs it is useful to assume that any probabilistic decision tree has the property that there is a constant p such that for any input X and any leaf l , the probability that the leaf l is reached is either p or 0. Any decision tree T not having this property can be modified to have the property without changing its sample height. First, by adding new randomizing nodes with outdegree 1, modify T so that the same number, say c , of randomizing nodes appear on every root-to-leaf path. Second, modify T so that all randomizing nodes have the same outdegree d

(where d can be taken as the least common multiple of the outdegrees of all randomizing nodes in the original T). Then $p = d^{-c}$. Details are left to the reader.

The proofs of the lower bounds for subtree and partial match samples are similar. In both cases we assume that the problem can be solved by a probabilistic decision tree T with sample height smaller than the desired lower bound. We find inputs D and $D \cup V$, with $r|D| < |D \cup V|/r$, such that a substantial fraction of the leaves with label between $|D|/r$ and $r|D|$ which are reached by T on input D are also reached by T on input $D \cup V$. From this we infer a contradiction, since an answer $\leq r|D|$ is incorrect for input $D \cup V$. There are two main steps.

I. Find a set $D \subseteq U$ with $|D| \approx N^{1/2}$ such that if $S \in \mathcal{C}$ and S is "large" (roughly $|S| \geq N^{1/2}$) then $D \cap S \neq \emptyset$. (Informally, if we restrict attention to inputs X with $D \subseteq X$, then all "large" samples supply no information about $X - D$ since the answer to any such sample is always that $X \cap S \neq \emptyset$.)

II. Find a collection of sets $V_1, \dots, V_t \subseteq U$ such that

(A) $|V_j| \geq (r^2 + 1)|D|$ for all j , and

(B) if $S_1, S_2, \dots, S_g \in \mathcal{C}$, each S_k is "small" (i.e., $S_k \cap D = \emptyset$ for $1 \leq k \leq g$) and $g \leq h$ where h is the lower bound on sample height to be proved, then the union of S_k for $1 \leq k \leq g$ intersects at most the fraction $(\frac{1}{2} - \delta)$ of the V_j 's.

Given I and II, the lower bound h on sample height is proved as follows. Let T be a decision tree with sample height h that approximates $|X|$ to within the factor r with error probability δ . If τ is a root-to-leaf path and $J \subseteq U$, say that τ is *valid* for J if the answers given at sample nodes along τ are consistent with the input being J . Let τ_1, \dots, τ_s be the root-to-leaf paths in T that are valid for D and such that the leaf label is between $|D|/r$ and $r|D|$. By letting the input be D , we must have $sp \geq 1 - \delta$, where p is the probability defined in the first paragraph of § 2.3. Consider the $s \times t$ 0-1 array $C = \{c_{ij}\}$ defined as follows. For $1 \leq i \leq s$ and $1 \leq j \leq t$, if S_1, S_2, \dots, S_g are the sample nodes on path τ_i that do not intersect D and if the union of the S_k for $1 \leq k \leq g$ intersects V_j then $c_{ij} = 1$; otherwise, $c_{ij} = 0$. By II(B), the density of 1's along any row of C is at most $(\frac{1}{2} - \delta)$. Therefore, there must be a column z such that the density of 1's in column z is at most $(\frac{1}{2} - \delta)$, so the density of 0's is at least $(\frac{1}{2} + \delta)$. But $c_{iz} = 0$ means that the path τ_i is valid for $D \cup V_z$. This is true because, by (I), any large sample on the path τ_i gives the same answer, " $\neq \emptyset$," for any input containing D , and by definition of c_{iz} no small sample on τ_i intersects V_z . By II(A), the answer at the leaf of any path τ_i is $\leq r|D| < |D \cup V_z|/r$. Therefore, when given the input $D \cup V_z$, the tree gives a too small answer with probability at least

$$(\frac{1}{2} + \delta)sp \geq (\frac{1}{2} + \delta)(1 - \delta) \geq \frac{1}{2} > \delta.$$

THEOREM 2.1. $H(N, \mathcal{P}_{\text{tree}}, \mathcal{C}_{\text{subtree}}, r, \delta) = \Omega(N^{1/2})$.

Proof. Recall from § 2.1 that F is the full binary tree of height n . The *depth* of a node of F is its distance to the root. $N = 2^{n+1} - 1$ is the number of nodes of F . Let

$$d = \lfloor (n - 1 - \log(r^2 + 1))/2 \rfloor,$$

and let D be the set of (labels of) nodes of F with depth $\leq d$. Let u_1, \dots, u_t be the nodes of depth d . For $1 \leq j \leq t$, let V_j be the set of (labels of) nodes in the full subtree of F rooted at u_j . A simple calculation shows that II(A) holds by choice of d , and that $t = \Omega(N^{1/2})$. From the tree structure it can be seen that if $S \in \mathcal{C}_{\text{subtree}}$ and S is "small," i.e., if $S \cap D = \emptyset$, then there is exactly one j such that $S \cap V_j \neq \emptyset$. (Specifically, if w is the root of the subtree S , then $S \cap D = \emptyset$ implies that the depth of w is greater than d , so exactly one of u_1, \dots, u_t is an ancestor of w .) Thus, a union of h subtree samples can intersect at most the fraction h/t of the V_j 's. Taking $h = (\frac{1}{2} - \delta)t$ gives the result. \square

THEOREM 2.2. $H(N, \mathcal{P}_{\text{word}}, \mathcal{C}_{\text{pm}}, r, \delta) = \Omega(N^{1/5})$.

Proof. Recall that $N = 2^m$. Say for simplicity that m is even. Using only the fact that \mathcal{C}_{pm} contains 3^m subsets, a simple probabilistic argument shows that there is a set D with $|D| = O(mN^{1/2})$ such that if $Q \in \mathcal{C}_{\text{pm}}$ and $|Q| \geq N^{1/2}$ then $Q \cap D \neq \emptyset$. Specifically, for each $u \in U$, put u in D with probability $q = \alpha m N^{-1/2}$ where α is a constant to be chosen later. By Chebyshev's inequality,

$$\Pr \{|D| \geq 2\alpha m N^{1/2}\} = O(N^{-1/2}).$$

If $Q \subseteq U$ and $|Q| \geq N^{1/2}$,

$$\Pr \{D \cap Q = \emptyset\} \leq (1 - q)^{|Q|} \leq e^{-\alpha m}.$$

Choose the constant α so that $3^m e^{-\alpha m} \leq \frac{1}{2}$. Then for all sufficiently large N , with nonzero probability $|D| = O(mN^{1/2})$ and D intersects all $Q \in \mathcal{C}_{\text{pm}}$ having $|Q| \geq N^{1/2}$.

If $w \in \{0, 1, *\}^m$ is the name of a partial match sample Q_w , let $*w$ be the number of $*$'s in w . Note that $|Q_w| = 2^{*w}$. Choose integer d such that

$$2^{m/2+d} \geq (r^2 + 1)|D|.$$

Note that $d = O(\log m)$. Let

$$\{V_1, \dots, V_t\} = \{Q_w \mid w \in \{0, 1, *\}^m \text{ and } *w = m/2 + d\}.$$

II(A) holds by choice of d . If Q_u is "small," i.e., if $*u \leq m/2$, we must find an upper bound for $f(u)$ defined to be the fraction of the V_j 's that Q_u intersects. Note that $Q_u \cap Q_w \neq \emptyset$ iff u matches w . Since $f(u)$ is an upper bound, we can assume that $*u = m/2$, and by symmetry assume that $u = u_1 u_2$ where u_1 is a string of $m/2$ 0's and 1's and u_2 is a string of $m/2$ $*$'s. Consider the w 's with $*w = m/2 + d$ that have k $*$'s in the first $m/2$ positions and therefore $m/2 + d - k$ $*$'s in the last $m/2$ positions. Letting $N(u, k)$ be the number of such w 's that match u ,

$$N(u, k) = \binom{m/2}{k} \binom{m/2}{m/2+d-k} 2^{k-d},$$

$$t = \binom{m}{m/2+d} 2^{m/2-d},$$

$$f(u) = \left(\sum_{k=d}^{m/2} N(u, k) \right) / t.$$

Since $d = O(\log m)$, there is a constant c such that

$$\binom{m}{m/2+d} \geq 2^m / m^c,$$

so $t \geq 2^{3m/2-d} / m^c$. Therefore,

$$f(u) \leq \left[\sum_{k=d}^{m/2} \binom{m/2}{k} 2^k \right] \left[\sum_{k=d}^{m/2} \binom{m/2}{k-d} \right] 2^{-3m/2} m^c \leq 3^{m/2} 2^{m/2} 2^{-3m/2} m^c.$$

Therefore, a union of $N^{1/5} (= 2^{m/5})$ partial match samples intersects at most the fraction

$$2^{m/5} \cdot f(u) \leq 2^{m/5} 3^{m/2} 2^{-m} m^c = o(1)$$

of the V_j 's, and the lower bound follows as outlined above. \square

THEOREM 2.3. $H(N, \mathcal{P}_{\text{tree}}, \mathcal{C}_{\text{subtree}}, r, \delta) = O((N \log N)^{1/2})$.

Proof. Number the nodes of F in inorder (to number a tree, number the left subtree, then the root, then the right subtree). For $u \in \{0, 1\}^{\leq n}$ let $d(u)$ be the inorder label of the node u . If X is a subtree of F , define

$$d(X) = \{d(u) \mid u \in X\}.$$

For $1 \leq a \leq b \leq N$, let

$$[a, b] = \{k \mid a \leq k \leq b\},$$

and define $\text{INT}(X, [a, b])$ to be 1 iff $d(X) \cap [a, b] \neq \emptyset$. Because the integer labeling is inorder, for each interval $[a, b]$ and any two nodes u and v with $d(u), d(v) \in [a, b]$, if w is the least common ancestor of u and v , then $d(w) \in [a, b]$. Therefore, for each $[a, b]$ there is a unique $w (=w(a, b))$ such that $d(w) \in [a, b]$ and, for all nodes $u, d(u) \in [a, b]$ implies that w is an ancestor of u . Since X satisfies the tree property

$$\text{INT}(X, [a, b]) = \text{INT}(X, S_{w(a, b)}).$$

Therefore, it suffices to estimate $|X|$ using “interval samples” of the form $[a, b]$. For some parameter M , the algorithm has two cases, $|X| \leq M$ and $|X| \geq M$. The first case has M phases and does not use randomization. In the first phase, by asking questions of the form $\text{INT}(X, [1, k])$ and doing binary search on k , find the smallest k , say k_1 , such that $k \in d(X)$; this takes $O(\log N)$ samples. During the second phase, by asking questions of the form $\text{INT}(X, [k_1 + 1, k])$ and again doing binary search on k , find the second smallest k with $k \in d(X)$, and so on. The first case uses $O(M \log N)$ samples. If $|X| < M$, then $|X|$ will be found exactly at phase number $|X| + 1$. If each phase in the first case finds a new element of $d(X)$, then we know that $|X| \geq M$, and the second case is invoked. In this case, since $|X| \geq M$, Chebyshev’s inequality implies that there is a constant α (depending only on r and δ) such that the algorithm approximates $|X|$ to within the factor r with error probability δ by making $\alpha N/M$ independent “singleton” samples of the form $[k, k]$, calculating the fraction γ of samples with $\text{INT}(X, [k, k]) = 1$, and answering $\lfloor \gamma N \rfloor$. The total number of samples for both cases is $O(M \log N + \alpha N/M)$ so choosing $M = (N/\log N)^{1/2}$ gives the result. \square

3. A general upper bound. An NP-machine is a nondeterministic polynomial-time Turing machine [3], [6]. Assume that NP-machines have at most two nondeterministic choices at each step so that accepting computations on inputs of length n can be represented as binary strings of length $p(n)$ where $p(n)$ is the machine’s polynomial time bound. If M is an NP-machine and $x \in \{0, 1\}^*$ is an input, let $\text{Acc}_M(x) \subseteq \{0, 1\}^{p(|x|)}$ be the set of accepting computations of M on input x . Define the function $C_M : \{0, 1\}^* \rightarrow \mathbf{N}$ by

$$C_M(x) = |\text{Acc}_M(x)|.$$

Valiant’s [17] class #P is

$$\#P = \{C_M \mid M \text{ is an NP-machine}\}.$$

For $A \subseteq \{0, 1\}^*$, $\#P^A$ is defined similarly except that M is a nondeterministic polynomial-time oracle machine [2], [15] that can construct binary strings and make decisions based on whether or not they belong to A . We also need a few classes of the relativized P-hierarchy. Let $P^A(\text{NP}^A)$ be the class of languages accepted by deterministic (nondeterministic) polynomial-time oracle machines with oracle A . If \mathcal{L} is a class of languages, let $P(\mathcal{L})(\text{NP}(\mathcal{L}))$ be the union of $P^A(\text{NP}^A)$ over all $A \in \mathcal{L}$. Let $\text{co-}\mathcal{L}$

be the class of complements of languages in \mathcal{L} . Define

$$\Sigma_2^{p,A} = \text{NP}(\text{NP}^A), \Pi_2^{p,A} = \text{co-}\Sigma_2^{p,A},$$

$$\Delta_2^{p,A} = \text{P}(\text{NP}^A), \Delta_3^{p,A} = \text{P}(\Sigma_2^{p,A}).$$

When A is not present, the empty oracle is assumed. We extend the classes $\Delta_2^{p,A}$ and $\Delta_3^{p,A}$ to include functions from $\{0, 1\}^*$ to \mathbb{N} ; since the “toplevel” machine is deterministic, the definition of this extension should be clear.

DEFINITION. If $f, g: \{0, 1\}^* \rightarrow \mathbb{N}$ and $r: \mathbb{N} \rightarrow \mathbb{R}$, we say that g r -approximates f if, for all n and all x of length n ,

$$\frac{f(x)}{r(n)} \leq g(x) \leq r(n) \cdot f(x).$$

THEOREM 3.1. Let $f \in \#P$ and let $\epsilon, d > 0$. There is a $g \in \Delta_3^p$ such that g $(1 + \epsilon n^{-d})$ -approximates f . Moreover, this relativizes to an arbitrary oracle A .

Proof. Let $f = C_M$ for some NP-machine M and let $p(n)$ be M 's polynomial time bound. Fix an input length n , and let $t = p(n)$. Consider the following predicate.

Hash(x, m):

There exist $m \times t$ 0-1 matrices H_1, \dots, H_m such that

- (2) for each $z \in \text{Acc}_M(x)$, there exists an i such that $H_i z \neq H_i z'$ for all $z' \in \text{Acc}_M(x) - \{z\}$.

Here, $H_i z$ means multiplication of the $m \times t$ matrix H_i by the t -vector z , yielding some m -vector in $\{0, 1\}^m$, where arithmetic is done modulo 2. The key fact, proved by Sipser [13], is that there is a fixed constant c such that

- (3) $|\text{Acc}_M(x)| \leq 2^{m-c} \Rightarrow \text{Hash}(x, m)$.

(Actually, this holds for any subset of $\{0, 1\}^t$, not just those of the form $\text{Acc}_M(x)$.) On the other hand, if $\text{Hash}(x, m)$ is true then, by (2), for each $z \in \text{Acc}_M(x)$ there is a unique pair $(i, H_i z)$ where $1 \leq i \leq m$ and $H_i z \in \{0, 1\}^m$, so

- (4) $\text{Hash}(x, m) \Rightarrow |\text{Acc}_M(x)| \leq m 2^m$.

Gács has observed that the predicate $\text{Hash}(x, m)$ belongs to Σ_2^p (see [13]). To see this, first note that the existential quantifier, “there exists an i ” in (2) has range $m = O(p(n))$, so it can be replaced by a deterministic search. Once this is done, the definition of $\text{Hash}(x, m)$ has an obvious $\exists \forall$ form. Therefore, a deterministic polynomial-time machine, making calls on an oracle for Hash, can find the minimum m such that $\text{Hash}(x, m)$ is true. If m is this minimum then, by (3) and (4),

$$2^{m-c-1} \leq |\text{Acc}_M(x)| \leq m 2^m,$$

so we have computed $C_M(x)$ to within the factor $m 2^{c+1}$. To achieve the smaller factor $1 + \epsilon n^{-d}$, by running M k times in series it is easy to modify M to an NP-machine R with

$$C_R(x) = (C_M(x))^k \quad \text{for all } x.$$

Applying the above hashing procedure to R , we can compute $C_R(x)$ to within the factor $m 2^{c+1}$, so we can compute $C_M(x)$ to within the factor $(m 2^{c+1})^{1/k}$, where now $m = O(k \cdot p(n))$. By choosing k to be a sufficiently large polynomial ($k = O(n^{d+1})$ works), this latter factor is less than $1 + \epsilon n^{-d}$. The relativized case is identical. \square

The next result shows that any attempt to replace Δ_3^P by Δ_2^P in Theorem 3.1 cannot proceed by a proof that relativizes to an arbitrary oracle.

THEOREM 3.2. *There are a recursive oracle A and a function $f \in \#P^A$ such that, for any constant r , if g r -approximates f then $g \notin \Delta_2^{P,A}$.*

Proof. Let $h(n) = n^{\log n}$. For $A \subseteq \{0, 1\}^*$, let

$$C_A(n) = |A \cap \{0, 1\}^n|.$$

The oracle A will be constructed such that, for all n ,

$$(5) \quad \text{either } C_A(n) \leq h(n) \text{ or } C_A(n) \geq 2^n - h(n).$$

Let $f(x) = C_A(|x|)$. Clearly $f \in \#P^A$. Let

$$L = \{x \mid C_A(|x|) \leq h(|x|)\}.$$

If $g \in \Delta_2^{P,A}$ and g r -approximates f for some constant r , then by using property (5) of A it is easy to see that $L \in \Delta_2^{P,A}$. We construct A so that $L \notin \Delta_2^{P,A}$.

Let M_1, M_2, \dots be an enumeration of all pairs (D_j, N_k) for $j, k \geq 1$, where D_1, D_2, \dots is an enumeration of the deterministic polynomial-time oracle Turing machines and N_1, N_2, \dots is an enumeration of the nondeterministic polynomial-time oracle Turing machines. Let p_j (resp., q_k) be the polynomial running time of D_j (resp., N_k). If $L \in \Delta_2^{P,A}$ then there is some $M_i = (D_j, N_k)$ such that D_j accepts L when D_j calls N_k as an oracle and N_k calls A as an oracle.

The construction of A proceeds by stages. During the construction we have two disjoint finite sets of strings B and \bar{B} that grow dynamically. At any point, B (resp., \bar{B}) is the set of strings that have been committed to be in A (resp., not in A). Strings are never removed from B or \bar{B} . Initially, $B = \bar{B} = \emptyset$. During the i th stage we extend the definitions of B and \bar{B} so that M_i does not accept L . Let n_1 be so large that

$$h(n_1) < 2^{n_1} - h(n_1).$$

Place all strings of length $\leq n_1$ in \bar{B} . The construction has the property that just before the start of stage i , a string is committed (in either B or \bar{B}) iff its length is $\leq n_i$.

Stage i . Let $M_i = (D_j, N_k)$. Choose $n > n_i$ so large that

$$p_j(n) \cdot q_k(p_j(n)) < h(n).$$

Set $n_{i+1} = q_k(p_j(n))$. If $n_i < |z| \leq n_{i+1}$ and $|z| \neq n$, place z in \bar{B} . Let $x = 0^n$. Start simulating D_j on input x until D_j makes its first oracle call, asking whether the string y is accepted by N_k with oracle A . Since $|y| \leq p_j(n)$, any oracle query “ $z \in A$?” made by N_k has $|z| \leq n_{i+1}$. Thus, at this point, the answer to any such query with $|z| \neq n$ is determined by the commitments to B and \bar{B} made so far. There are two possibilities.

1. For all ways of extending B and \bar{B} by adding uncommitted strings of length n , N_k does not accept y . In this case, the answer to D_j 's query is that N_k does not accept y . Continue simulating D_j until its next oracle call.
2. If the first case does not hold, then N_k has an accepting computation α on input y . Let S be the set of uncommitted strings of length n that are queried along α . Note that $|\alpha| \leq n_{i+1}$, so $|S| \leq n_{i+1}$. For each $z \in S$, commit z to be in either B or \bar{B} depending on whether the answer to the query “ $z \in A$?” in α is “yes” or “no,” respectively. After these commitments, N_k will accept y no matter how B and \bar{B} are extended further, since a nondeterministic machine accepts if it has any accepting computation, and further additions to B and \bar{B} cannot invalidate the accepting computation α . Continue simulating D_j until its next oracle call.

Each subsequent oracle call of D_j is handled similarly by either case 1 or 2. At the end of D_j 's computation, at most $p_j(n) \cdot n_{i+1} < h(n)$ strings of length n have been committed. If D_j accepts (resp., rejects) x , place all uncommitted strings of length n in B (resp., \bar{B}). In either case, D_j makes an error. \square

4. BPP and the P-hierarchy. A *probabilistic oracle Turing machine* has both coin-tossing states as in the definition of probabilistic Turing machines [4] and oracle query states as in the definition of oracle Turing machines [2], [6]. Following Gill [4] define BPP^A to be the class of languages accepted by polynomial-time probabilistic oracle Turing machines which, for all inputs, have error probability $\leq \frac{1}{2} - \epsilon$ for some fixed $\epsilon > 0$ when the oracle A is used.

Sipser and Gács [13] show that, for any oracle A ,

$$BPP^A \subseteq \Sigma_2^{P,A} \cap \Pi_2^{P,A}.$$

THEOREM 4.1. *There is a recursive oracle A such that*

$$BPP^A \not\subseteq \Delta_2^{P,A}.$$

Proof. Let $h(n)$, A and L be as in the proof of Theorem 3.2. Since $L \notin \Delta_2^{P,A}$ we only have to observe that $L \in BPP^A$. Given an input of length n , the probabilistic oracle Turing machine generates a random string z of length n and asks whether $z \in A$. If $z \in A$ then the machine rejects, or if $z \notin A$ then the machine accepts. The error probability is $\leq n^{\log n} / 2^n$ which is less than $\frac{1}{4}$ for all sufficiently large n . \square

5. Conclusion. It has been shown that any function in $\#P$ can be approximated by a function in Δ_3^P . Concerning lower bounds, if the $\#P$ problem is based on an NP-complete problem, for example, counting the number of satisfying truth assignments of a given propositional formula, it is easy to see that computing r -approximate solutions is NP-hard for any constant r . However, for approximately computing the permanent of a 0-1 matrix, where the corresponding existential question is solvable in polynomial time, the issue of lower bounds is open.

Question. Classify the computational complexities of approximately computing the permanent of a 0-1 matrix and approximately counting the number of satisfying assignments of a propositional formula. In particular, is the former problem NP-hard? Is the latter problem \mathcal{L} -hard for some class \mathcal{L} of the P-hierarchy above NP?

Recently, Karp and Luby [7] have discovered polynomial-time probabilistic approximation algorithms for certain problems in $\#P$ such as counting the number of satisfying assignments of a propositional formula given in disjunctive normal form.

Acknowledgments. I am grateful to Richard Lipton for suggesting the question of whether the size of a tree could be estimated using subtree samples and for many helpful discussions. One of the referees provided an extensive list of comments which helped improve the presentation of this material.

REFERENCES

- [1] D. ANGLUIN, *On counting problems and the polynomial time hierarchy*, Theoret. Comput. Sci., 12 (1980), pp. 161-173.
- [2] T. BAKER, J. GILL AND R. SOLOVAY, *Relativizations of the P = ?NP question*, this Journal, 4 (1975), pp. 431-442.
- [3] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979.
- [4] J. GILL, *Computational complexity of probabilistic Turing machines*, this Journal, 6 (1977), pp. 675-695.

- [5] L. M. GOLDSCHLAGER, *An approximation algorithm for computing the permanent*, Lecture Notes in Mathematics, 829, Springer-Verlag, Berlin, pp. 141-147.
- [6] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.
- [7] R. KARP AND M. LUBY, *Monte-Carlo algorithms for enumeration and reliability problems*, Proc. 24th IEEE Symposium on Foundations of Computer Science, 1983, pp. 56-64.
- [8] D. E. KNUTH, *Estimating the efficiency of backtrack programs*, Math. of Comput., 29 (1975), pp. 121-136.
- [9] E. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.
- [10] M. LUBY, personal communication.
- [11] U. MANBER AND M. TOMPA, *Probabilistic, nondeterministic, and alternating decision trees*, Proc. 14th ACM Symposium on Theory of Computing, 1982, pp. 234-244.
- [12] M. O. RABIN, *Probabilistic algorithms in finite fields*, this Journal, 9 (1980), pp. 273-280.
- [13] M. SIPSER, *A complexity theoretic approach to randomness*, Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 330-335.
- [14] R. SOLOVAY AND V. STRASSEN, *A fast Monte-Carlo test for primality*, this Journal, 6 (1977), pp. 84-85.
- [15] L. J. STOCKMEYER, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1-22.
- [16] ———, *The complexity of approximate counting*, Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 118-126.
- [17] L. G. VALIANT, *The complexity of computing the permanent*, Theoret. Comput. Sci., 8 (1979), pp. 189-201.
- [18] ———, *The complexity of enumeration and reliability problems*, this Journal, 8 (1979), pp. 410-421.

AN EFFICIENT PARALLEL BICONNECTIVITY ALGORITHM*

ROBERT E. TARJAN† AND UZI VISHKIN‡

Abstract. In this paper we propose a new algorithm for finding the blocks (biconnected components) of an undirected graph. A serial implementation runs in $O(n+m)$ time and space on a graph of n vertices and m edges. A parallel implementation runs in $O(\log n)$ time and $O(n+m)$ space using $O(n+m)$ processors on a concurrent-read, concurrent-write parallel RAM. An alternative implementation runs in $O(n^2/p)$ time and $O(n^2)$ space using any number $p \leq n^2/\log^2 n$ of processors, on a concurrent-read, exclusive-write parallel RAM. The last algorithm has optimal speedup, assuming an adjacency matrix representation of the input.

A general algorithmic technique that simplifies and improves computation of various functions on trees is introduced. This technique typically requires $O(\log n)$ time using processors and $O(n)$ space on an exclusive-read exclusive-write parallel RAM.

Key words. parallel graph algorithm, biconnected components, blocks, spanning tree

1. Introduction. In this paper we consider the problem of computing the blocks (biconnected components) of a given undirected graph $G=(V, E)$. As a model of parallel computation, we use a concurrent-read, concurrent-write parallel RAM (CRCW PRAM). All the processors have access to a common memory and run synchronously. Simultaneous reading by several processors from the same memory location is allowed as well as simultaneous writing. In the latter case one processor succeeds but we do not know in advance which. This model, used for instance in [SV82], is a member of a family of models for parallel computation. (See [BH82], [SV81], [V83c].)

We propose a new algorithm for finding blocks. We discuss three implementations of the algorithm:

1. A linear-time sequential implementation.
2. A parallel implementation using $O(\log n)$ time, $O(n+m)$ space, and $O(n+m)$ processors, where $n=|V|$ and $m=|E|$.
3. An alternative parallel implementation using $O(n^2/p)$ time, $O(n^2)$ space, and any number $p \leq n^2/\log^2 n$ of processors. This implementation uses a concurrent-read, exclusive-write parallel RAM (CREW PRAM). This model differs from the CRCW PRAM in not allowing simultaneous writing by more than one processor into the same memory location. The speed-up of this implementation is optimal in the sense that the time-processor product is $O(n^2)$, which is the time required by an optimal sequential algorithm if the input representation is an adjacency matrix.

Implementation 2 is faster than any of the previously known parallel algorithms [SJ81], [Ec79b], [TC84]. Eckstein's algorithm [Ec79b] uses $O(d \log^2 n)$ time and $O((n+m)/d)$ processors, where d is the diameter of the graph. The first (resp. second) algorithm of Savage and Ja'Ja' [SJ81] uses $O(\log^2 n)$ (resp. $O((\log^2 n) \log k)$) time, where k is the number of blocks, and $O(n^3/\log n)$ (resp. $O(mn + n^2 \log n)$) processors.

* Received by the editors August 11, 1983, and in final revised form August 22, 1984. This is a revised and expanded version of the paper *Finding biconnected components and computing tree functions in logarithmic parallel time*, appearing in the 25th Annual Symposium on Foundations of Computer Science, Singer Island, FL, October 24-26, 1984, © 1984 IEEE.

† AT & T Bell Laboratories, Murray Hill, New Jersey 07974.

‡ Courant Institute, New York University, New York, New York 10012 and (present address) Department of Computer Science, School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel. The research of this author was supported by the U.S. Department of Energy under grant DE-AC02-76ER03077, by the National Science Foundation under grants NSF-MCS79-21258 and NSF-DCR-8318874, and by the U.S. Office of Naval Research under grant N0014-85-K-0046.

Tsin and Chin's algorithm [TC84] matches the bounds of our implementation 3. These algorithms use the CREW PRAM model, which is somewhat weaker than the CRCW PRAM model. However, Eckstein [Ec79a] and Vishkin [V83a] present general simulation methods that enable us to run implementation 2 on a CREW PRAM in $O(\log^2 n)$ time, without increasing the number of processors. On sparse graphs, the resulting algorithm uses fewer processors than either our implementation 3 or the algorithm of Tsin and Chin.

We achieve our improvements through two new ideas:

1. A block-finding algorithm that uses any spanning tree. The previously known linear-time algorithm for finding blocks uses a depth-first spanning tree [Ta72]. Depth-first search seems to be inherently serial; i.e. there is no apparent way to implement it in poly-log parallel time. The algorithm uses a reduction from the problem of computing biconnected components of the input graph to the problem of computing connected components of an auxiliary graph. This reduction can be computed efficiently enough both sequentially and in parallel that the running time of the fastest parallel connectivity algorithm becomes the only obstacle to a further improvement in implementation 2. (See the discussion in § 5.)

2. A novel algorithmic technique for parallel computations on trees is introduced. Given a tree, the technique uses an Euler tour of a graph obtained from the tree by adding a parallel edge for each edge of the tree. Therefore, we call it the *Euler tour technique on trees*. This technique allows the computations of various kinds of information about the tree structure in $O(\log n)$ time using $O(n)$ processors and $O(n)$ space on an exclusive-read exclusive-write parallel RAM. This model differs from the CREW PRAM in not allowing simultaneous reading from the same memory location. In the present paper we show how to use this Euler tour technique in order to compute preorder and postorder numbering of the vertices of a tree, number of descendants for all vertices, and other tree functions. Recently Vishkin [V84] proposed further extensions of this technique. (See § 5.) After the appearance of the first version of the present paper Awerbuch et al. [AIS84] and independently Atallah and Vishkin [AV84] essentially applied Euler tours on trees to finding Euler tours of general Eulerian graphs. See [AV84] for an explanation of this connection. The previously best known general technique for parallel computations on trees is the *centroid decomposition* method, which gives $O(\log^2 n)$ -time algorithms. See [M83] for a discussion of this method and its use. The centroid decomposition method is the backbone of an earlier paper by Winograd [Wi75].

The idea of reducing the biconnectivity problem to a connectivity problem on an auxiliary graph was discovered independently by Tsin and Chin [TC84], who used the idea in their block-finding algorithm. However, their algorithm has two drawbacks:

- (1) Their auxiliary graph contains many more edges than ours. This complicates the computation of the auxiliary graph and, more important, does not lead to a fast parallel algorithm using only a linear number of processors. One of the elegant features of our algorithm is that the same reduction is used in all three implementations.

- (2) Their computation of preorder and postorder numbers and number of descendants in trees takes $O(\log n)$ time using $n^2/\log n$ processors—almost the square of the number of processors that we use.

The remainder of the paper consists of four sections. In § 2 we develop the block-finding algorithm and give a linear-time sequential implementation. In § 3 we describe our $O(\log n)$ -time parallel implementation and present the Euler tour technique. Section 4 sketches our alternative parallel implementation. In § 5 we discuss variants of the algorithm for solving two additional problems: finding bridges and

directing the edges of a biconnected graph to make it strongly connected. We also discuss possible future work.

Note. If a parallel algorithm runs in $O(t)$ time using $O(p)$ processors then it also runs in $O(t)$ time using p processors. This is because we can always save a constant factor in the number of processors at the cost of the same constant factor in running time.

Historical remark. A variant of the block-finding algorithm presented here was first discovered by R. Tarjan in 1974 [T82]. U. Vishkin independently rediscovered a similar algorithm in 1983 and proposed a parallel implementation and the Euler tour technique [V83b]. Subsequent simplification by the two authors working together resulted in the present paper.

2. Finding blocks. Let $G = (V, E)$ be a connected undirected graph. Let R be the relation on the edges of G defined by $e_1 R e_2$ if and only if $e_1 = e_2$ or e_1 and e_2 are on a common simple cycle¹ of G . It is known that R is an equivalence relation [Ha69]. The subgraphs of G induced by the equivalence classes of R are the *blocks* (sometimes called *biconnected components*) of G . The vertices in two or more blocks are the *cut vertices* (sometimes called *articulation points*) of G ; these are the vertices whose removal disconnects G . The edges in singleton equivalence classes are the *bridges* of G ; these are the edges whose removal disconnects G . (See Fig. 1.)

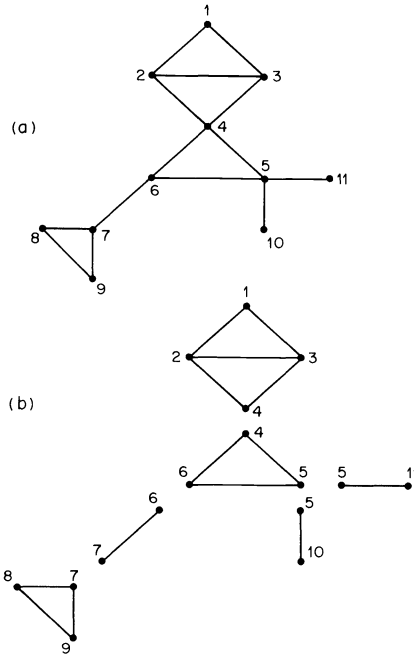


FIG. 1. (a) An undirected graph. (b) Its blocks. Vertices 4, 5, 6 and 7 are cut vertices. Edges $\{6, 7\}$, $\{5, 10\}$, and $\{5, 11\}$ are bridges.

We can compute the equivalence classes of R , and thus the blocks of G , in $O(n + m)$ serial time using depth-first search [Ta72], where $n = |V|$ and $m = |E|$. Unfortunately, this algorithm seems to have no fast parallel implementation. In this

¹ In this paper a *cycle* is a path starting and ending at the same vertex and repeating no edge; a cycle is *simple* if it repeats no vertex except the first, which occurs exactly twice.

section we develop an $O(n + m)$ -time serial algorithm that is well-suited for parallel implementation. The algorithm can use any spanning tree, not just a depth-first spanning tree.

We shall define an auxiliary graph G' of G whose connected components correspond to the blocks of G . The vertices of G' are the edges of G ; if S is a set of edges in G , S induces a block of G if and only if S induces a connected component of G' . Let T be any rooted spanning tree of G . We shall denote the edges of T by $v \rightarrow w$, where v is the parent of w , denoted by $p(w)$. Let the vertices of T be numbered from 1 to n in preorder and identify each vertex by its number. G' contains each edge of G as a vertex and all edges of the following forms (see Fig. 2):

- (i) $\{\{u, w\}, \{v, w\}\}$, where $u \rightarrow w$ is an edge of T and $\{v, w\}$ is an edge of $G - T$ such that $v < w$.
- (ii) $\{\{u, v\}, \{x, w\}\}$, where $u \rightarrow v$ and $x \rightarrow w$ are edges of T and $\{v, w\}$ is an edge of $G - T$ such that v and w are unrelated in T .
- (iii) $\{\{u, v\}, \{v, w\}\}$, where $u \rightarrow v$ and $v \rightarrow w$ are edges of T and some edge of G joins a descendant of w with a nondescendant of v .

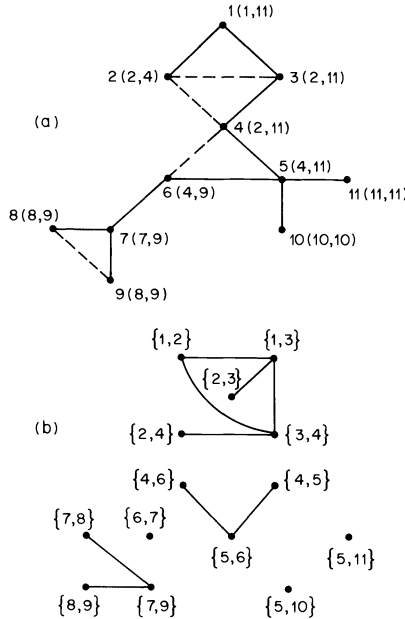


FIG. 2. (a) A spanning tree of the graph in Fig. 1. Dashed edges are nontree edges. Vertices are numbered in preorder. Numbers in parentheses are the low and high number of each vertex. (b) The auxiliary graph G' .

The intuition behind this construction is that every edge of $G - T$ defines a cycle consisting of this edge and the path in T joining its endpoints. All edges on this cycle are in the same biconnected component. We add enough edges to G' so that the vertices in G' corresponding to the edges on the cycle are in the same connected component.

THEOREM 1. *Two edges of G are in a common block of G if and only if as vertices of G' they are in a common connected component of G' .*

Proof. Any edge $\{x, y\}$ of $G - T$ defines a simple cycle of G , consisting of edge $\{x, y\}$ and the unique path in T joining x and y . These cycles are a cycle basis of G ; the edge set of any cycle is the mod-two sum of the edge sets of appropriate basis cycles [Be73]. Define the relation R' by $e_1 R' e_2$ if and only if e_1 and e_2 are two edges of G on a common basis cycle, and let R'^* be the reflexive, transitive closure of R' .

We claim $R'^* = R$. Since R is an equivalence relation and $R' \subseteq R$, we have $R'^* \subseteq R$. To prove the converse, suppose $e_1 R e_2$. Then e_1 and e_2 are on a common simple cycle, which is a mod-two sum of basis cycles C_1, C_2, \dots, C_k . Without loss of generality we can order C_1, C_2, \dots, C_k so that C_i for $i > 1$ has at least one edge in common with some C_j such that $j < i$. (Otherwise the mod-two sum of C_1, C_2, \dots, C_k would induce a disconnected subgraph.) It follows by induction on k that all edges in C_1, C_2, \dots, C_k are equivalent under R'^* , and in particular $e_1 R'^* e_2$. Thus $R \subseteq R'^*$.

Let $\{u, v\}$ and $\{x, w\}$ be adjacent in G' . If case (i) holds, $\{u, v\}$ is on the basis cycle defined by $\{x, w\}$. (In this case $x = v$.) If Case (ii) holds, $\{u, v\}$ and $\{x, w\}$ are on the basis cycle defined by $\{v, w\}$. If Case (iii) holds, say $\{y, z\}$ is an edge with y a descendant of w and z a nondescendant of $v = x$, then $\{u, v\}$ and $\{x, w\}$ are on the basis cycle defined by $\{y, z\}$. Thus in all cases $\{u, v\}$ and $\{x, w\}$ are in the same block of G .

Conversely, let $\{x, y\}$ be an edge of $G - T$ defining a basis cycle consisting of edge $\{x, y\}$, edges on the tree path from z to x , and edges on the tree path from z to y , where z is the nearest common ancestor of x and y . Without loss of generality suppose $x < y$. By Case (i), $\{x, y\}$ and $\{p(y), y\}$ are adjacent in G' . The existence of $\{x, y\}$ implies by Case (iii) that any two edges on the tree path from z to x are adjacent in G' . Similarly any two edges on the tree path from z to y are adjacent. If $z = x$, the tree path from z to x is empty. Otherwise (i.e. $z \neq x$), x and y are unrelated, and by Case (ii) $\{p(x), x\}$ and $\{p(y), y\}$ are adjacent in G' . Thus all edges on the basis cycle are in the same connected component of G' . The theorem follows.

Theorem 1 gives the following $O(n+m)$ -time serial algorithm for finding blocks:

Step 1. Find a spanning tree T of G using any linear-time search method. Number the vertices of G from 1 to n in preorder and identify each vertex by its preorder number. Compute the number of descendants $nd(v)$ of each vertex v by processing the vertices in postorder using the recurrence $nd(v) = 1 + \sum \{nd(w) \mid v \rightarrow w \text{ in } T\}$. (We regard every vertex as a descendant of itself.) A vertex w is a descendant of another vertex v if and only if $v \leq w \leq v + nd(v) - 1$ [Ta74a].

Step 2. For each vertex v , compute $low(v)$, the lowest vertex that is either a descendant of v or adjacent to a descendant of v by an edge of $G - T$, and $high(v)$, the highest vertex that is either a descendant of v or adjacent to a descendant of v by an edge of $G - T$. The complete set of $2n$ low and $high$ vertices can be computed in $O(n+m)$ time by processing the vertices of T in postorder using the following recurrences:

$$low(v) = \min (\{v\} \cup \{low(w) \mid v \rightarrow w \text{ in } T\} \cup \{w \mid \{v, w\} \text{ in } G - T\});$$

$$high(v) = \max (\{v\} \cup \{high(w) \mid v \rightarrow w \text{ in } T\} \cup \{w \mid \{v, w\} \text{ in } G - T\}.$$

Step 3. Construct G'' , the subgraph of G' induced by the edges of T , as follows. (The edges of G'' are those implied by cases (ii) and (iii).) For each edge $\{w, v\}$ in $G - T$ such that $v + nd(v) \leq w$, add $\{\{p(v), v\}, \{p(w), w\}\}$ to G'' (Case (ii)). For each edge $v \rightarrow w$ of T such that $v \neq 1$ add $\{\{p(v), v\}, \{v, w\}\}$ to G'' if $low(w) < v$ or $high(w) \geq v + nd(v)$ (Case (iii).)

Step 4. Find the connected components of G'' using any kind of linear-time search.

Step 5. Extend the equivalence relation on the edges of T (the vertices of G'') to the edges of $G - T$ by defining $\{v, w\}$ equivalent to $\{p(w), w\}$ for each edge $\{v, w\}$ of $G - T$ such that $v < w$ (Case (i).)

It is easy to implement this algorithm to run in $O(n+m)$ time using standard techniques. (See [Ta72]). If only a serial implementation is desired, the algorithm can be simplified somewhat (see [Ta82]); the algorithm as presented is designed for easy

parallel implementation. Note that each edge of $G - T$ is a vertex of degree one in G' , and G'' contains $n - 1$ vertices and at most $m - 1$ edges.

Remark. Although we have assumed that G is connected, we can use the algorithm to find the blocks of a disconnected graph by applying it to each of the connected components (in series in the case of the implementation in this section, in parallel in the case of the implementations in §§ 3 or 4). This does not change the resource bounds of the algorithm.

3. A fast parallel implementation. In this section we describe how to implement the block-finding algorithm of § 2 to run in $O(\log n)$ time using $O(n + m)$ processors on a CRCW PRAM. We shall emphasize the ideas involved, only sketching the details. As the input representation, we assume that the vertex set is $V = \{1, 2, \dots, n\}$ and that each undirected edge $\{i, j\}$ is represented by two directed edges (i, j) and (j, i) . Each vertex i has a list of its outgoing edges: $adj(i)$ points to the first such edge and $next((i, j))$ points to the edge after (i, j) on i 's list. (If there is no such edge, $next((i, j)) = \text{null}$.) Each edge (i, j) also has a pointer to its reversal (j, i) . Each vertex i and each directed edge (i, j) has its own processor, denoted by $pr(i)$ and $pr(i, j)$, respectively.

Remark. This input representation is the most convenient one for our purposes, but it is not the only one that will work. For example, we can begin with an array of the $2m$ directed edges in arbitrary order and use the $O(\log m)$ -time, $O(m)$ -processor sorting algorithm of Ajtai, Komlós, and Szemerédi [AKS83] to sort the edges by first component. Once the edges are sorted, it is easy to construct incidence lists. Sorting the edges (i, j) lexicographically on $(\min\{i, j\}, \max\{i, j\})$ allows the construction of pointers between each edge and its reversal. Thus we obtain the desired input representation. While the asymptotic running time of this sorting algorithm is only $O(\log m)$, the constant factor is huge. Instead of this algorithm, we can use the randomized sorting algorithm of Reif and Valiant [RV83]. It will sort in time $O(\log m)$ almost surely using m processors. A third possibility is to perform this sorting in time $O(\log n)$ and m processors using an adaptation of the simple notion of "orthogonal trees". However, this takes $O(n^2)$ space. For more information on such sorting algorithms see Thompson [Th83].

Step 1. Construction of a spanning tree and computation of the preorder number and number of descendants of each vertex.

First we construct an unrooted spanning tree by using a modification of the Shiloach-Vishkin connected components algorithm [SV82]. We assume some familiarity with this algorithm. The algorithm maintains for each vertex v a pointer $D(v)$. Initially $D(v) = v$ for all vertices v . As the algorithm proceeds, the D -pointers are the parent pointers of a forest, each tree of which contains vertices known to be in a single connected component of the graph. (If v is the root of a tree in this D -forest, $D(v) = v$.) The D -pointers are changed by two kinds of steps:

Shortcutting. Replace $D(i)$ by $D(D(i))$ for some vertex i . Such a step changes the structure of the D -forest by moving v and its descendants closer to the root of its tree, but does not change the vertex partition defined by the D -trees.

Hooking. Replace $D(D(i))$ by $D(j)$, where $D(i)$ is the root of a D -tree, j is a vertex in another D -tree, and $\{i, j\}$ is an edge in the graph.

We modify the Shiloach-Vishkin algorithm so that all the edges are initially marked as nontree edges, and each time a hooking step is performed, the corresponding graph edge $\{i, j\}$ is marked as a tree edge. When the algorithm finishes, all the vertices are in a single D -tree, and the marked edges define a spanning tree. The original algorithm

runs in $O(\log n)$ time using $O(n + m)$ processors; these bounds are not affected by the modifications for computing a spanning tree.

One detail of this method deserves further discussion. Processors corresponding to several directed edges (i, j) may simultaneously try to write to the same location $D(D(i))$ to cause a hooking, but only one succeeds. In order to keep track of which one succeeds, we use an auxiliary array α . When a processor $pr((i, j))$ tries to cause a hooking step to take place, it first writes its name into $\alpha(D(i))$ by the assignment $\alpha(D(i)) \leftarrow pr((i, j))$. For a fixed value of $D(i)$, only one such processor succeeds. The successful processor $pr((i, j))$ then carries out the actual hooking step and marks both (i, j) and (j, i) .

Remark. This idea for obtaining a spanning tree from a connected components computation has been used before. In particular Savage and Ja'Ja' [SJ81] used it to derive a minimum spanning forest algorithm from the connectivity algorithm of Hirschberg, Chandra and Sarwate [HCS79].

Having determined the edges of an unrooted spanning tree, we choose a root and number the vertices of the resulting rooted tree in preorder. To do this we first construct for each vertex i a list of the outgoing edges corresponding to tree edges. We can do this in $O(\log m) = O(\log n)$ time with $O(m)$ processors by using a standard "doubling" technique [Wy79]. For each (i, j) , we initialize $treenext((i, j)) = next((i, j))$ and then repeat the following step, in parallel on all edges (i, j) , $\lceil \log m \rceil$ times (until none of the $treenext$ values change): if $treenext((i, j))$ is not null and not marked, replace $treenext((i, j))$ by $treenext(treenext((i, j)))$. Once all the $treenext$ values are computed, we define $treadj(i)$, for each vertex i , to be $adj(i)$ if $adj(i)$ is null or marked, $treenext(adj(i))$ otherwise. The $treadj$ and $treenext$ maps define incidence lists for the spanning tree.

Next, we construct a circular list corresponding to an Eulerian tour of the directed version of the spanning tree. For each edge (i, j) , the next edge $tournext((i, j))$ in the tour is $treenext((j, i))$ if $treenext((j, i))$ is not null, $treadj(i, j)$ otherwise. This tour corresponds to the order of advancing and retreating along edges during a depth-first transversal of the tree, starting at an arbitrary vertex. To root the tree, we break the Eulerian tour at an arbitrary edge, causing some edge, say (i, j) , to be the first edge on the list. Vertex i becomes the root of the tree. We call the broken list the *traversal list*. This traversal list is the backbone of the Euler tour technique that is introduced in this paper. In the sequel, we show that this list is the key to computing a number of tree functions.

We can number the edges of the traversal list from 1 to $2n - 2$ in traversal order in $O(\log n)$ time with $O(n)$ processors by using the doubling technique to compute for each edge (i, j) the number of edges from (i, j) to the end of the list. We do this by initializing $numtoend((i, j)) = 1$ and $ptr((i, j)) = \text{null}$ for all $((i, j))$. Once this computation is complete, the number of edge (i, j) is $2n - 1 - numtoend((i, j))$.

Of two edges (i, j) and (j, i) , the lower-numbered one corresponds to an advance from i to j along tree edge $\{i, j\}$ and the higher-numbered one to a retreat from j to i along $\{i, j\}$. Using the edge numbers, we can thus mark each directed edge as either an advance edge or a retreat edge. For each vertex j other than the root, there is exactly one advance edge (i, j) ; the parent $p(j)$ of j in the tree is i .

In the traversal list, the advance edges (i, j) occur in preorder on j . We can thus number the vertices in preorder using doubling, much as we computed the edge numbers. The only differences are that we initialize $numtoend(i, j)$ to be 1 if (i, j) is an advance edge, 0 otherwise, and when the computation is complete, if (i, j) is an

advance edge, we define $n+1 - \text{numtoend}(i, j)$ to be the reorder number of vertex j . Once preorder numbers are computed, we replace each occurrence of a vertex by its preorder number, retaining an inverse map to restore the original vertex names when the computation is complete. (For each number i , we remember $\text{vertex}(i)$, the vertex with number i .)

Remark. Although not needed in this paper, a similar computation will number the vertices in postorder; for each vertex j other than the tree root, there is exactly one retreat edge (j, i) , and the retreat edges appear in the transversal list in postorder on j .

The last part of Step 1 is the computation of the number of descendants $nd(j)$ of each vertex j . If j is not the tree root, $nd(j)$ is just the number of advance edges from $(p(j), j)$ to the end of the list (including $(p(j), j)$) minus the number of advance edges from $(j, p(j))$ to the end of the list. Two doubling computations, one of which we have already done to compute preorder numbers, and a parallel subtraction give the number of descendants of all the vertices.

Step 2. Computation of $\text{low}(j)$ and $\text{high}(j)$ for each vertex j .

We shall describe how to compute low ; the computation of high is similar. Using doubling on the adjacency lists, we can compute $\text{locallow}(j) = \min(\{j\} \cup \{k | (j, k) \text{ is an unmarked (nontree) edge}\})$ for each vertex j in $O(\log n)$ time using $O(m)$ processors. Below we assume without loss of generality that n is a power of 2. We define an auxiliary value $\text{globalow}[i, j] = \min(\{\text{localow}(k) | i \leq k \leq j\})$, i.e., $\text{globalow}[i, j]$ is the minimum of localow over the interval $[i, i+1, \dots, j]$. For each $0 \leq \alpha \leq \log n$ we compute globalow of the intervals $[(k-1)2^\alpha + 1, \dots, k2^\alpha]$ for $1 \leq k \leq n/2^\alpha$. (The total number of such intervals is $O(n)$. They have the property that any interval $[i, \dots, j]$, $1 \leq i \leq j \leq n$, can be represented as a union of at most $2 \log n$ of them.)

Initialization. Assign $\text{globalow}[i, i] \leftarrow \text{localow}(i)$ for all $1 \leq i \leq n$.

for $\alpha \leftarrow 1$ to $\log n$ pardo

 for $0 \leq k \leq (n/2^\alpha) - 1$ do

$\text{globalow}[k2^\alpha + 1, (k+1)2^\alpha] \leftarrow$

$\min(\text{globalow}[k2^\alpha + 1, (2k-1)2^{\alpha-1}],$

$\text{globalow}[(2k-1)2^{\alpha-1} + 1, (k+1)2^\alpha])$

 end for

end for

This computation takes $O(\log n)$ time using n processors. (Actually, $n/\log n$ processors suffice but this is not important here.)

We compute $\text{low}(j)$ for each vertex j using the formula

$$\text{low}(j) = \min \{ \text{localow}(k) | j \leq k \leq j + nd(j) - 1 \}.$$

That is, we compute $\text{globalow}[j, j + nd(j) - 1]$, for each vertex j . The computation below uses the property that the interval $[j, \dots, j + nd(j) - 1]$ is a union of at most $2 \log n$ intervals on which globalow has already been computed. The variables $\text{little}(j)$ and $\text{big}(j)$ initially mark the endpoints of the interval. During the course of the computation the interval $[\text{little}(j), \dots, \text{big}(j)]$ contains the subinterval of $[j, \dots, j + nd(j) - 1]$ that has not yet been taken into account in the computation of $\text{low}(j)$.

```

for  $2 \leq j \leq n$  pardo
  Initialize:  $little(j) \leftarrow j$ ;  $big(j) \leftarrow j + nd(j) - 1$ ;
              $low(j) \leftarrow n + 1$  (Comment: This is a default value)
  for  $\alpha \leftarrow 1$  to  $\log n$  do
    if  $little(j) - 1$  is not divisible by  $2^\alpha$ 
    then  $low(j) \leftarrow \min(low(j), globalow[little(j), little(j) + 2^{\alpha-1} - 1])$ 
          $little(j) \leftarrow little(j) + 2^{\alpha-1}$ 
    end if
    if  $big(j)$  is not divisible by  $2^\alpha$ 
    then  $low(j) \leftarrow \min(low(j), globalow[big(j) - 2^{\alpha-1} + 1, big(j)])$ 
          $big(j) \leftarrow big(j) - 2^{\alpha-1}$ 
    end if
    if  $little(j) > big(j)$ 
    then Halt and output  $low(j)$ 
    end if
  end for
end for

```

It is easy to verify the following. (1) All our requests for values of *globalow* are for intervals that have been previously computed. (2) The intervals that are taken into account in the computation of *low*(*j*) actually cover the interval $[j, \dots, j + nd - 1]$. (3) The whole computation of Step 2 takes $O(\log n)$ time using $O(n)$ processors.

Step 3. Construction of the auxiliary graph G'' .

This computation requires only $O(1)$ time using $O(m)$ processors, since testing the appropriate condition for each possible edge of G'' takes $O(1)$ time. After this test, which takes place in parallel, we have a set of at most $m - 1$ processors, each of which knows an edge of G'' .

Step 4. Finding the connected components of G'' .

We apply the connected components algorithm of Shiloach and Vishkin. The information computed in Step 3 is sufficient as input to this algorithm, which takes $O(\log n)$ time and $O(n + m)$ processors. Once the algorithm finishes, each vertex (i, j) of G'' (advance edge of the spanning tree) has a *D*-pointer to a canonical "vertex" (x, y) representing the connected component containing (i, j) .

Step 5. Extension of the equivalence relation found in Step 4 to the edges of $G - T$.

For each nontree edge (i, j) such that $i < j$, we assign $D((i, j)) \leftarrow D((p(j), j))$. This takes $O(1)$ time and $O(m)$ processors.

This completes the computation except for restoring the original vertex names. An inspection of the various steps shows that none uses more than $O(\log m) = O(\log n)$ time, more than $O(n + m)$ space, or more than $O(n + m)$ processors. The only place concurrent writing is used is in the connected components algorithm, used in Steps 1 and 4.

4. An alternative parallel implementation. In this section we develop an implementation of the block-finding algorithm that runs in $O(\log^2 n)$ time using $O(n^2/\log^2 n)$ processors on a CREW PRAM, assuming that the input graph is represented by an adjacency matrix. Since we can always trade time for processors, this method gives an $O(n^2/p)$ time algorithm using p processors, for any $p \leq n^2/\log^2 n$. This algorithm has optimal speed-up, assuming an adjacency matrix representation of the input. We shall not go through the details of the implementation but merely mention where it differs from the $O(\log n)$ -time implementation of the previous section.

There are two known connected components algorithms that run in $O(\log^2 n)$ time using $O(n^2/\log^2 n)$ processors: the algorithm of Vishkin [V81], which runs on a

CRCW PRAM, and the algorithm of Chin, Lam, and Chen [CLC81], which runs on a CREW PRAM. Although the latter is more complicated, we shall use it instead of the former in Steps 1 and 4, since it uses a less powerful computation model. Chin, Lam, and Chen describe how to adapt their algorithm to compute a (minimum) spanning forest.

Step 1. Construction of a spanning tree and computation of the preorder number and number of descendants of each vertex.

We apply the algorithm of Chin, Lam, and Chen to mark the entries in the adjacency matrix corresponding to tree edges. We can convert each row of the adjacency matrix to an incidence list for the corresponding vertex (of edges incident in the spanning tree) by using a balanced binary tree with n leaves to guide the computation. (For each marked entry, we need to compute the next marked entry in the row.) The computation is similar to a standard partial-sum computation and takes $O(\log^2 n)$ time with $O(n/\log^2 n)$ processors (see for instance [V81]). Since we can carry out the computation for all rows in parallel, the total time is $O(\log^2 n)$ with $O(n^2/\log^2 n)$ processors. Establishing pointers between each directed edge (i, j) and its reverse is easy. Now we have the representation of the unrooted spanning tree used in § 3. The remainder of the Step 1 computation proceeds as in § 3, taking $O(\log n)$ time on $O(n)$ processors.

Step 2. Computation of *low* and *high*.

Computing *locallow*(j) requires n parallel minimum computations. Each takes $O(\log^2 n)$ time using $O(n/\log^2 n)$ processors [Wy79], a total of $O(n^2/\log^2 n)$ processors. The remainder of the *low* computation proceeds as in § 3 taking $O(\log n)$ time using $O(n)$ processors. The computation of *high* is similar.

Step 3. Construction of the auxiliary graph G'' .

This is easy in $O(\log^2 n)$ time with $O(n^2/\log^2 n)$ processors.

Step 4. Finding the connected components of G'' .

Step 5. Extension of the equivalence relation found in Step 4 to the edges of $G - T$. This is easy in $O(\log^2 n)$ time with $O(n^2/\log^2 n)$ processors.

5. Extensions and future work. There are two related problems that can be solved using variants of our algorithm, in the same resource bounds. Neither of these requires the second connected-component-finding step (Step 4). The first is the problem of finding all bridges of a graph. The bridges are just the one-edge biconnected components. Thus we can use the biconnected components algorithm directly. However, there is a simpler algorithm. Suppose we number the vertices in preorder with respect to any spanning tree; identify vertices by number; compute $nd(v)$, the number of descendants of the vertex, for each vertex v ; and compute the *low* and *high* functions defined in § 2. A tree edge $v \rightarrow w$ with v the parent of w is a bridge if and only if $w \leq low(w)$ and $high(w) \leq w + nd(w) - 1$, i.e. if and only if both *low*(w) and *high*(w) are descendants of w [Ta74b], [TC83]. No nontree edge is a bridge. By applying this test, we can find all bridges in $O(\log n)$ time and $O(n+m)$ processors on a CRCW PRAM using the algorithm of § 2, or in $O(n^2/p)$ time using any number $p \leq n^2/\log^2 n$ of processors on a CREW PRAM using the algorithm of § 3. The latter bounds for bridge-finding were first obtained by Tsin and Chin [TC83] using this approach; the former bounds are new.

The second problem is that of directing the edges of a bridgeless graph so that the resulting directed graph is strongly connected. Atallah [At84] proposed an algorithm for this problem that runs in $O(\log n)$ time using $O(n^3)$ processors on a CRCW PRAM. Vishkin [V84] gave an algorithm with the same resource bounds as our method for finding bridges and biconnected components. We shall propose an alternative, simpler

algorithm. As above, assume that we have found a spanning tree, numbered the vertices in preorder, identified each vertex by its number, and computed $low(v)$ for each vertex v . Let $\{v, w\}$ be a nontree edge. We call $\{v, w\}$ a *back edge* if v and w are related in the tree and a *cross edge* otherwise. We can determine in $O(1)$ time and $O(m)$ processors the cross edges and back edges, since a nontree edge $\{v, w\}$ with $v < w$ is a back edge if and only if w is a descendant of v , i.e. $w \leq v + nd(v) - 1$. We define $lowback(v)$ analogously to $low(v)$ but using only back edges, as follows:

$$lowback(v) = \min(\{v\} \cup \{lowback(v) \mid v \rightarrow w \text{ in } T\} \cup \{w \mid \{v, w\} \text{ is a backedge}\}).$$

We can compute $lowback$ just as we computed low , in the same resource bounds.

Now suppose G has no bridges. To convert G to a strongly connected directed graph, we direct the edges as follows:

- (i) If $\{v, w\}$ is a back edge with $v < w$, direct $\{v, w\}$ from w to v .
- (ii) If $\{v, w\}$ is a cross edge with $v < w$, direct $\{v, w\}$ from v to w .
- (iii) If $\{v, w\}$ is a tree edge with $v < w$, direct $\{v, w\}$ from v to w if $lowback(w) < w$ or if $low(w) \geq w$, and from w to v otherwise.

The intuition behind this construction is to direct back edges from descendant to ancestor and tree edges from parent to child. This suffices if there are no cross edges (i.e. the tree is a depth-first spanning tree). To handle the cross edges we direct them from lower to higher endpoint and reverse the natural directions of some of the tree edges as described in (iii).

THEOREM 2. *The directed graph formed by applying rules (i), (ii), and (iii) is strongly connected.*

Proof. We must show that every vertex is reachable from vertex 1 (the tree root) and vertex 1 is reachable from every vertex. We show that every vertex v is reachable from vertex 1 by induction on the preorder number of v . Obviously vertex 1 is reachable from itself. Suppose vertices $1, 2, \dots, v-1$ are reachable from vertex 1 and consider vertex v . There is some tree edge $\{u, v\}$ with $u < v$. If this edge is directed from u to v , then v is reachable from vertex 1. Otherwise, by rule (iii), $low(v) < v$ and $lowback(v) \geq v$. This means that there is a cross edge directed from $low(v)$ to some descendant of v , say x .

Vertex x is reachable from $low(v)$ and hence from 1 by the induction hypothesis. Furthermore, v is reachable from x by a directed path consisting of tree edges and back edges. Otherwise, let $y \neq v$ be the lowest ancestor of x and descendant of v reachable from v by such path. We know $low(x) \leq low(v) < v \leq x$. By rule (iii), it must be the case that $lowback(y) < y$; otherwise the tree edge $\{p(y), y\}$ is directed from y to $p(y)$, contradicting the choice of y . But this implies, also by rule (iii), that there is a directed cycle containing y and $p(y)$ consisting of a back edge from a descendant of y to $lowback(y)$ and all tree edges on the tree path between these vertices. This also contradicts the choice of y . We conclude that v is indeed reachable from x , and hence from 1. By induction all vertices are reachable from 1.

It remains for us to show that vertex 1 is reachable from every vertex. This will follow if we can prove that from any vertex $v \neq 1$ we can reach a vertex larger in postorder. If $lowback(v) < v$, there is a directed path from v to $lowback(v)$ by rule (iii). If $lowback(v) \geq v$ but $low(v) < v$, there is a directed edge from v to its parent. The only remaining possibility is $low(v) \geq v$. In this case $high(v) > v + nd(v) - 1$, i.e. $high(v)$ is not a descendant of v , for otherwise the tree edge $\{p(v), v\}$ would be a bridge. Let $\{x, high(v)\}$ be an edge connecting a descendant x of v to $high(v)$. This edge must be a cross edge, directed from x to $high(v)$. We claim that every descendant of v , including x , is reachable from v (by a directed path containing only descendants of v). This

follows from the fact that $low(v) \cong v$ using an argument like that used to prove that every vertex is reachable from vertex 1. Hence $high(v)$ is reachable from v . In all cases a vertex larger than v in postorder is reachable from v , and it follows that vertex 1 is reachable from every vertex. \square

Directing the edges according to rules (i)-(iii) takes $O(1)$ time using $O(m)$ processors once the vertices are numbered in preorder and low and $lowback$ are computed.

We close this section and the paper with a few remarks about future work. The parallel tree computations we have used may have applications to other graph problems. This deserves study. Also, there are still open problems concerning parallel biconnectivity algorithms. The algorithm of this section, as does the algorithm of Tsin and Chin [TC84], has optimal speed-up for dense graphs but not for sparse ones, whereas the algorithm of § 3 is off by a factor of $\log n$ from optimal speed-up. A question worth exploring is whether there is an $O((n+m)/p)$ -time algorithm using p processors, for p sufficiently small (say $p \leq (n+m)/\log^2 n$ or $p \leq (n+m)/\log n$.) Such an algorithm is unknown even for the problem of computing connected components.

Suppose that an algorithm of time $O((n+m)/p)$ could be found for the problem of computing connected components. Then the implementation of § 3 implies a block-finding algorithm of time $O((n \log n + m)/p)$ using $p \leq n \log n + m$ processors, provided we are given a proper input representation. In order to see this, consider the following representation of the input graph for the block-finding problem. The vertex set is $V = \{1, 2, \dots, n\}$. Each edge $\{i, j\}$ is represented by two directed edges (i, j) and (j, i) . The $2m$ directed edges of the graph appear in ascending lexicographic order in a vector of length $2m$. (That is, $(i_1, j_1) < (i_2, j_2)$ if $i_1 < i_2$ or $i_1 = i_2$ and $j_1 < j_2$.) Each vertex i has a pointer to its first outgoing edge. The implementation of § 3 still requires the following modification. Recall the construction of the list of outgoing edges in the tree for every vertex. This was done using doubling, which required $O(\log n)$ time using only $O(m/\log m)$ processors. Instead, we construct a sorted vector (similar to the input vector) of length $2n - 2$ that contains all directed edges of the tree. This takes time $O(\log n)$ using $O(m)$ processors: For each directed edge in the tree we need to find its serial number relative to the other directed edge of the tree. We use a balanced binary tree with $2m$ leaves, one for each input directed edge, to guide the computation, which is a standard partial sum computation where each active leaf enters one and gets in return its serial number relative to other active leaves. This is similar to the computation following Step 1 of this section. A similar remark applies to the computation of $localow(j)$ (just before the construction of the tree).

REFERENCES

- [AIS84] B. AWERBUCH, A. ISRAELI AND Y. SHILOACH, *Finding Euler circuits in logarithmic parallel time*, Proc. Sixteenth ACM Symposium on Theory of Computing, 1984, pp. 249-257.
- [AKS83] M. AJTAI, K. KOMLÓS, AND E. SZEMERÉDI, *An $O(n \log n)$ sorting network*, Proc. Fifteenth ACM Symposium on Theory of Computing, 1983, pp. 1-9.
- [At84] M. J. ATALLAH, *Parallel strong orientation of an undirected graph*, Inform. Proc. Letters, 18 (1984), pp. 37-39.
- [AV84] M. ATALLAH AND U. VISHKIN, *Finding Euler tours in parallel*, J. Comp. Sys. Sci., to appear.
- [Be73] C. BERGE, *Graphs and Hypergraphs*, North-Holland, Amsterdam, 1973.
- [BH82] A. BORODIN AND J. E. HOPCROFT, *Routing, merging and sorting on parallel models of computation*, Proc. Fourteenth ACM Symposium on Theory of Computing, 1982, pp. 334-338.
- [CLC81] F. Y. CHIN, J. LAM, AND I. CHEN, *Optimal parallel algorithms for the connected component problem*, Proc. 1981 International Conference on Parallel Processing, 1981, pp. 170-175.

- [Ec79a] D. M. ECKSTEIN, *Simultaneous memory access*, Technical Report TR-79-6, Computer Science Department, Iowa State Univ., Ames, Iowa, 1979.
- [Ec79b] ———, *BFS and biconnectivity*, Technical Report TR-79-11, Computer Science Department, Iowa State Univ., Ames, Iowa, 1979.
- [Ha69] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.
- [HCS79] D. S. HIRSCHBERG, A. J. CHANDRA, AND D. V. SARWATE, *Computing connected components on parallel computers*, *Comm. ACM*, 22 (1979), pp. 461–464.
- [M83] N. MEGIDDO, *Applying parallel computation algorithms in the design of serial algorithms*, *J. Assoc. Comput. Mach.*, 1983, pp. 852–865.
- [RV83] J. REIF AND L. J. VALIANT, *A logarithmic time sort for linear size networks*, *Proc. Fifteenth ACM Symposium on Theory of Computing*, 1983, pp. 10–16.
- [SJ81] C. SAVAGE AND J. JA'JA', *Fast, efficient parallel algorithms for some graph problems*, this Journal, 10 (1981), pp. 682–691.
- [SV81] Y. SHILOACH AND U. VISHKIN, *Finding the maximum, merging, and sorting in a parallel computation model*, *J. Algorithms*, 2 (1981), pp. 88–102.
- [SV82] ———, *An $O(\log n)$ parallel connectivity algorithm*, *J. Algorithms*, 3 (1982), pp. 57–63.
- [TA72] T. E. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.
- [Ta74a] ———, *Finding dominators in directed graphs*, this Journal, 3 (1974), pp. 62–89.
- [Ta74b] ———, *A note on finding the bridges of a graph*, *Inform. Proc. Letters*, 2 (1974), pp. 160–161.
- [Ta82] ———, *Graph partitions defined by simple cycles*, Technical Memorandum, AT&T Bell Laboratories, Murray Hill, NJ, 1982.
- [TC83] Y. H. TSIN AND F. Y. CHIN, *A general program scheme for finding bridges*, *Inform. Proc. Letters*, 17 (1983), pp. 269–272.
- [TC84] ———, *Efficient parallel algorithms for a class of graph theoretic problems*, this Journal, 13 (1984), pp. 580–599.
- [Th83] C. D. THOMPSON, *The VLSI complexity of sorting*, *IEEE Trans. Comput.*, C-32 (1983), pp. 1171–1184.
- [Ts82] Y. H. TSIN, *A generalization of Tarjan's depth first search algorithm for the biconnectivity problem*, Dept. Computing Science, Univ. Alberta, Edmonton, Alberta, Canada, 1982.
- [TV84] R. E. TARJAN AND U. VISHKIN, *Finding biconnected components and computing tree functions in logarithmic parallel time*, *Proc. 25th Annual IEEE Symposium on Foundations of Computing*, 1984, pp. 12–20.
- [V81] U. VISHKIN, *An optimal parallel connectivity algorithm*, Technical Report RC 9149, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1981; *Disc. Appl. Math.*, to appear.
- [V83a] ———, *Implementation of simultaneous memory address access in models that forbid it*, *J. Algorithms*, 4 (1983), pp. 45–50.
- [V83b] ———, *$O(\log n)$ and optimal parallel biconnectivity algorithms*, Technical Report # 69, Computer Science Dept., New York Univ., New York, 1983.
- [V83c] ———, *Synchronous parallel computation—a survey*, Technical Report # 71, Computer Science Dept., New York Univ., New York, 1983.
- [V84] ———, *An efficient parallel strong orientation*, Technical Report # 109, Computer Science Dept., New York Univ., New York, 1984.
- [Wi75] S. WINOGRAD, *On the evaluation of certain arithmetic expressions*, *J. Assoc. Comput. Mach.*, 22 (1975), pp. 477–492.
- [Wy79] J. C. WYLLIE, *The complexity of parallel computation*, Technical Report TR 79-387, Dept. Computer Science, Cornell Univ., Ithaca, NY, 1979.

DISTRIBUTED MULTI-DESTINATION ROUTING: THE CONSTRAINTS OF LOCAL INFORMATION*

JEFFREY M. JAFFE†

Abstract. In computer networks, message routing is often accomplished by network nodes using local information. The unavailability of global information intuitively makes hard routing problems virtually impossible. This paper formalizes this intuition by examining a hard (NP-complete) routing problem, the problem of multi-destination routing. It is shown that with only limited information it is *impossible* to optimize network utilization for the multi-destination routing problem. Moreover, it is impossible to even approximate optimality to within a specific tolerance. Several versions of this result are proved; the versions differ in terms of the amount of information available at a node, and the extent to which the problem cannot be approximated. An improved local information algorithm is presented which is best possible amongst local information algorithms.

Key words. computer networks, distributed algorithms, routing, Steiner trees, spanning trees

1. Introduction. In computer networks, the nodes of a network cooperate to accomplish network service functions such as routing and flow control [1]. Each node initially has a local view of the network, and through distributed protocols the nodes enlarge their view of the network. However, due to the potentially large amount of information about the network, it is rare that each node has global topological information. In this paper we investigate the impact of nonglobal knowledge in the context of message routing.

Message routing has tended to be from a single source node to a single destination node. With the advent of office communications, there are likely to be applications whereby a memo needs to be sent from a single source node to multiple destinations. This paper focuses on the limitations of local information when a "multi-destination routing algorithm" attempts to minimize the (weighted) number of links traversed. In particular, we prove that with limited information, the multi-destination routing problem cannot even be approximated.

Routing to *all* destinations has been studied by some authors (e.g. [2]), but studies on routing to a proper subset of all network nodes are relatively few [3], [4]. Nevertheless, the mathematical equivalent of multi-destination routing (the Steiner tree problem) has been extensively studied (e.g. [5]).

Our intention here is not to devise polynomial time heuristics for the Steiner tree problem, but rather to rigorously analyze the constraints of limited information as it relates to computing Steiner trees. Thus our emphasis is on lower bounds; evaluating the quality of algorithms that are restricted in the amount of information available to them.

Thus, in this work we focus primarily on algorithms executed by the source node to route to multiple destinations with *local*, *nearby* or *destination* information. With local information, the source may route "optimally" to any single destination; with nearby information; the source obtains the local information of its neighbors; and with destination information, the source obtains (by querying the destinations) a subset of the local information of all of the destinations. It is shown (in § 3) that all local and nearby information algorithms result in routings which are at least $2m/3$ times worse than optimal in the worst case (routing to m destinations). This serves to explain

* Received by the editors April 29, 1982, and in final revised form June 15, 1984.

† IBM T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.

why the best known algorithms are no better than $2m/3$ times worse than optimal. Any destination information algorithm is at least twice worse than optimal. This explains why the best known algorithms are no better than twice worse than optimal [3], [5].

While the primary emphasis is on lower bounds, the evaluation of limited information constraints has led to the discovery of a new Steiner tree heuristic whose performance ($2m/3$) is surprisingly good considering the fact that it uses only local information by the source node. This algorithm is described in § 4.

In § 5 we generalize the lower bound results to environments where the source node cooperates with certain intermediate nodes to determine how to route the messages. Even in that case, "destination" information does not suffice to do better than twice worse than optimal in the worst case.

In a number of related papers (on locating a center [6], address assignment [7], and flow control [8]), it has been shown that limited information does not permit one to optimize certain functions. Here, we show that the multi-destination routing problem cannot even be approximated. This is similar to results in [9] where it is shown that "limited" information does not even permit the approximation of certain scheduling problems.

2. Definitions. A computer communications network is modeled as an undirected graph with node set N and link set L . With each $l \in L$, a cost $c(l) \in \mathbb{N}$ specifies the desirability of using link l for routing messages (lower cost means more desirable).

An instance of a *multiple destination routing problem* consists of a graph (N, L) , a *source* node $s \in N$ and a set of *destination* nodes $D \subseteq N$ with $s \notin D$. A *routing* R , for an instance of the problem is a rooted directed multigraph that contains:

- (1) a node set that is a subset of N ;
- (2) a root s ;
- (3) all nodes of D ;
- (4) an arbitrary subset of $N - D$.

When designing algorithms in which network nodes use only limited information, it is sometimes impossible for a node to construct the precise link by link routing since it lacks sufficient information. Nonetheless, in practice a node with limited information can guide the selection of the routing by specifying in which order destinations are to be routed. In particular, a node may construct a multigraph which contains s and D and some edges between nodes in the multigraph. The convention is that any edge between two nodes represents the shortest path between them in the original graph if they are not connected in the original graph. If they are connected, the routing also explicitly states (by labeling an edge with a "1 or 0") if the connection is to be via the shortest path between them or by the direct edge between them. In general "guiding" based on shortest paths is commonly used for single destination routing in the sense that a source node often does not know the entire shortest path but just the first node on the path, yet it knows the destination.

Thus the multigraph specifies the order in which nodes are traversed, and implies the use of the shortest paths between them. Note that since this abbreviated form of the routing is used, when interpreted as a set of paths from the source in the network, it may actually no longer be a tree at all due to multiple traversals of links.

Given a routing one must decide how to assign a cost to a routing, to compare it to other routings. Several possibilities suggest themselves. In one possibility, the optimality criteria would be to minimize the average cost (delay) "experienced" by each destination. That would model the situation where a short message must be routed

quickly to each destination. However, in the practical environment that motivated this research, the actual problem was to distribute rather lengthy documents through the network. In that case average delay was less important (it was not “interactive” type traffic), but total network link utilization was important. This motivates the definition of *network cost* (NC) of a routing R to be defined by

$$NC(R) = \sum_{\text{edges, } e \text{ in } R} sd(e),$$

where $sd(e)$ is the shortest distance in (N, L) between the two nodes adjacent to the edge e in R . If the edge e represented a direct link in the original graph, then $sd(e)$ is taken to be the cost of that link.

Let $f: N \times L \times s \times D \times c(L) \rightarrow \mathbf{N}$ be a function that takes *all* of the parameters of a multiple destination routing problem and transforms (or reduces) them into some integer (or tuple of integers represented as an integer in some canonical way). An *f-information multi-destination* routing algorithm is a function g such that g provides a routing for any instance of a multiple destination routing problem and such that the routings of two problems are identical when the “ f -values” of the problems are the same.

A *local* information algorithm is an algorithm in which a node “uses” its entire shortest path to every node in the network and the costs of those paths. The idea behind local information is that it is essentially the minimal information that is needed to do any sensible routing calculation by a node. It must at least know some way to get to each destination, and the shortest path is something which is easily calculable and often calculated in networks in a distributed manner [10], [11]. In particular what a node has available for every other node in the network is the “next node” on the path to the other node as well as the total cost to that node. In case of ties, the node may know each potential next node.

The formal way in which local information algorithms fit into the f -information formalism is as follows: A function, f^1 on $N, L, s, D,$ and $c(L)$ is defined which maps the input parameters into the “shortest path table at s ” (see Table 1) and a listing of the destination, D . Next any encoding function is used to map the shortest path table and the names of elements of D into some integer format (e.g., concatenating all table

TABLE 1
Sample shortest path table at s .

| Network node | Next node | Cost |
|--------------|-----------------------|------|
| N_1 | Neighbor ₁ | |
| N_2 | Neighbor ₂ | |
| \vdots | \vdots | |
| s | — | 0 |
| \vdots | \vdots | |
| N_x | | |
| \vdots | \vdots | |
| $N_{ N }$ | Neighbor ₁ | |

entries into a single integer) to yield the desired function f . A local information algorithm (presumably executed by s) then determines a routing based only on this shortest path information, and will construct a routing. The same routing must arise from any situation in which s has the same set of shortest path tables. The actual routing in the network may differ due to differences in the way that edges in the routing are interpreted as paths in the network.

A *nearby* information algorithm is one in which a node uses its own local information and its neighbors' local information. In such algorithms the node communicates with adjacent nodes to obtain additional information. In [3] simulation studies show that nearby information algorithms perform better than local information algorithms. No cost is assigned to the communication cost of acquiring the neighbors' tables for two reasons. First of all, if one assumes that network topology (i.e. the graph structure) does not change too often, then a node may permanently store its neighbors' tables. Furthermore, since the application environment assumes very large file transmissions, the cost of accumulating one's neighbors' shortest path tables would be dwarfed, anyway by the actual file transmission costs.

The *formal* method of defining nearby algorithms is a trivial modification of the formalism for local algorithms, and we do not elaborate in detail. Basically, the function f maps N, L, s, D and $c(L)$ into N using some standard concatenation of the shortest path tables of s, s 's neighbors and a listing of elements of D .

In routing large (resource consuming) files, a node might be allowed to accumulate distant information. A natural place from which to accumulate it (without learning the whole topology) is from the destinations. We define *destination* information algorithms to be those in which a node has both its own local information, plus the shortest paths (and costs) between every pair of destination nodes and from the source to each destination. Basically, destination information can be obtained by querying the destinations about their paths, before sending the message. Once again we do not charge a node for the cost of obtaining the destination information. For one thing, as indicated earlier, any information accumulation may be considered cheap for routing a large file. Moreover, in the above definition, one does not obtain the full shortest path table from each destination. This, indeed would be costly— $O(mn)$ entries for $n = |N|$ and $m = |D|$. Only the shortest path information between destinations ($O(m^2)$) is collected here.

Destination information algorithms for a node n are placed into our formalism as follows. A standard encoding (as above) is taken of n 's shortest path tables, as well as a table of shortest paths between every pair of destinations.

The *performance* of an algorithm is the worst case value of

$$\frac{NC(R)}{NC(R^*)}$$

where R is the routing chosen by the algorithm and R^* is the optimal routing (i.e., the one with smallest value of NC).

To place the results of the next section in perspective, we cite some positive results from [3]. There is a local (and nearby) information algorithm whose performance (as a function of m , the number of destinations) is equal to m , and a destination information algorithm whose performance is 2. In the next section we show that in fact *all* local (and nearby) information algorithms must have performance at least as bad as $2m/3$, and *all* destination information algorithms must have performance at least as bad as 2. Thus it is impossible to improve the local information algorithms by much, and the

destination information algorithm of [3] is, in a sense, the best possible for the practical algorithms under consideration. Moreover in § 4 we introduce a new local information algorithm with performance of $2m/3$ to tighten the corresponding lower bound of § 3.

3. Limited information source node algorithms. In this section we explore the limitations that occur when the source node has only local, nearby or destination information. With local or nearby information, any algorithm is at least as bad as $2m/3$. With destination information, every algorithm must be at least twice optimal. The first result is proved for the nearby information case—the local information case follows automatically.

THEOREM 1. *Any algorithm in which the routing is determined by the source node using only nearby information has worst case performance of at least $(2m/3) - \delta$ for any $\delta > 0$.*

Proof. This result is proved by concentrating on the situation where the cost to each destination from the source node equals $1 + \epsilon$. The cost through its single neighbor, N , is 1. (The neighbor is ϵ distance from the source node.) Also, there are no nodes in the network other than D , s and N , with $s, N \notin D$ (see Fig. 1). It will be shown that under these very restrictive circumstances, no algorithm can approximate R^* . To accomplish this, we fix any given nearby information algorithm and analyze its behavior.

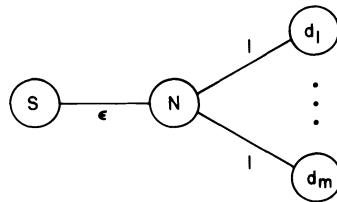


FIG. 1

The first question we ask is: “What routing is chosen by s in this case?” There is a unique routing R generated by the algorithm, irrespective of the internal interconnection pattern of D due to the fact that the local information of both s and N is independent of the internal interconnection pattern of D . Without loss of generality it goes from s to N and then somehow to the nodes of D . The key to the proof is to show that whatever routing is chosen is $2m/3$ times worse than optimal for *some* configuration of the nodes of D .

Name the m destinations d_1, \dots, d_m and let Fig. 2 represent the routing chosen by s in this instance (i.e., when its nearby information is specified by Fig. 1). Every such routing is essentially a tree rooted at N which indicates the directions in which

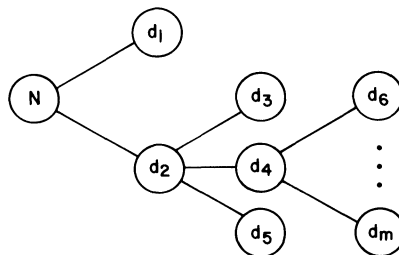


FIG. 2

the messages are sent. For example, Fig. 3 gives the special case of Fig. 2 in which the routing generated by s is that each node receive the message on the shortest path from s .

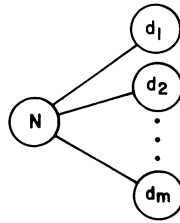


FIG. 3

Those nodes which are connected directly to N in the routing (d_1 and d_2 in Fig. 2) are referred to as *first level nodes*. Those nodes at distance i from N in the routing are called *i th level nodes*. It will be shown that any routing given in Fig. 2 is ineffective by specifying an interconnection of D based on the levels of the nodes.

Consider Fig. 4 which indicates that all m destinations are roughly speaking in two clusters both at distance 1 from N . Nodes within each cluster are within cost ϵ of each other, $\epsilon \ll 1$.

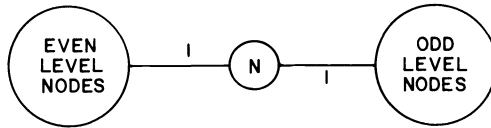


FIG. 4

Clearly $NC(R^*)$ for the network of Fig. 4 is $2 + \delta$ for some $\delta \ll 1$. However, the routing, R , generated by the algorithm (Fig. 2) usually does much worse. The cost from N to each first level node is 1. The cost from each i th level node ($i = 1, \dots$) to its $(i + 1)$ st level nodes that it connects to is 2. If there are k first level nodes and $m - k$ nodes at other levels, then $NC(R) = 2(m - k) + k = 2m - k$. Thus

$$(1) \quad \frac{NC(R)}{NC(R^*)} = \frac{2m - k}{2 + \delta}$$

To get the result of $2m/3$ stated in the theorem, we consider the configuration of Fig. 5 where all costs are ϵ within the cluster. In that case $NC(R^*) \leq 1 + \delta'$. Once again, if R is the routing with k first level nodes, $NC(R) \geq k + \delta''$ for the configuration of Fig. 5. Thus

$$(2) \quad \frac{NC(R)}{NC(R^*)} \geq \frac{k + \delta''}{1 + \delta'}$$

Combining (1) and (2) proves the result of Theorem 1. If $k \geq 2m/3$, then in the network of Fig. 5, $NC(R)/NC(R^*) \geq (2m/3) - \delta'''$ for an appropriate δ''' . If $k < 2m/3$

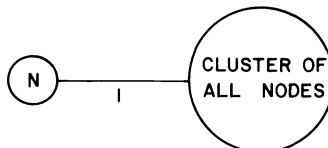


FIG. 5

then in the network of Fig. 4,

$$(3) \quad \frac{NC(R)}{NC(R^*)} \geq \frac{2m-k}{2+\delta} \geq \frac{2m-2m/3}{2+\delta} = \frac{2m}{3} - \delta'''$$

for an appropriate δ''' (chosen as a function of m).

It is considerably more difficult to arrive at lower bounds on classes of algorithms with more information available and intermediate node computation. As such, we will slowly build up a series of more difficult results using similar constructions. We begin by continuing our restriction to source node algorithms.

THEOREM 2. *Any algorithm in which the source node calculates the routing using only destination information has performance which is at least as bad as $(2m/(m+1)) - \delta$ for any $\delta > 0$.*

Proof. Consider the network of Fig. 6. The optimum routing is through intermediate node I_0 with $NC = 5 + 5\epsilon$. If s routes directly to each destination then NC equals 8. If there were m destinations configured in a similar manner, $NC(R^*) = (m+1)(1+\epsilon)$ but $NC(R) = 2m$, where R is any routing that bifurcates (i.e., splits to two or more different nodes) only at s or nodes in D . The result of the theorem is thus proved, unless s chooses a routing that bifurcates at I_0 .

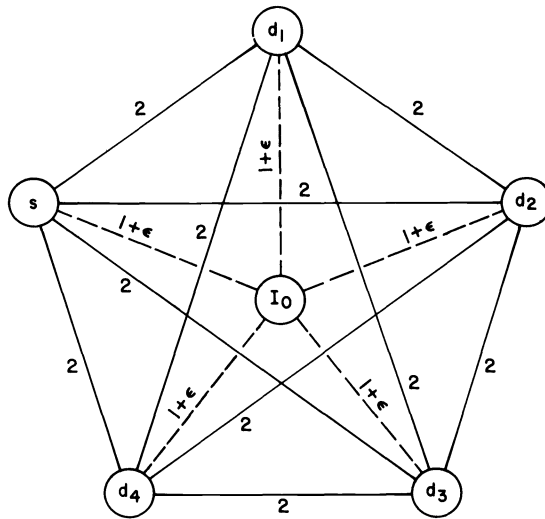


FIG. 6

Let us assume then that s chooses to bifurcate at I_0 . All s knows about I_0 (from its local information) is that I_0 is some node at distance $1 + \epsilon$ away from s . Additionally, s learns nothing new about I_0 from its destination information. Now, consider Fig. 7, which is identical to Fig. 6 except that there are k nodes I_1, \dots, I_k that are identical to I_0 from the point of view of s 's information. If s chooses to route to I_0 and then "use" the shortest paths from I_0 to the destinations (by specifying only I_0 as its next node in the routing), then the routing is at least as bad as $2m/m+1$ as long as the nodes are renumbered so that I_0 is not the central one. For if the routing ends up as in Fig. 8, NC could be as large as $m(3+\epsilon) + 1 + \epsilon$ by bifurcating at I_j . Similarly, any use of a combination of the I_j 's does not get NC to be smaller than $2m$. (Note that k is chosen to be a large function of m .) However, if the I_j 's are not used, then (as in Fig. 6), the performance is still at least as bad as $2m/m+1$.

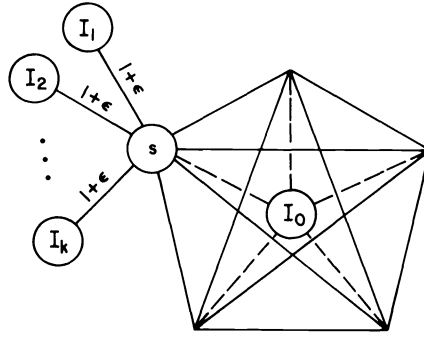


FIG. 7

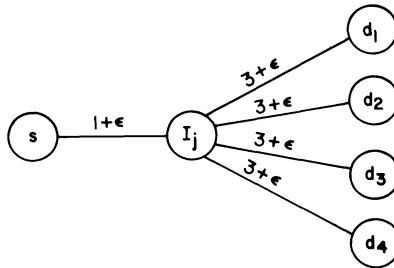


FIG. 8

4. A new local information algorithm. Theorem 2 is best possible in the sense that there is a destination algorithm with performance of $2m/(m+1)$ [3]. However, the best known local information algorithm has performance m [3], leaving a gap between the lower bound of Theorem 1 and the best known local information algorithm. This section provides the result that there is a source node, local information algorithm with performance $2m/3$. This is despite the intuition that one might expect that with only local information, the source can do no better than to send to each destination on its shortest path (with performance m).

The intuition for the algorithm comes from the proof of Theorem 1. In that proof, it was essentially shown that s should not send to *all* destinations directly on their shortest paths since they may all be very close to each other. On the other hand, if s *assumes* that they are very close to reach other, they may in fact be far away. In the new algorithm s will assume that some $(m/3)$ destinations are close to each other, but most $(2m/3)$ are not. In this way, if all nodes are close to each other, the performance will improve (over shortest path routing) due to the $m/3$ nodes that s assumed were close. On the other hand, if the nodes are not close, the performance will be good due to the $2m/3$ nodes that s assumed were not close.

To describe the algorithm, let $D = \{d_1, \dots, d_m\}$ be the destination set. Define $c(a, b)$ to be the cost of the minimal cost path from node a to node b . Node s will choose a routing based only on the relative values of $c(s, d_i)$ which are available in s 's local information. For simplicity, we assume in the rest of this section that m is divisible by 3.

Without loss of generality, assume that the d_i are indexed such that $c(s, d_1) \leq c(s, d_2) \leq \dots \leq c(s, d_m)$. The routing that s chooses is to send to each of the closest $k = 2m/3$ destinations directly on the shortest paths to them. The other $m/3$ are routed to in a chain starting from d_k . The order in which they appear in the chain is actually

arbitrary, but for definiteness we assume an order of $d_k, d_{k+1}, d_{k+2}, \dots, d_m$. That is, d_k routes the message to d_{k+1} , who forwards it to d_{k+2} , and so forth until d_m (see Fig. 9). Note that it is quite possible that the routing will not be a tree in the original graph.

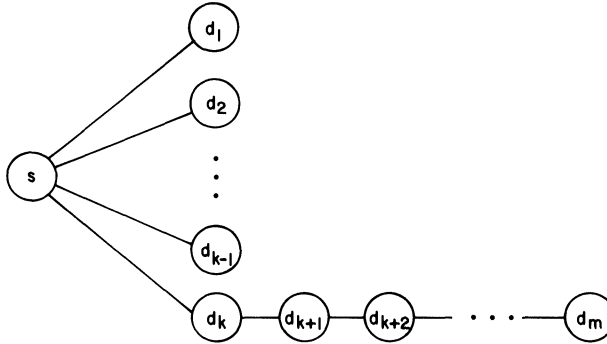


FIG. 9. New local information algorithm.

THEOREM 3. *Given an instance of the multi-destination routing problem, the above local information, source node algorithm has performance $2m/3$.*

Proof. Given any instance of the problem, from the algorithm, we can write down

$$(4) \quad NC(R) = \sum_{i=1}^k c(s, d_i) + \sum_{i=k}^{m-1} c(d_i, d_{i+1}).$$

Since d_k is the furthest of the first k destinations, we may observe

$$(5) \quad NC(R) \leq \frac{2m}{3} c(s, d_k) + \sum_{i=k}^{m-1} c(d_i, d_{i+1}).$$

Consider the $m - k$ costs from d_k to d_m . Let j be the index which has the largest of those costs (i.e., $c(d_j, d_{j+1}) \geq c(d_i, d_{i+1})$ for $i = k, \dots, m - 1$). Then by our choice of j , we may write

$$(6) \quad NC(R) \leq \frac{2m}{3} c(s, d_k) + \frac{m}{3} c(d_j, d_{j+1}).$$

The technique that will be used to prove Theorem 3 will be to examine the optimal routing R^* for the instance of the problem being studied and to locate d_j, d_{j+1} and d_k in that routing (see Fig. 10).

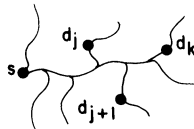
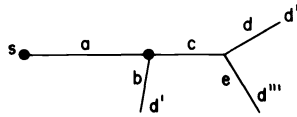


FIG. 10. Optimal routing.

Given the optimal routing R^* , define R' to be the subtree of R^* consisting only of the paths from s to d_j, d_{j+1} and d_k . Note that this tree may consist of three distinct chains (if the paths do not intersect) or may contain only one or two links leaving s if the paths intersect at some point. One way to describe R' is as follows. The tree is routed at s , and the routing begins for some number of links (possibly none) as a single chain. At some node, the chain splits into two or three pieces with the path for

one of d_j, d_{j+1} and d_k going off in one direction or stopping at that node, and the path for the other two going off in the second direction (or all three in different directions). Finally (if two stayed together), there is a splitting off of the final two nodes (see Fig. 11 — d', d'', d''' are placeholders for d_j, d_{j+1}, d_k).



In Fig. 11, a, b, c, d, e represent the costs of the various path segments. The case $a = c = 0$ is the case that R' is three separate chains. If $a \neq 0$ and $b = 0$, then d' is on the path to d'' and d''' , if $c = 0$, three path segments split at a single point; if $d = 0$ then d'' is on the path to d''' and so forth. A case by case analysis on the position of d_k will prove Theorem 3.

Case 1. $d' = d_k$. In this case $c(s, d_k) \leq a + b$ since at least one path at cost $a + b$ goes from s to d_k . Similarly $c(d_j, d_{j+1}) \leq d + e$. Thus, from (6),

$$NC(R) \leq \frac{2m}{3}(a + b) + \frac{m}{3}(d + e).$$

Since

$$NC(R^*) \geq NC(R') \geq a + b + c + d + e,$$

it follows immediately that

$$NC(R)/NC(R^*) \leq 2m/3.$$

Case 2. $d' \neq d_k$. Assume, without loss of generality, that $d_k = d''$. Then $c(s, d_k) \leq a + c + d$. Also $c(d_j, d_{j+1}) \leq b + c + e$.

Note that $c(s, d_k) \leq c(s, d')$ since $c(s, d_k) \leq c(s, d_j) \leq c(s, d_{j+1})$. Thus $c(s, d_k) \leq a + b$. It follows that $c(s, d_k) \leq \min\{a + b, a + c + d\} \leq \frac{1}{2}((a + b) + (a + c + d))$. Thus,

$$\begin{aligned} NC(R) &\leq (2m/3)(\frac{1}{2})((a + b) + (a + c + d)) + (m/3)(b + c + e) \\ &= (m/3)(2a + 2b + 2c + d + e) \leq (2m/3) NC(R^*). \end{aligned}$$

Thus although no local information, source node algorithm can perform better than $2m/3$, this algorithm achieves that bound—and has better worst case performance than shortest path routing.

5. Intermediate node algorithms. Until now we have assumed that from some information base, the source node calculates the entire routing. In fact, although the “routing” is calculated by the source, the actual node by node message transfer is done with the assistance of intermediate nodes. Each node only knows the *next node* on the shortest path to each destination; the next node then forwards the message to its next node and so forth. In this section, we generalize the assistance provided by intermediate nodes to include assistance in the actual determination of the routing. Intermediate node algorithms often have more capability than source node algorithms. In particular, the “nearest neighbor multi-destination routing algorithm” (an intermediate node, local information algorithm) has better worst case performance than the shortest path algorithm (a source node, local information algorithm) [3]. In the next

section we show that it is still impossible to have performance better than $2m/(m+1)$, even if intermediate nodes participate in the routing and all nodes have destination information. We first formally define this class of algorithms.

A *routing choice* by a node, N , with *responsibility* for a destination set D is a partition of D into k subsets D_1, D_2, \dots, D_k , and an assignment of *responsibility* for each subset, to neighbors N_1, \dots, N_k of N . The *complete routing initiated by N* in a particular instance of a multi-destination routing problem in which N has responsibility for D as above and assigns responsibility to N_1, \dots, N_k is defined in the following recursive fashion:

- (1) The root is N .
- (2) Nodes N_1, \dots, N_k are one link away from N .
- (3) Rooted at each of N_1, \dots, N_k is the complete routing initiated by N_1, \dots, N_k (for D_1, \dots, D_k).

Note if D is empty, the complete routing is N itself. If the various nodes determine routing choices in sufficiently uncoordinated fashions, the complete routing may be infinite! Even if the structure is finite, the set of paths traversed may not form a tree in the original graph, and the message may traverse the same link multiple times. In these cases, NC is counted slightly differently according to the actual utilization of the communication links. In particular NC for an instance (with source s) is the sum of the weights of the links in the complete routing of s (here weights are as defined in the original graph).

An *intermediate node algorithm* is one with the property that every node (including the source) accepts responsibility for a given destination set and makes a routing choice.

An *intermediate node, destination information algorithm* is an intermediate node algorithm with the property that the routing choice made by any node, N , depends only on the destination information of the list of destinations that N has responsibility for. That is, N 's f -information consists of its local information and the shortest path information between destinations for which it is responsible.

Another class of algorithms that may be of interest are those in which every node passes its accumulated information set to its neighbors when it transfers responsibility. A *historical intermediate node, destination information algorithm* is one where the routing choice may depend on the destination information of any predecessor of N in the routing. This gives N the advantage of seeing the local information of N 's predecessors.

6. Limitations of intermediate node algorithms. This section completes our discussion by showing that even potentially powerful sets of information are insufficient from the point of view of approximating NC.

THEOREM 4. *Any intermediate node destination information algorithm has performance which is at least $(2m/(m+1)) - \delta$ times worse than optimal.*

Proof. Consider Fig. 12 which is a superset of the network of Fig. 7. If s tries to do better than $2m/(m+1)$ by trying to route to I_{00} at distance $1 + \epsilon$ (trying to give responsibility to I_{00}), it may end up being sent not to I_{00} but to some other node, I_{jk} . In that case, I_{jk} has no choice but to return the message to s . But then s has the same information as it did originally and will send it back to I_{jk} and the message will loop and never get anywhere.

On the other hand, if s sent responsibility to I_{jk} for a subset of D , then looping forever need not occur since s 's destination set has changed. But in that case, s has incurred a cost of $2 + 2\epsilon$ without really accomplishing anything. For further exploration of the I_{jk} 's only incurs cost without serving to locate I_{00} . In a similar manner, each receiving destination cannot afford to send it to one of its neighboring I_{jk} 's so it must

send it directly to the other destinations. This continues until the entire routing is completed at cost 2 per destination or $2m$.

This result is slightly unsatisfying as it disallows the use of information gathered at the source, by the source's neighbors. The generalization of destination information intermediate node algorithms to the case that histories are allowed would solve the problem of the example of Fig. 12. The next result shows, however, that even histories do not help in all cases.

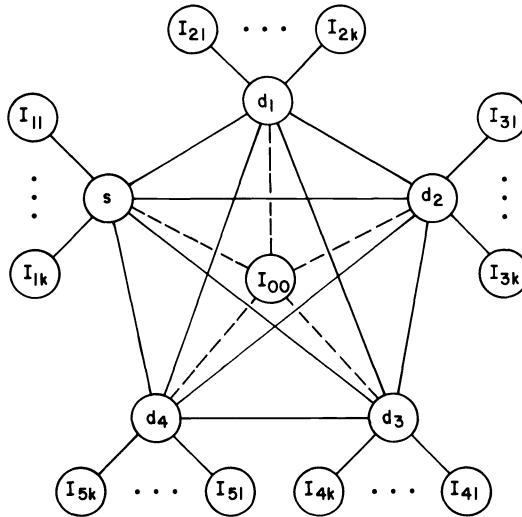


FIG. 12

THEOREM 5. Any historical, intermediate node, destination information algorithm has performance which is at least $(2m/(m + 1)) - \delta$ times worse than optimal.

Proof. We start again with Fig. 12, and argue that s cannot afford to “explore” its neighboring I_{jk} 's. As long as there are many of them (let's say more than $2m$ of them), in the worst case s will explore them all at excessive cost without having made any progress on the routing. Thus, for reasons similar to those in previous proofs, the routing essentially must start by s sending the message directly to one or many destinations.

It is at this point that the proof becomes more complicated than that of Theorem 4. Consider for example the construction of Fig. 12, and assume that s transfers all responsibility to d_1 . Since d_1 has access to s 's destination (and local) information, d_1 can distinguish between I_{00} and the I_{2j} 's — I_{00} is the only one at distance $1 + \epsilon$ from both s and d_1 . Thus in Fig. 12, d_1 could use this information to route to I_{00} , which then routes directly to the other destinations.

The construction that shows that such routings do not work in general is quite complex to be presented all at once, so we first give an example which shows that routing from s to d_1 to I_{00} to the other destinations does not work in general. Consider Fig. 13 which is the same as Fig. 12, augmented with nodes J_1, \dots, J_L at distance $1 + \epsilon$ from s and d_1 . These nodes are indistinguishable from I_{00} for d_1 if all that d_1 has is s 's and d_1 's local and destination information. The presence of the nodes effectively prevents the usage of I_{00} , unless more information is available than that of s , d_1 , the J_i 's, the I_{1k} 's and the I_{2k} 's. Again, d_1 essentially must route directly to other destinations.

The general proof continues along the lines described above. Fix an algorithm, and, assume that when s sees a collection of nodes $1 + \epsilon$ distance away, and a set of destinations with a distance of 2 away, that it routes to d_1 . (We have essentially shown that s must do so.) Assume further that if d_1 sees a collection of nodes that are $1 + \epsilon$ away both from d_1 and from s , and a set of destinations at cost 2 away, that it routes to d_j (again there is no choice!). Then the next step in the construction is to place a large number of nodes at distance $1 + \epsilon$ from s , d_1 and d_j . Then d_j will have no choice but to route directly to another destination. Continuing the construction forces the algorithm to require at least cost 2 for every destination.

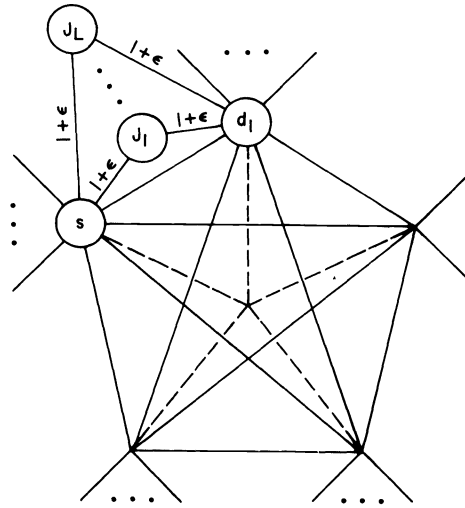


FIG. 13

In fact, a single construction simultaneously proves that all historical intermediate node, destination information algorithms are at least twice worse than optimal, as we proceed to sketch. The construction starts with s , D and I_{00} as in Fig. 6. In addition, for every subset $S \subseteq \{s\} \cup D$, there is a large collection of nodes at distance $1 + \epsilon$ from every node in S , and not connected to any other node. Inductively assume that only nodes in D ever receive responsibility. Then if $d \in D$ has to make a routing choice after a subset S has previously been on the path to d (and thus have contributed their information), d cannot seek out I_{00} since it would only find the blind alley corresponding to $S \cup \{d\}$. Thus it must route directly to some destination and the induction is verified.

We remark that the construction can be enhanced to show that nearby information does not help either. As in the proof of Theorem 1, one needs to place neighbors very close to s (and to the nodes of D). The rather messy details of this construction are left to the reader.

7. Summary. This paper has explained why it is hard to approximate the multiple destination routing problem using various forms of local information. Under various sets of assumptions, we have shown that the best known algorithms [3] that use certain types of information are about as good as possible. Table 2 summarizes these results. Open problems include tightening the bounds of Table 2 for intermediate node, local algorithms.

TABLE 2

| | Best known algorithm | Best possible algorithm |
|---------------------------------------|----------------------|-------------------------|
| Source, local | $2m/3$ | $2m/3$ |
| Source, nearby | $2m/3$ | $2m/3$ |
| Source, destination | 2 | 2 |
| Historical, intermediate, local | $1 + \log m$ | 2 |
| Historical, intermediate, nearby | $1 + \log m$ | 2 |
| Historical, intermediate, destination | 2 | 2 |

Acknowledgment. The author would like to acknowledge useful discussions with Alan Baratz and Bharath Kadaba.

REFERENCES

- [1] M. SCHWARTZ, *Computer Communication Network Design and Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1977.
- [2] Y. K. DALAL AND R. M. METCALFE, *Reverse path forwarding of broadcast packets*, *Comm. ACM*, 21 (1978), pp. 1040-1048.
- [3] K. BHARATH-KUMAR AND J. M. JAFFE, *Routing to multiple-destinations in computer networks*, *IEEE Trans. Comm.*, COM-31, (1983), pp. 343-351.
- [4] D. WALL, Ph.D. Thesis, Stanford Univ., Stanford, CA, 1980.
- [5] E. N., GILBERT AND H. O. POLLAK, *Steiner minimal tree*, *SIAM J. Appl. Math.*, 16 (1968), pp. 1-29.
- [6] D. ANGLUIN, *Local and global properties in networks of processors*, *Proc. Twelfth Annual ACM Symposium on Theory of Computing*, 1980, pp. 82-93.
- [7] S. R. SOLOWAY, *On the node-numbering problem*, *Proc. Eighteenth Annual Allerton Conference on Communication, Control, and Computing*, 1979, pp. 514-523.
- [8] J. M. JAFFE, *Flow control power is non-decentralizable*, *IEEE Trans. Comm.*, 24 (1981), pp. 1301-1306.
- [9] E. DAVIS AND J. M. JAFFE, *Algorithms for scheduling tasks on unrelated processors*, *J. Assoc. Comput. Mach.*, 28 (1981), pp. 721-736.
- [10] M. SCHWARTZ AND T. E. STERN, *Routing techniques used in computer communication networks*, *IEEE Trans. Comm.*, Special Issue on Computer Network Architectures and Protocols, COM-28 (1980), pp. 539-552.
- [11] J. M. MCQUILLAN, G. FALK AND I. RICHER, *A review of the development and performance of the ARPANET routing algorithm*, *IEEE Trans. Comm.*, COM-26 (1978), pp. 1802-1811.

FINDING MINIMAL PASS SEQUENCES FOR ATTRIBUTE GRAMMARS*

HENK ALBLAS†

Abstract. The generally used pass oriented evaluation strategies for attribute grammars, where different instances of the same attribute in any derivation tree are restricted to be evaluated in one left-to-right or right-to-left pass with for each derivation tree the same pass number, are referred to as simple multi-pass strategies.

The determination of a shortest sequence of pass directions, such that an associated distribution of the attributes over the passes meets the simple multi-pass requirements, is known to be NP-complete.

A polynomial time algorithm is discussed that delivers a shortest sequence of pass directions for several simple multi-pass grammars. It starts from a given sequence of pass directions and tries to find a shorter one by considering possible distributions of the attributes over the passes and crossing out empty passes. An extension of the algorithm also reconsiders and possibly changes directions in order to find passes that become empty. The resulting sequences of pass directions are minimal with respect to the subsequence ordering, i.e., no subsequence exists for which an associated distribution of the attributes over the passes meets the simple multi-pass requirements.

The algorithms discussed in this paper proved to be successful in delivering sequences of pass directions of minimal length for several practical example grammars.

Key words. attribute grammars, multi-pass evaluators, minimization algorithms

1. Introduction. Attribute grammars [6] are used to describe the semantics of programming languages.

For the generation of compilers from such a semantic description several tree traversal strategies have been developed to evaluate the semantic attributes within the derivation tree of a program.

In pass-oriented evaluation strategies, as suggested by Bochmann [2] and Jazayeri and Walter [5], a bounded number of depth-first left-to-right and/or right-to-left traversals of the derivation tree are made.

Both Bochmann and Jazayeri and Walter in their papers [2], [5] made the more or less implicit assumption that different instances of the same attribute in any derivation tree should be evaluated during the same pass with for each derivation tree the same pass number. In [1] a multi-pass attribute grammar satisfying this restriction is called *simple*, whereas the general (unrestricted) multi-pass attribute grammars are called *pure*.

An interesting problem is the determination of the minimal m for which a sequence of m pass directions (left-to-right and/or right-to-left) can be found, such that an associated distribution of the attributes over the passes meets the simple multi-pass requirements. In [10], [11] this problem was pointed out to be NP-complete. Also in [10], [11] a polynomial time algorithm was developed that delivers a sequence of pass directions of minimal length for a subclass of simple multi-pass grammars of practical importance.

In this paper an alternative polynomial time algorithm is discussed. It starts from a given sequence of pass directions and tries to find a shorter one. This process is repeated until it is found that further improvements are impossible.

This paper is organized as follows: Section 2 provides an introduction to the basic concepts associated with attribute grammars. In § 3 the principles of simple multi-pass

* Received by the editors September 13, 1983, and in revised form May 17, 1984.

† Department of Informatics, Twente University of Technology, P.O. Box 217, 7500 AE Enschede, the Netherlands.

evaluation are summarized. In § 4, for a given sequence of pass directions, two sequences of pass functions are defined. A sequence of functions ranging from the function with minimal pass numbers to the function with maximal pass numbers and a sequence of functions in reversed order. Algorithms are discussed that compute the next function in a sequence from the preceding one. In § 5 it is pointed out that during the successive computations of pass functions within a sequence, the sequence of pass directions can possibly be shortened by crossing out passes that become empty. Finally an algorithm is discussed where, during the computation of a pass function from the preceding one in a sequence, pass directions are reconsidered and if necessary changed in order to find empty passes. In § 6 we discuss the effectiveness of the minimization algorithms presented in § 5.

2. Basic concepts. An *attribute grammar* (AG) is based on a context-free grammar G , which is augmented with attributes and attribute evaluation rules.

The underlying grammar G is a 4-tuple (V_N, V_T, P, S) , where V_N and V_T denote the finite sets of nonterminal and terminal symbols respectively, P is the set of productions and S is the start symbol. We write V for $V_N \cup V_T$.

We assume that the grammar G is *reduced* in the sense that each nonterminal symbol is accessible from the start symbol and can generate a string which contains no nonterminal symbols.

A production $p \in P$ is denoted as $p: X_{p0} \rightarrow X_{p1}X_{p2} \cdots X_{pn_p}$, where $n_p \geq 0$, $X_{p0} \in V_N$ and $X_{pk} \in V$ for $1 \leq k \leq n_p$.

Each symbol $X \in V$ has a set $A(X)$ of attributes which can be partitioned into two disjoint subsets $I(X)$ and $S(X)$ of *inherited* and *synthesized* attributes respectively. For $X = S$ and $X \in V_T$ we require $I(X) = \emptyset$.

The set of all attributes will be denoted by A , i.e., $A = \bigcup_{x \in V} A(X)$. Attributes of different symbols are different. An attribute a of symbol X is also denoted $a(X)$.

Production p is said to have the *attribute occurrence* (a, p, k) if $a \in A(X_{pk})$. The set of attribute occurrences of production p can be partitioned into two disjoint subsets of defined occurrences and used occurrences denoted by $DO(p)$ and $UO(p)$ respectively.

These subsets are defined as follows:

$$DO(p) = \{(s, p, 0) | s \in S(X_{p0})\} \cup \{(i, p, k) | i \in I(X_{pk}) \wedge 1 \leq k \leq n_p\},$$

$$UO(p) = \{(i, p, 0) | i \in I(X_{p0})\} \cup \{(s, p, k) | s \in S(X_{pk}) \wedge 1 \leq k \leq n_p\}.$$

Associated with each production p is a set of attribute evaluation rules which specify how to compute the values of the attribute occurrences in $DO(p)$. The evaluation rule defining attribute occurrence (a, p, k) has the form

$$(a, p, k) := f((a_1, p, k_1), (a_2, p, k_2), \dots, (a_m, p, k_m))$$

where $(a, p, k) \in DO(p)$, f is a total function and $(a_j, p, k_j) \in UO(p)$ for $1 \leq j \leq m$. We say that (a, p, k) *depends on* (a_j, p, k_j) for $1 \leq j \leq m$.

For each sentence of G a derivation tree exists. The nodes of the tree are labeled with symbols from V . For each interior node there is a production $X_{p0} \rightarrow X_{p1}X_{p2} \cdots X_{pn_p}$, such that the node is labeled with X_{p0} and its n_p sons are labeled with $X_{p1}, X_{p2}, \dots, X_{pn_p}$, respectively. We say that production p applies at that node.

Given a derivation tree, instances of attributes are attached to the nodes in the following way: if node N is labeled with grammar symbol X , then for each attribute $a \in A(X)$ an instance of a is attached to node N . We say that the derivation tree has *attribute instance* (a, N) .

Let N_0 be a node, p the production applied at N_0 and N_1, N_2, \dots, N_{n_p} the sons of N_0 from left-to-right respectively. An attribute evaluation instruction

$$(a, N_k) := f((a_1, N_{k_1}), (a_2, N_{k_2}), \dots, (a_m, N_{k_m}))$$

is associated with attribute instance (a, N_k) if the attribute evaluation rule

$$(a, p, k) := f((a_1, p, k_1), (a_2, p, k_2), \dots, (a_m, p, k_m))$$

is associated with production p .

The task of an attribute evaluator is to compute the values of all attribute instances attached to the derivation tree, by executing the attribute evaluation instructions associated with these attribute instances. In general the order of evaluation is free, with the only restriction that an attribute evaluation instruction cannot be executed before the values of its arguments are defined. Initially the values of all attribute instances attached to the derivation tree are undefined, with the exception of the (synthesized) attribute instances associated with terminal symbols. The latter are determined by the parser.

In this paper the visiting order of the nodes of the derivation tree is *pass-oriented*, i.e., a bounded number of depth-first left-to-right and/or right-to-left traversals of the tree are made during which the instances of the attributes are evaluated.

3. Simple multi-pass evaluation. From [1] we repeat some terminology, definitions and theorems concerning simple multi-pass evaluation, i.e., attribute evaluation in successive passes, where all different instances of the same attribute in any derivation tree are evaluated during the same pass with the same pass number for each derivation tree.

To describe the ordering of left-to-right and right-to-left passes and the distribution of the attributes over the passes we use the following notation.

The directions of the successive passes are indicated by a sequence $\langle d_1, \dots, d_m \rangle$ where d_i ($1 \leq i \leq m$) denotes the direction of the i th pass, which is either L (left-to-right) or R (right-to-left).

A *partial partition* of the set A of attributes into a sequence of mutually disjoint subsets will be denoted by $\langle A_1, \dots, A_m \rangle$. Such a partition is *complete* if $\cup_{i=1}^m A_i = A$. In this paper a partition may include empty subsets.

A partial partition $\langle A_1, \dots, A_m \rangle$ of the set A of attributes is *correct* with respect to a given sequence $\langle d_1, \dots, d_m \rangle$ of pass directions if the instances of all attributes in set A_i ($1 \leq i \leq m$) can be evaluated during the i th pass of a simple multi-pass evaluator.

An attribute grammar is *simple m -pass* if with respect to some sequence $\langle d_1, \dots, d_m \rangle$ of pass directions a correct complete partition of the set A of attributes exists.

An attribute grammar is *simple multi-pass* if it is simple m -pass for some m (this is called simple BD multi-pass in [1], where BD stands for Both Directions).

With each complete partition $\langle A_1, \dots, A_m \rangle$ of the set A of attributes of an attribute grammar a *pass function* $\text{pass}: A \rightarrow \{1, \dots, m\}$ can be associated, as follows: $\text{pass}(a) = i$ if $a \in A_i$. The pass function is *correct* if the partition is correct.

The principle of simple multi-pass evaluation is that in each context the same pass number is associated with different instances of the same attribute. This leads to *precedence relations* [1] among attributes.

The relation $a \text{ prec } b$ between attributes a and b holds if a production $X_{p_0} \rightarrow X_{p_1} X_{p_2} \dots X_{p_{n_p}}$ exists with attribute occurrences (a, p, j) and (b, p, k) such that (b, p, k) depends on (a, p, j) .

The relation $a L b$ between attributes a and b holds if $a \text{ prec } b$ and for each production $X_{p0} \rightarrow X_{p1} X_{p2} \cdots X_{pn_p}$ with attribute occurrences (a, p, j) and (b, p, k) such that (b, p, k) depends on (a, p, j) the following condition is satisfied: if $1 \leq k \leq n_p$ then $j < k$.

The relation $a \bar{L} b$ between attributes a and b holds if $a \text{ prec } b$ but not $a L b$.

Figure 1 pictures a production $X_{p0} \rightarrow X_{p1} \cdots X_{pk} \cdots X_{pj} \cdots X_{pn_p}$, where $k \leq j$. Attributes a and b are associated with grammar symbols X_{pj} and X_{pk} respectively. The arc from a to b indicates that inherited attribute occurrence (b, p, k) depends on synthesized attribute occurrence (a, p, j) . Such a dependency leads to the relation $a \bar{L} b$.

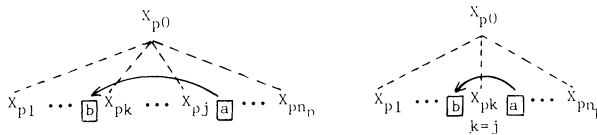


FIG. 1. Dependencies which cause the relation $a \bar{L} b$.

The relation $a R b$ between attributes a and b holds if $a \text{ prec } b$ and for each production $X_{p0} \rightarrow X_{p1} X_{p2} \cdots X_{pn_p}$ with attribute occurrences (a, p, j) and (b, p, k) such that (b, p, k) depends on (a, p, j) the following condition is satisfied: if $1 \leq k \leq n_p$ then $j = 0$ or $j > k$.

The relation $a \bar{R} b$ between attributes a and b holds if $a \text{ prec } b$ but not $a R b$.

Figure 2 pictures a production $X_{p0} \rightarrow X_{p1} \cdots X_{pj} \cdots X_{pk} \cdots X_{pn_p}$, where $k \geq j$. Attributes a and b are associated with grammar symbols X_{pj} and X_{pk} respectively. The arc from a to b indicates that inherited attribute occurrence (b, p, k) depends on synthesized attribute occurrence (a, p, j) . Such a dependency leads to the relation $a \bar{R} b$.

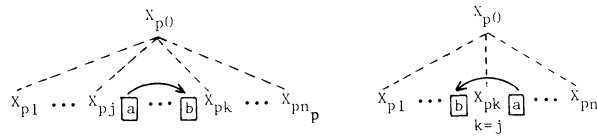


FIG. 2. Dependencies which cause the relation $a \bar{R} b$.

The following theorem characterizes the correct partitions in terms of the introduced precedence relations.

THEOREM 3.1 [1, Thm. 4.1]. *A complete partition $\langle A_1, \dots, A_m \rangle$ of the set A of attributes of an attribute grammar is correct with respect to a given sequence $\langle d_1, \dots, d_m \rangle$ of pass directions if and only if for the corresponding pass function and for all attributes a and b :*

- (i) if $a \text{ prec } b$ then $\text{pass}(a) \leq \text{pass}(b)$;
- (ii) if $a \bar{L} b$ and $\text{pass}(a) = \text{pass}(b)$ then this pass is a right-to-left pass;
- (iii) if $a \bar{R} b$ and $\text{pass}(a) = \text{pass}(b)$ then this pass is a left-to-right pass.

The precedence relations of an attribute grammar AG can be represented by a directed graph in the following way:

Each attribute of AG is represented by a vertex. Arc (a, b) is contained in the graph if the relation $a \text{ prec } b$ holds between attributes a and b . To each arc two labels are assigned in the following way: if the relation $a L b$ holds, then arc (a, b) has label L , otherwise \bar{L} ; if the relation $a R b$ holds, then arc (a, b) has label R , otherwise \bar{R} .

The graph associated with attribute grammar AG will be denoted by $P(AG)$ and will be called the *precedence graph* of AG. Notice that the vertices of the precedence graph are attributes and not attribute instances. Paths, cycles and their labeling in the precedence graph play an essential role in the theory of simple multi-pass evaluation.

From Theorem 3.1, it immediately follows that:

- (i) all instances of all attributes of a cycle have to be evaluated during the same pass;
- (ii) for a pass direction d (left-to-right or right-to-left): if an arc labeled \bar{d} is part of a cycle, then it is impossible to evaluate the instances of the attributes of that cycle during a d -pass;
- (iii) if both arcs labeled \bar{L} and arcs labeled \bar{R} are part of a cycle, then it is impossible to evaluate the instances of the attributes of that cycle during any pass.

In fact the labeling of the cycles conclusively decides whether an attribute grammar meets the simple multi-pass requirements.

THEOREM 3.2 [1]. *An attribute grammar is simple multi-pass if and only if its precedence graph has no cycles with both \bar{L} -arcs and \bar{R} -arcs.*

4. Pass directions and pass functions. Theorem 3.1 states that knowledge of the precedence relations between the attributes of an attribute grammar AG suffices to determine whether a given complete partition $\langle A_1, \dots, A_m \rangle$ of the set A of attributes of AG is correct with respect to a given sequence $\langle d_1, \dots, d_m \rangle$ of pass directions. In this section we will use this theorem to get more insight in the set of correct pass functions with respect to a given sequence of pass directions.

Given an attribute grammar AG with A the set of attributes of AG, let F be the set of correct pass functions with respect to a given sequence $\langle d_1, \dots, d_m \rangle$ of pass directions. On F we define a natural partial ordering as follows.

For all $f, g \in F: f \leq g$ if and only if for all $a \in A f(a) \leq g(a)$.

The question is whether in F there exist unique correct pass functions with minimal and maximal pass numbers respectively. To answer this question we define the functions $\min(f, g)$ and $\max(f, g)$ for f and $g \in F$, as follows:

$$\min(f, g)(a) = \text{MIN}(f(a), g(a)) \quad \text{for all } a \in A,$$

$$\max(f, g)(a) = \text{MAX}(f(a), g(a)) \quad \text{for all } a \in A,$$

where MIN and MAX are the usual minimum and maximum functions on natural numbers.

LEMMA 4.1. *Let F be the set of correct pass functions with respect to a given sequence $\langle d_1, \dots, d_m \rangle$ of pass directions of an attribute grammar AG. For all $f, g \in F$ the functions $\min(f, g)$ and $\max(f, g)$ are also in F .*

Proof. We prove the lemma for $\min(f, g)$. The proof for $\max(f, g)$ proceeds along the same lines.

We have to prove that $\min(f, g)$ fulfills the criteria (i)-(iii) of Theorem 3.1. Let a and b be attributes of AG.

(i) If $a \text{ prec } b$ then $f(a) \leq f(b)$ and $g(a) \leq g(b)$. Hence, $\text{MIN}(f(a), g(a)) \leq f(a) \leq f(b)$ and $\text{MIN}(f(a), g(a)) \leq g(a) \leq g(b)$. Hence, $\text{MIN}(f(a), g(a)) \leq \text{MIN}(f(b), g(b))$.

(ii) Let $a \bar{L} b$ and $\min(f, g)(a) = \min(f, g)(b)$. Assume that $\min(f, g)(b) = f(b)$ (the case that it is $g(b)$ is symmetric). If $\min(f, g)(b) = f(b)$ then $f(b) = \min(f, g)(b) = \min(f, g)(a) \leq f(a)$. Hence, $f(a) = f(b)$. From Theorem 3.1(ii) it follows that $d_{f(b)} = R$, and consequently $d_{\min(f, g)(b)} = R$.

(iii) Exchange L and R in (ii). \square

THEOREM 4.1. *Let F be the set of correct pass functions with respect to a given sequence $\langle d_1, \dots, d_m \rangle$ of pass directions of an attribute grammar AG. If F is not empty then in F there exist unique correct pass functions minpass and maxpass with minimal and maximal pass numbers respectively.*

Proof. The set A of attributes of AG is finite and, hence the set of correct pass functions $A \rightarrow \{1, \dots, m\}$ is finite. Let the set F of correct pass functions be $\{f_1, f_2, \dots, f_p\}$. From Lemma 4.1 it follows that the functions minpass = $\min(f_1, \min(f_2, \dots \min(f_{p-2}, \min(f_{p-1}, f_p)) \dots))$ and maxpass = $\max(f_1, \max(f_2, \dots \max(f_{p-2}, \max(f_{p-1}, f_p)) \dots))$ are correct pass functions. Clearly, minpass and maxpass are the functions with minimal and maximal pass numbers respectively. \square

Remark. Let F be the set of correct pass functions with respect to a given sequence $\langle d_1, \dots, d_m \rangle$ of pass directions of an attribute grammar AG. The relation \leq is a partial ordering on F . Every pair of elements $f, g \in F$ has both a least upper bound (lub) and a greatest lower bound (glb), namely:

$$\text{lub of } f \text{ and } g = \max(f, g),$$

$$\text{glb of } f \text{ and } g = \min(f, g).$$

Hence, F is a lattice under \leq .

Theorem 4.1 now immediately follows because every finite lattice has a unique minimal and maximal element (F is finite because A is finite).

Now, with respect to a given sequence of pass directions for an attribute grammar AG, we consider some specific pass functions and also discuss algorithms to compute them. The usually computed pass function [1], [2], [5], [10], [11] is the function minpass that associates with each attribute the minimal possible pass number. The opposite result is the function that delivers maximal possible pass numbers. All the other pass functions have their values in between these extremes.

In this paper we make, with respect to a given sequence $\langle d_1, \dots, d_m \rangle$ of pass directions, use of two special classes of functions minmaxpass $_k$ ($0 \leq k \leq m$) and maxminpass $_k$ ($1 \leq k \leq m + 1$), defined as follows:

$$\text{minmaxpass}_k(a) = \begin{cases} \text{minpass}(a) & \text{for } \text{minpass}(a) \leq k, \\ \text{maxpass}(a) & \text{for } \text{minpass}(a) > k, \end{cases}$$

$$\text{maxminpass}_k(a) = \begin{cases} \text{maxpass}(a) & \text{for } \text{maxpass}(a) \geq k, \\ \text{minpass}(a) & \text{for } \text{maxpass}(a) < k. \end{cases}$$

The function minmaxpass $_k$ partitions the set A of attributes into two disjoint subsets $A_{\leq k}$ and $A_{> k}$

$$\underbrace{\langle A_1, \dots, A_k \rangle}_{A_{\leq k}}, \underbrace{\langle A_{k+1}, \dots, A_m \rangle}_{A_{> k}}.$$

$A_{\leq k} = \{a \mid \text{minpass}(a) \leq k\}$, i.e., the set of all attributes that can be evaluated within the first k passes. $A_{> k} = \{a \mid \text{minpass}(a) > k\}$, i.e., the set of all attributes that cannot be evaluated within the first k passes.

$A_{\leq k}$ is partitioned into the sequence $\langle A_1, \dots, A_k \rangle$ of disjoint subsets such that $A_i = \{a \mid \text{minpass}(a) = i\}$ for $1 \leq i \leq k$, i.e., $\langle A_1, \dots, A_k \rangle$ is the partition such that the attributes of $A_{\leq k}$ are evaluated at the earliest possible pass, (i.e., the first k elements of the partition corresponding to minpass). $A_{> k}$ is partitioned into the sequence $\langle A_{k+1}, \dots, A_m \rangle$ of disjoint subsets such that $A_i = \{a \mid \text{minpass}(a) > k \text{ and } \text{maxpass}(a) =$

$i\}$ for $k + 1 \leq i \leq m$, i.e., $\langle A_{k+1}, \dots, A_m \rangle$ is the partition such that the attributes of $A_{>k}$ are evaluated at the latest possible pass.

Observe that $\text{minmaxpass}_0 = \text{maxpass}$ and $\text{minmaxpass}_{m-1} = \text{minmaxpass}_m = \text{minpass}$.

For the maxminpass functions analogous remarks hold, as follows. The function maxminpass_k partitions the set A of attributes into two disjoint subsets $A_{<k}$ and $A_{\geq k}$,

$$\underbrace{\langle A_1, \dots, A_{k-1} \rangle}_{A_{<k}} \quad \underbrace{\langle A_k, \dots, A_m \rangle}_{A_{>k}}$$

$A_{\geq k} = \{a \mid \text{maxpass}(a) \geq k\}$, i.e., the set of all attributes whose evaluation can be postponed until after the $(k - 1)$ th pass. $A_{<k} = \{a \mid \text{maxpass}(a) < k\}$, i.e., the set of all attributes whose evaluation cannot be postponed until after the $(k - 1)$ th pass.

$A_{\geq k}$ is partitioned into the sequence $\langle A_k, \dots, A_m \rangle$ of disjoint subsets such that $A_i = \{a \mid \text{maxpass}(a) = i\}$ for $k \leq i \leq m$, i.e., $\langle A_k, \dots, A_m \rangle$ is the partition such that the attributes of $A_{\geq k}$ are evaluated at the latest possible pass (i.e., the last $m - k + 1$ elements of the partition corresponding to maxpass). $A_{<k}$ is partitioned into the sequence $\langle A_1, \dots, A_{k-1} \rangle$ of disjoint subsets such that $A_i = \{a \mid \text{maxpass}(a) < k \text{ and } \text{minpass}(a) = i\}$ for $1 \leq i \leq k - 1$, i.e., $\langle A_1, \dots, A_{k-1} \rangle$ is the partition such that the attributes of $A_{<k}$ are evaluated at the earliest possible pass.

Observe that $\text{maxminpass}_1 = \text{maxminpass}_2 = \text{maxpass}$ and $\text{maxminpass}_{m+1} = \text{minpass}$.

Now we prove that the functions minmaxpass_k ($0 \leq k \leq m$) and maxminpass_k ($1 \leq k \leq m + 1$) are correct pass functions.

THEOREM 4.2. *Let AG be an attribute grammar and $\langle d_1, \dots, d_m \rangle$ a sequence of pass directions such that, with respect to $\langle d_1, \dots, d_m \rangle$ a nonempty set of correct pass functions for AG exists. Then the pass functions minmaxpass_k ($0 \leq k \leq m$) and maxminpass_k ($1 \leq k \leq m + 1$) are correct with respect to $\langle d_1, \dots, d_m \rangle$.*

Proof. We prove the theorem for the functions minmaxpass_k ($0 \leq k \leq m$). The proof for maxminpass_k ($1 \leq k \leq m + 1$) is analogous.

We have to show that with respect to $\langle d_1, \dots, d_m \rangle$ the function minmaxpass_k ($0 \leq k \leq m$) fulfills the criteria (i)-(iii) of Theorem 3.1.

Since the set of correct pass functions with respect to $\langle d_1, \dots, d_m \rangle$ is not empty, there exist correct pass functions minpass and maxpass with respect to $\langle d_1, \dots, d_m \rangle$.

Let a and b be attributes of AG. If $a \text{ prec } b$ then $\text{minpass}(a) \leq \text{minpass}(b)$. Hence, we consider 3 cases.

Case 1. a and $b \in A_{\leq k}$. From $\text{minmaxpass}_k(a) = \text{minpass}(a)$ and $\text{minmaxpass}_k(b) = \text{minpass}(b)$ and the fact that minpass is a correct pass function with respect to $\langle d_1, \dots, d_m \rangle$, it follows that criteria (i)-(iii) hold for the function minmaxpass_k on a and b .

Case 2. a and $b \in A_{>k}$. From $\text{minmaxpass}_k(a) = \text{maxpass}(a)$ and $\text{minmaxpass}_k(b) = \text{maxpass}(b)$ and the fact that maxpass is a correct pass function with respect to $\langle d_1, \dots, d_m \rangle$, it follows that criteria (i)-(iii) hold for the function minmaxpass_k on a and b .

Case 3. $a \in A_{\leq k}$ and $b \in A_{>k}$. From $\text{minmaxpass}_k(a) \leq k$ and $\text{minmaxpass}_k(b) > k$ it follows that $\text{minmaxpass}_k(a) < \text{minmaxpass}_k(b)$. Hence, criteria (i)-(iii) hold for the pass function minmaxpass_k on a and b . \square

In the following we will show a correspondence between the two sequences of pass functions $\langle \text{minmaxpass}_0, \text{minmaxpass}_1, \dots, \text{minmaxpass}_m \rangle$ and $\langle \text{maxminpass}_{m+1}, \text{maxminpass}_m, \dots, \text{maxminpass}_1 \rangle$. For the explanation of that point for an

attribute grammar AG we need, besides its precedence graph $P(AG)$, also its *reversed precedence graph* with arcs in the opposite direction. This reversed precedence graph will be denoted by $\tilde{P}(AG)$ and defined as follows.

For each vertex in $P(AG)$ there is a vertex in $\tilde{P}(AG)$. Arc (a, b) is contained in $\tilde{P}(AG)$ if arc (b, a) exists in $P(AG)$. Both arcs (a, b) in $\tilde{P}(AG)$ and (b, a) in $P(AG)$ have the same labels.

In § 3 precedence relations were defined between the attributes of an attribute grammar AG. These relations were presented by a directed graph, called the precedence graph of AG. More generally we can define a precedence graph as a directed graph where each arc has label L or \bar{L} and each arc has label R or \bar{R} . Notice that, according to this new definition, for each attribute grammar AG both $P(AG)$ and $\tilde{P}(AG)$ are precedence graphs.

Up to now we defined our precedence relations with respect to the attributes of an attribute grammar, but we can also define precedence relations with respect to a precedence graph as follows.

DEFINITION 4.1. With respect to a precedence graph the following precedence relations hold between the nodes a and b :

- (i) $a \text{ prec } b$ if arc (a, b) exists;
- (ii) $a L b$ if arc (a, b) exists and has label L ;
- (iii) $a \bar{L} b$ if arc (a, b) exists and has label \bar{L} ;
- (iv) $a R b$ if arc (a, b) exists and has label R ;
- (v) $a \bar{R} b$ if arc (a, b) exists and has label \bar{R} .

Up to now we considered correct partitions of the set A of attributes of an attribute grammar AG with respect to a given sequence $\langle d_1, \dots, d_m \rangle$ of pass directions. This is equivalent to the correct partitions of the set of nodes of $P(AG)$ with respect to sequence $\langle d_1, \dots, d_m \rangle$. In the following we will also consider correct partitions of the nodes of $\tilde{P}(AG)$ with respect to sequence $\langle d_m, \dots, d_1 \rangle$, i.e., the sequence of pass directions in reversed order.

For that reason, if necessary, we will talk of the partition of the set of nodes of a precedence graph instead of the set of attributes of an attribute grammar. Similarly we will also talk of the pass function for a precedence graph instead of the pass function for an attribute grammar.

For the correctness of a partition of the set of nodes of a precedence graph and its associated pass function with respect to a given sequence of pass directions, we use Theorem 3.1 as a definition.

DEFINITION 4.2. A complete partition $\langle A_1, \dots, A_m \rangle$ of the set A of nodes of a precedence graph is correct with respect to a given sequence $\langle d_1, \dots, d_m \rangle$ of pass directions if for the corresponding pass function and for each pair of nodes a and b holds:

- (i) if $a \text{ prec } b$ then $\text{pass}(a) \leq \text{pass}(b)$;
- (ii) if $a \bar{L} b$ and $\text{pass}(a) = \text{pass}(b)$ then this pass is a right-to-left pass;
- (iii) if $a \bar{R} b$ and $\text{pass}(a) = \text{pass}(b)$ then this pass is a left-to-right pass.

Now we consider the mapping h which associates with each pass function p a pass function $h(p) = p'$ as follows. If p is the pass function associated with partition $\langle A_1, \dots, A_m \rangle$, then p' is the pass function associated with $\langle A_m, \dots, A_1 \rangle$. Denoting the partition $\langle A_m, \dots, A_1 \rangle$ by $\langle A'_1, \dots, A'_m \rangle$ implies $A'_i = A_{m-i+1}$ for $1 \leq i \leq m$ and $p'(a) = m - p(a) + 1$ for all $a \in A$.

Let F be the set of correct pass functions with respect to $\langle d_1, \dots, d_m \rangle$ for graph $P(AG)$ and let F' be the set of correct pass functions with respect to $\langle d_m, \dots, d_1 \rangle$ for graph $\tilde{P}(AG)$. In Lemma 4.2 we will prove that if $p \in F$ is correct with respect to $\langle d_1, \dots, d_m \rangle$ for $P(AG)$ then $h(p) = p'$ is correct with respect to $\langle d_m, \dots, d_1 \rangle$ for

$\tilde{P}(AG)$, i.e., that h is a mapping from F into F' . In Lemma 4.3 we will prove that h is a bijection from F onto F' .

LEMMA 4.2. h is a mapping from F into F' .

Proof. Let p be a pass function which is correct with respect to $\langle d_1, \dots, d_m \rangle$ for $P(AG)$. Let $h(p) = p'$ be the pass function such that $p'(a) = m - p(a) + 1$ for all $a \in A$. We have to prove that p' is correct with respect to $\langle d'_1, \dots, d'_m \rangle$ for $\tilde{P}(AG)$, where $d'_i = d_{m-i+1}$.

We verify that p' fulfills the criteria (i)-(iii) of Definition 4.2. Let a and b be nodes of $P(AG)$ and $\tilde{P}(AG)$.

(i) If a prec b in $\tilde{P}(AG)$ then b prec a in $P(AG)$. From Definition 4.2(i) it follows that $p(b) \leq p(a)$. Hence $p'(a) \leq p'(b)$.

(ii) We will prove that, if $a \bar{L} b$ in $\tilde{P}(AG)$ and $p'(a) = p'(b)$ then $d'_{p'(b)} = R$. If $a \bar{L} b$ in $\tilde{P}(AG)$ then $b \bar{L} a$ in $P(AG)$. If $p'(b) = p'(a)$ then $p(b) = p(a)$. From $b \bar{L} a$ in $P(AG)$ and $p(b) = p(a)$ follows (Definition 4.2(ii)): $d_{p(b)} = R$ and hence $d'_{m-p(b)+1} = R$ and hence $d'_{p'(b)} = R$.

(iii) Exchange L and R in (ii). \square

Analogously to the definition of mapping h from F into F' we define the mapping h^{-1} from F' into F as follows. If p is a correct pass function with respect to $\langle d_1, \dots, d_m \rangle$ for $\tilde{P}(AG)$ then $p' = h^{-1}(p)$ is a correct pass function with respect to $\langle d'_1, \dots, d'_m \rangle$ for $P(AG)$, where $d'_i = d_{m-i+1}$ and such that $p'(a) = m - p(a) + 1$ for all $a \in A$. Clearly h^{-1} is the inverse of h .

LEMMA 4.3. h is a bijection from F onto F' .

Proof. Clearly $h^{-1}h$ is the identity mapping from F onto F and hh^{-1} is the identity mapping from F' onto F' . \square

In the following lemma we consider the way in which related pass functions in F and F' are ordered in both sets.

LEMMA 4.4. From f and $g \in F$ such that $f \leq g$ follows that $h(f) \geq h(g)$. From f and $g \in F'$ such that $f \leq g$ follows that $h^{-1}(f) \geq h^{-1}(g)$.

Proof. Follows immediately from the definitions of h and h^{-1} . \square

From Lemmas 4.3 and 4.4 we immediately conclude the following theorem.

THEOREM 4.3. h is an isomorphism of the lattices $\langle F, \leq \rangle$ and $\langle F', \geq \rangle$.

So lattice $\langle F', \geq \rangle$ is isomorphic to the dual of lattice $\langle F, \leq \rangle$. This is the formal way of expressing the duality between evaluating attributes as early as possible and evaluating attributes as late as possible (by the pass functions minpass and maxpass respectively) or combinations of these strategies.

When it is not clear from the context whether pass function p in F or p in F' is meant we denote them by $F-p$ and $F'-p$ respectively. Now, from Theorem 4.3 immediately follows the next corollary.

COROLLARY 4.1.

$$h(F - \text{minpass}) = F' - \text{maxpass},$$

$$h(F - \text{maxpass}) = F' - \text{minpass}.$$

Finally we prove relations between the minmaxpass and maxminpass functions in both sets F and F' .

THEOREM 4.4. (i) $h(F - \text{minmaxpass}_k) = F' - \text{maxminpass}_{m-k+1}$ ($0 \leq k \leq m$).

(ii) $h(F - \text{maxminpass}_k) = F' - \text{minmaxpass}_{m-k+1}$ ($1 \leq k \leq m+1$).

Proof. We prove (i). The proof of (ii) is analogous. By definition,

$$F - \text{minmaxpass}_k(a) = \begin{cases} F - \text{minpass}(a) & \text{for } F - \text{minpass}(a) \leq k, \\ F - \text{maxpass}(a) & \text{for } F - \text{minpass}(a) > k. \end{cases}$$

Hence, applying h to both sides,

$$h(F\text{-minmaxpass}_k)(a) = \begin{cases} h(F\text{-minpass})(a) & \text{for } h(F\text{-minpass})(a) \geq m - k + 1, \\ h(F\text{-maxpass})(a) & \text{for } h(F\text{-minpass})(a) < m - k + 1. \end{cases}$$

Notice that $p(a) \leq k$ implies $h(p)(a) \geq m - k + 1$ and $p(a) > k$ implies $h(p)(a) < m - k + 1$.

By Corollary 4.1 the right side may be rewritten as

$$\begin{aligned} &F' - \text{maxpass}(a) \quad \text{for } F' - \text{maxpass}(a) \geq m - k + 1, \\ &F' - \text{minpass}(a) \quad \text{for } F' - \text{maxpass}(a) < m - k + 1. \end{aligned}$$

Hence, $h(F\text{-minmaxpass}_k) = F' - \text{maxminpass}_{m-k+1}$. \square

In the following, by a pass function and a partition of the set of attributes of an attribute grammar AG with respect to a sequence of pass directions, we mean the pass function and the partition for graph $P(\text{AG})$ and not for graph $\tilde{P}(\text{AG})$, unless explicitly stated otherwise.

We now discuss how to compute the partition associated with the pass function minmaxpass_{k+1} from the partition associated with the pass function minmaxpass_k ($0 \leq k < m - 1$). Notice that $\text{minmaxpass}_m = \text{minmaxpass}_{m-1}$.

Let AG be an attribute grammar and $\langle d_1, \dots, d_k, d_{k+1}, d_{k+2}, \dots, d_m \rangle$ a sequence of pass directions for which a correct complete partition of the set A of attributes of AG exists and let $\langle A_1, \dots, A_k, A_{k+1}, A_{k+2}, \dots, A_m \rangle$ for $0 \leq k < m - 1$, be the partition associated with the pass function minmaxpass_k with respect to $\langle d_1, \dots, d_k, d_{k+1}, d_{k+2}, \dots, d_m \rangle$. We have to compute the partition $\langle A_1, \dots, A_k, A'_{k+1}, A'_{k+2}, \dots, A'_m \rangle$ associated with the pass function minmaxpass_{k+1} ($0 \leq k < m - 1$).

Clearly the members of A'_{k+1} are all those attributes that cannot be evaluated within the first k passes, but that can be evaluated at the $(k+1)$ th pass.

The computation of A'_{k+1} proceeds as follows: Initially it is assumed that, besides the attributes from A_{k+1} that certainly belong to A'_{k+1} also all the attributes from $\bigcup_{i=k+2}^m A_i$ belong to A'_{k+1} . Nonmembers of A'_{k+1} will be successively deleted. From Theorem 3.1 it follows that all attributes a have to be deleted from A'_{k+1} for which an attribute $b \in A'_{k+1}$ exists such that $\text{arc}(b, a)$ is labeled \bar{d}_{k+1} (i.e., if $d_{k+1} = L$ then \bar{L} else \bar{R}). Furthermore all those attributes have to be deleted that depend (indirectly) on such attributes a . The deletion process continues until no more deletions are possible.

Finally we compute the sets A'_{k+2}, \dots, A'_m by deleting from A_{k+2}, \dots, A_m all the attributes that made a forward move to A'_{k+1} .

Algorithm 4.1 is a slightly extended version of the algorithm given by Bochmann in [2], Jazayeri and Walter in [5] and Alblas in [1]. Besides the set A'_{k+1} it also computes the sets A'_{k+2}, \dots, A'_m . The distribution of the attributes over these sets plays an essential role in our algorithm in the next section that minimizes the number of passes.

ALGORITHM 4.1. Computation of the partition associated with the pass function minmaxpass_{k+1} from the partition associated with minmaxpass_k with respect to a sequence $\langle d_1, \dots, d_k, d_{k+1}, d_{k+2}, \dots, d_m \rangle$ of pass directions.

Input: $P(\text{AG})$; k ($0 \leq k < m - 1$); pass direction d_{k+1} ; partition $\langle A_1, \dots, A_k, A_{k+1}, A_{k+2}, \dots, A_m \rangle$ associated with pass function minmaxpass_k .

Output: partition $\langle A_1, \dots, A_k, A'_{k+1}, A'_{k+2}, \dots, A'_m \rangle$ associated with pass function minmaxpass_{k+1} .

Algorithm:

```

begin
   $D := \bigcup_{i=k+2}^m A_i$ ;
  repeat
    delete a vertex  $a$  from  $D$ 
    if in  $P(AG)$  there exists a vertex  $b$  and an arc  $(b, a)$  such that
      ( $b \in A_{k+1} \cup D$  and arc  $(b, a)$  has label  $\bar{d}_{k+1}$ )
      or
      ( $b \notin \bigcup_{i=1}^{k+1} A_i \cup D$ )
    until no more vertices can be deleted from  $D$ ;
     $A'_{k+1} := A_{k+1} \cup D$ ;
    for  $i$  from  $k+2$  to  $m$ 
      do  $A'_i := A_i - D$  od
  end
  
```

Let n be the number of attributes of an attribute grammar AG . If we count the number of times the label of an arc of graph $P(AG)$ is examined, then Algorithm 4.1 takes time $O(n^3)$ in the worst case and $O(n^2)$ in the best case.

In [1] for the special cases where only left-to-right passes or only right-to-left passes are executed, for paths and attributes in graph $P(AG)$ d -cost functions ($d = L$ or R) are defined as follows.

DEFINITION 4.3. The d -cost of a path is the number of arcs labeled \bar{d} on the path.

DEFINITION 4.4. For each pair of attributes a and b

$$d\text{-cost}(a, b) = \begin{cases} \text{the maximal } d\text{-cost over all paths from } a \text{ to } b & \text{if a path from } a \text{ to } b \text{ exists,} \\ -\infty & \text{if no path from } a \text{ to } b \text{ exists.} \end{cases}$$

Using these cost functions we can formulate the following theorem [1, Lemma 6.1].

THEOREM 4.5. Let "pass" be a correct pass function with respect to the sequence $\langle d, d, \dots \rangle$ of pass directions, where $d = L$ or $d = R$, for an attribute grammar AG . If a path exists in $P(AG)$ from attribute a to attribute b , then $\text{pass}(b) \cong \text{pass}(a) + d\text{-cost}(a, b)$.

The d -cost functions can be used for the deletion of nonmembers from A'_{k+1} in Algorithm 4.1. All attributes $a \in A'_{k+1}$ for which an attribute $b \in A'_{k+1}$ exists such that $d_{k+1}\text{-cost}(b, a) \geq 1$ are precisely the attributes $c \in A'_{k+1}$ for which an attribute $b \in A'_{k+1}$ exists such that arc (b, c) is labeled \bar{d}_{k+1} and the attributes that depend indirectly on such attributes c .

Using the d -cost functions the repeat statement of Algorithm 4.1 can be rewritten as follows:

```

for all vertices  $a \in D$ 
  delete  $a$  from  $D$ 
  if in  $P(AG)$  there exists a vertex  $b$  such that
     $b \in A_{k+1} \cup D$  and  $d_{k+1}\text{-cost}(b, a) > 1$ .
  
```

In [1] it is pointed out that the computation of the d -cost functions takes time $O(n^3)$, where n is the number of attributes of the grammar. If we count the number of times $d\text{-cost}(b, a)$ is examined for any b, a , then the revised version of Algorithm 4.1 takes time $O(n^2)$. Hence, the revised version is more efficient if a sequence of pass functions minmaxpass_k (for k from 1 to $m-1$) has to be computed.

We end this section with the computation of the partition associated with the pass function maxinpass_{k-1} from the partition associated with the pass function maxinpass_k ($2 < k \leq m + 1$). Notice that $\text{maxinpass}_1 = \text{maxinpass}_2$.

Theorem 4.4 states that for the computation of pass function maxinpass_k ($2 \leq k \leq m$) with respect to sequence $\langle d_1, \dots, d_m \rangle$ of pass directions for graph $P(\text{AG})$, the following 3 actions deliver the desired result.

1. put the sequence of pass directions into the reversed order;
2. compute the partition associated with the pass function $\text{minmaxpass}_{m-k+1}$ ($2 \leq k \leq m$) with respect to the reversed sequence of pass directions for graph $P(\text{AG})$;
3. put the resulting partition into the reversed order.

Hence the pass function maxinpass_k ($2 \leq k \leq m$) can easily be computed using the algorithm for $\text{minmaxpass}_{m-k+1}$ ($2 \leq k \leq m$).

ALGORITHM 4.2. Computation of the partition associated with the pass function maxinpass_{k-1} from the partition associated with maxinpass_k with respect to a sequence $\langle d_1, \dots, d_{k-1}, d_k, \dots, d_m \rangle$ of pass directions.

Input: $P(\text{AG})$; k ($2 < k \leq m + 1$); pass direction d_{k-1} ; partition $\langle A_1, \dots, A_{k-1}, A_k, \dots, A_m \rangle$ associated with pass function maxinpass_k .
Output: partition $\langle A'_1, \dots, A'_{k-1}, A_k, \dots, A_m \rangle$ associated with pass function maxinpass_{k-1} .

Algorithm:

begin

construct $\tilde{P}(\text{AG})$ from $P(\text{AG})$;

apply Algorithm 4.1 to compute, with respect to sequence $\langle d_m, \dots, d_k, d_{k-1}, \dots, d_1 \rangle$ of pass directions, the partition associated with $\text{minmaxpass}_{m-k+2}$ from the partition associated with $\text{minmaxpass}_{m-k+1}$, with the following input and output:

Input: $\tilde{P}(\text{AG})$; $m - k + 1$; pass direction d_{k-1} ; partition $\langle A_m, \dots, A_k, A_{k-1}, \dots, A_1 \rangle$ associated with pass function $\text{minmaxpass}_{m-k+1}$ ($2 < k \leq m + 1$),

Output: partition $\langle A_m, \dots, A_k, A'_{k-1}, \dots, A'_1 \rangle$ associated with pass function $\text{minmaxpass}_{m-k+2}$ ($2 < k \leq m + 1$);

the result is partition $\langle A'_1, \dots, A'_{k-1}, A_k, \dots, A_m \rangle$ associated with the pass function maxinpass_{k-1} ($2 < k \leq m + 1$).

end

The following example illustrates the minmaxpass and maxinpass functions for a small and simple attribute grammar. The reader is invited to apply Algorithms 4.1 and 4.2 to compute the various pass functions.

Example 4.1. Consider attribute grammar AG1 with $V_N = \{Z, A, B\}$, $V_T = \{t\}$ and $P = \{Z \rightarrow AB, A \rightarrow t, B \rightarrow t\}$.

The only possible sentence is tt . Figure 3 shows the attributed derivation tree of

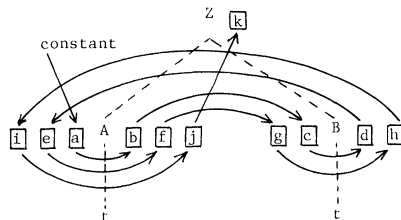


FIG. 3. Attributed derivation tree of grammar AG1.

tt with attribute instances and their dependencies. The associated precedence graph is given in Fig. 4.

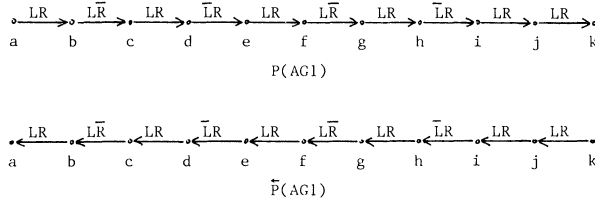


FIG. 4. $P(AG1)$ and $\bar{P}(AG1)$.

For sequence $\langle L, R, L, R \rangle$ of pass directions the partition associated with pass function minpass is: $\langle \{a, b, c, d\}, \{e, f\}, \{g, h\}, \{i, j, k\} \rangle$. (For an algorithm to compute minpass see [1], [2] or [5].) The partitions associated with the various maxminpass functions are

- maxminpass₅ = $\langle \{a, b, c, d\}, \{e, f\}, \{g, h\}, \{i, j, k\} \rangle = \text{minpass}$
- maxminpass₄ = $\langle \{a, b, c, d\}, \{e, f\}, \{ \}, \{g, h, i, j, k\} \rangle$
- maxminpass₃ = $\langle \{a, b, c, d\}, \{ \}, \{e, f\}, \{g, h, i, j, k\} \rangle$
- maxminpass₂ = $\langle \{a, b\}, \{c, d\}, \{e, f\}, \{g, h, i, j, k\} \rangle = \text{maxpass}$
- maxminpass₁ = $\langle \{a, b\}, \{c, d\}, \{e, f\}, \{g, h, i, j, k\} \rangle = \text{maxpass}$

Now, from the partition associated with pass function maxpass the various minmaxpass functions are computed.

- minmaxpass₀ = $\langle \{a, b\}, \{c, d\}, \{e, f\}, \{g, h, i, j, k\} \rangle = \text{maxpass}$
- minmaxpass₁ = $\langle \{a, b, c, d\}, \{ \}, \{e, f\}, \{g, h, i, j, k\} \rangle$
- minmaxpass₂ = $\langle \{a, b, c, d\}, \{e, f\}, \{ \}, \{g, h, i, j, k\} \rangle$
- minmaxpass₃ = $\langle \{a, b, c, d\}, \{e, f\}, \{g, h\}, \{i, j, k\} \rangle = \text{minpass}$
- minmaxpass₄ = $\langle \{a, b, c, d\}, \{e, f\}, \{g, h\}, \{i, j, k\} \rangle = \text{minpass}$

Observe that with respect to sequence $\langle L, R, L, R \rangle$ of pass directions partitions $\langle A_1, A_2, A_3, A_4 \rangle$ exist for which A_2 or A_3 is empty. In the following section we will use this information to cross out passes.

5. Minimizing the number of passes. Given a sequence $\langle d_1, \dots, d_m \rangle$ of pass directions for an attribute grammar AG, such that with respect to $\langle d_1, \dots, d_m \rangle$ a correct pass function for AG exists, the question arises whether a subsequence $\langle d_{i_1}, d_{i_2}, \dots, d_{i_k} \rangle$ ($1 \leq i_1 < i_2 < \dots < i_k \leq m$) of $\langle d_1, \dots, d_m \rangle$ exists such that with respect to this subsequence a correct pass function for AG exists.

To answer this question we first consider the partition $\langle A_1, \dots, A_k, A_{k+1}, \dots, A_m \rangle$ associated with the pass function minmaxpass_k with respect to $\langle d_1, \dots, d_k, d_{k+1}, \dots, d_m \rangle$. For all $a \in A_{k+1}$ holds: minpass(a) $\geq k+1$ and maxpass(a) = $k+1$. Hence, the elements of A_{k+1} have pass number $k+1$ for all correct pass functions with respect to $\langle d_1, \dots, d_k, d_{k+1}, \dots, d_m \rangle$.

Similarly, for partition $\langle A_1, \dots, A_{k-1}, A_k, \dots, A_m \rangle$ associated with the pass function maxminpass_k with respect to $\langle d_1, \dots, d_{k-1}, d_k, \dots, d_m \rangle$, the elements of A_{k-1} have pass number $k-1$ for all correct pass functions with respect to $\langle d_1, \dots, d_{k-1}, d_k, \dots, d_m \rangle$.

From these two observations we immediately conclude the following theorem.

THEOREM 5.1. *Given a sequence $\langle d_1, \dots, d_i, \dots, d_m \rangle$ of pass directions for an attribute grammar AG, such that with respect to this sequence a correct pass function for AG exists.*

With respect to this sequence of pass directions the following statements are equivalent:

A correct pass function exists for which the i th pass is empty.

For pass function minmaxpass_{i-1} ($1 \leq i \leq m$) the i th pass is empty.

For pass function maxminpass_{i+1} ($1 \leq i \leq m$) the i th pass is empty.

Example 5.1. Consider the precedence graph $P(\text{AG1})$ in Fig. 4. From the various minmaxpass and maxminpass functions for sequence $\langle L, R, L, R \rangle$ of pass directions (see Example 4.1) it follows that the second and the third pass are the only candidates to be crossed out. Notice that the subsequences $\langle L, L, R \rangle$ and $\langle L, R, R \rangle$ are the shortest possible subsequences for which a correct pass function exists and that it is not possible to cross out both the second and the third pass.

Now we consider two algorithms to cross out passes systematically from a given sequence $\langle d_1^{\text{old}}, \dots, d_m^{\text{old}} \rangle$ of pass directions, where $m = m_s$, the initial number of passes.

Algorithm 5.1 starts with the partition $\langle A_1^{\text{old}}, \dots, A_m^{\text{old}} \rangle$ associated with pass function $\text{minmaxpass}_0 = \text{maxpass}$. Now for k from 0 up to $m - 2$ action 1 or 2 is taken.

1. If in the partition $\langle A_1^{\text{new}}, \dots, A_k^{\text{new}}, A_{k+1}', A_{k+2}', \dots, A_m' \rangle$ associated with minmaxpass_k with respect to sequence $\langle d_1^{\text{new}}, \dots, d_k^{\text{new}}, d_{k+1}', d_{k+2}', \dots, d_m' \rangle$ of pass directions the set A_{k+1}' is empty, then pass direction d_{k+1}' is crossed out from the sequence of pass directions and the set A_{k+1}' from the partition associated with minmaxpass_k . The remaining pass directions and sets of the partition are renumbered such that again a consecutive sequence of pass numbers results. Observe that after the deletion of the $(k + 1)$ th pass and the renumbering of the remaining passes the new partition has the minmaxpass_k property with respect to the new sequence of pass directions.

2. If the $(k + 1)$ th pass cannot be crossed out, then Algorithm 4.1 is applied to compute the partition $\langle A_1^{\text{new}}, \dots, A_k^{\text{new}}, A_{k+1}^{\text{new}}, A_{k+2}'', \dots, A_m'' \rangle$ associated with minmaxpass_{k+1} from the partition $\langle A_1^{\text{new}}, \dots, A_k^{\text{new}}, A_{k+1}', A_{k+2}', \dots, A_m' \rangle$ associated with minmaxpass_k , both with respect to the same sequence

$$\begin{aligned} &\langle d_1^{\text{new}}, \dots, d_k^{\text{new}}, d_{k+1}', d_{k+2}', \dots, d_m' \rangle \\ &= \langle d_1^{\text{new}}, \dots, d_k^{\text{new}}, d_{k+1}^{\text{new}}, d_{k+2}', \dots, d_m' \rangle \end{aligned}$$

of pass directions.

The repetition ends with the sequence $\langle d_1^{\text{new}}, \dots, d_m^{\text{new}} \rangle$ of pass directions and the partition $\langle A_1^{\text{new}}, \dots, A_m^{\text{new}} \rangle$ associated with the pass function $\text{minmaxpass}_m = \text{minpass}$, where $m = m_f$ the final number of passes. If passes are crossed out then the final value m_f of m is less than its initial value m_s .

ALGORITHM 5.1. Computation of a minimal subsequence $\langle d_1^{\text{new}}, \dots, d_{m_f}^{\text{new}} \rangle$ of pass directions and its associated pass function minpass from the sequence $\langle d_1^{\text{old}}, \dots, d_{m_s}^{\text{old}} \rangle$ of pass directions and its associated pass function maxpass .

Input: $P(\text{AG})$; sequence $\langle d_1^{\text{old}}, \dots, d_{m_s}^{\text{old}} \rangle$ of pass directions; partition $\langle A_1^{\text{old}}, \dots, A_{m_s}^{\text{old}} \rangle$ associated with pass function maxpass .

Output: sequence $\langle d_1^{\text{new}}, \dots, d_{m_f}^{\text{new}} \rangle$ of pass directions; partition $\langle A_1^{\text{new}}, \dots, A_{m_f}^{\text{new}} \rangle$ associated with pass function minpass .

Algorithm:

begin

$m := m_s; k := 0;$

$\langle A_1', \dots, A_m' \rangle := \langle A_1^{\text{old}}, \dots, A_m^{\text{old}} \rangle;$

{ maxpass partition with respect to sequence $\langle d_1^{\text{old}}, \dots, d_m^{\text{old}} \rangle$ of pass directions}

$\langle d_1', \dots, d_m' \rangle := \langle d_1^{\text{old}}, \dots, d_m^{\text{old}} \rangle;$

while $k \leq m - 2$


```

do
  consider partition  $\langle A_1^{\text{new}}, \dots, A_k^{\text{new}}, A'_{k+1}, A'_{k+2}, \dots, A'_m \rangle$  associated with
  minmaxpassk with respect to sequence
   $\langle d_1^{\text{new}}, \dots, d_k^{\text{new}}, d'_{k+1}, d'_{k+2}, \dots, d'_m \rangle$  of pass directions;
  if  $A_{k+1} = \emptyset$ 
  then {cross out and renumber}
     $\langle A'_{k+1}, \dots, A'_{m-1} \rangle := \langle A'_{k+2}, \dots, A'_m \rangle;$ 
     $\langle d'_{k+1}, \dots, d'_{m-1} \rangle := \langle d'_{k+2}, \dots, d'_m \rangle;$ 
     $m := m - 1$ 
  else apply Algorithm 4.1 to compute the partition associated with minmax-
  passk+1 from the partition associated with minmaxpassk, with the
  following input and output:
    Input:  $P(\text{AG})$ ;  $k$ ; pass direction  $d'_{k+1}$ ; partition
     $\langle A_1^{\text{new}}, \dots, A_k^{\text{new}}, A'_{k+1}, A'_{k+2}, \dots, A'_m \rangle$  associated with min-
    maxpassk.
    Output: partition  $\langle A_1^{\text{new}}, \dots, A_k^{\text{new}}, A_{k+1}^{\text{new}}, A''_{k+2}, \dots, A''_m \rangle$  associated
    with minmaxpassk+1;
     $\langle A'_{k+2}, \dots, A'_m \rangle := \langle A''_{k+2}, \dots, A''_m \rangle$ ;  $d_{k+1}^{\text{new}} := d'_{k+1}$ ;  $k := k + 1$ 
  fi
od;
mf := m
end
    
```

If we count the number of times the d -cost function between attributes is examined, then the else-part of Algorithm 5.1 takes time $O(n^2)$, where n is the number of attributes of the grammar. Hence, Algorithm 5.1 takes time $O(m_f \cdot n^2)$.

Example 5.2. Consider the application of Algorithm 5.1 to graph $P(\text{AG1})$ in Fig. 4 (see Example 4.1).

The input is $P(\text{AG1})$ sequence $\langle L, R, L, R \rangle$ of pass directions and partition $\langle \{a, b\}, \{c, d\}, \{e, f\}, \{g, h, i, j, k\} \rangle$ associated with maxpass.

The second partition considered by Algorithm 5.1 is $\langle \{a, b, c, d\}, \{ \}, \{e, f\}, \{g, h, i, j, k\} \rangle$. This partition, associated with minmaxpass₁, includes an empty subset on the second position. Hence, the second pass is crossed out. The remaining partition is $\langle \{a, b, c, d\}, \{e, f\}, \{g, h, i, j, k\} \rangle$ associated with minmaxpass₁, which is correct with respect to the sequence $\langle L, L, R \rangle$ of pass directions. The new value of m is 3.

Algorithm 5.1 finally delivers partition $\langle \{a, b, c, d\}, \{e, f, g, h\}, \{i, j, k\} \rangle$ associated with minmaxpass₂ = minmaxpass₃ = minpass with respect to sequence $\langle L, L, R \rangle$ of pass directions.

Remark. Algorithm 5.1 can easily be changed such that it finds all correct subsequences: in Case 1 (when A'_{k+1} is empty) consider both actions 1 and 2. A recursive algorithm can be written to get all minimal (with respect to the “subsequence” criterion) subsequences and so in particular all minimal length subsequences.

Now, we consider the second algorithm to cross out passes systematically from a given sequence of pass directions. Algorithm 5.2 starts with the partition $\langle A_1^{\text{old}}, \dots, A_{m_s}^{\text{old}} \rangle$ associated with pass function minpass with respect to sequence $\langle d_1^{\text{old}}, \dots, d_{m_s}^{\text{old}} \rangle$ of pass directions and ends with the partition $\langle A_1^{\text{new}}, \dots, A_{m_f}^{\text{new}} \rangle$ associated with pass function maxpass with respect to sequence $\langle d_1^{\text{new}}, \dots, d_{m_f}^{\text{new}} \rangle$ of pass directions. As in Algorithm 5.1 for each partition associated with a maxminpass function an empty pass is crossed out and the remaining passes are renumbered in order to get a consecutively numbered sequence of passes.

ALGORITHM 5.2. Computation of a minimal subsequence $\langle d_1^{new}, \dots, d_{m_f}^{new} \rangle$ of pass directions and its associated pass function maxpass from the sequence $\langle d_1^{old}, \dots, d_{m_s}^{old} \rangle$ of pass directions and its associated pass function minpass.

Input: $P(AG)$; sequence $\langle d_1^{old}, \dots, d_{m_s}^{old} \rangle$ of pass directions; partition $\langle A_1^{old}, \dots, A_{m_s}^{old} \rangle$ associated with pass function minpass.

Output: sequence $\langle d_1^{new}, \dots, d_{m_f}^{new} \rangle$ of pass directions; partition $\langle A_1^{new}, \dots, A_{m_f}^{new} \rangle$ associated with pass function maxpass.

Algorithm:

begin

construct $\tilde{P}(AG)$ from $P(AG)$;

apply Algorithm 5.1 with the following input and output:

Input: $\tilde{P}(AG)$; sequence $\langle d_{m_s}^{old}, \dots, d_1^{old} \rangle$ of pass directions; partition $\langle A_{m_s}^{old}, \dots, A_1^{old} \rangle$ associated with pass function maxpass,

Output: sequence $\langle d_{m_f}^{new}, \dots, d_1^{new} \rangle$ of pass directions; partition $\langle A_{m_f}^{new}, \dots, A_1^{new} \rangle$ associated with pass function minpass;

the result is sequence $\langle d_1^{new}, \dots, d_{m_f}^{new} \rangle$ of pass directions and partition $\langle A_1^{new}, \dots, A_{m_f}^{new} \rangle$ associated with pass function maxpass

end

Example 5.3. Consider the application of Algorithm 5.2 to graph $P(AG1)$ in Fig. 4 (see Example 4.1).

The input is $P(AG1)$, sequence $\langle L, R, L, R \rangle$ of pass directions and partition $\langle \{a, b, c, d\}, \{e, f\}, \{g, h\}, \{i, j, k\} \rangle$.

The second partition considered by Algorithm 5.2 is: $\langle \{a, b, c, d\}, \{e, f\}, \{ \}, \{g, h, i, j, k\} \rangle$, the partition associated with maxminpass_4 . The third pass is crossed out. The remaining partition is: $\langle \{a, b, c, d\}, \{e, f\}, \{g, h, i, j, k\} \rangle$, the partition associated with maxminpass_3 , which is correct with respect to $\langle L, R, R \rangle$. The new value of m is 3.

Algorithm 5.2 finally delivers partition $\langle \{a, b\}, \{c, d, e, f\}, \{g, h, i, j, k\} \rangle$ associated with $\text{maxminpass}_2 = \text{maxminpass}_1 = \text{maxpass}$ with respect to sequence $\langle L, R, R \rangle$ of pass directions.

Both Algorithms 5.1 and 5.2 consider all the passes of the initial sequence of pass directions. Theorem 5.1 states that the remaining passes can not be crossed out. Hence, both algorithms deliver a sequence of pass directions such that with respect to this sequence a correct partition of the attributes can be found whereas for none of its subsequences a correct partition of the attributes exists, i.e., the resulting sequences are minimal with respect to the ‘‘subsequence ordering’’ (instead of length).

Notice that for grammar AG1 and sequence $\langle L, R, L, R \rangle$ of pass directions in Example 4.1, Algorithms 5.1 and 5.2 deliver different sequences of pass directions of the same length. In Example 5.4 we demonstrate that Algorithms 5.1 and 5.2 can give results of different lengths.

Example 5.4. Consider attribute grammar AG2 with $V_N = \{Z, A, B, C, D\}$, $V_T = \{t\}$ and $P = \{Z \rightarrow ABCD, A \rightarrow t, B \rightarrow t, C \rightarrow t, D \rightarrow t\}$.

The only possible sentence is $tttt$. Figure 5 shows the attributed derivation tree of

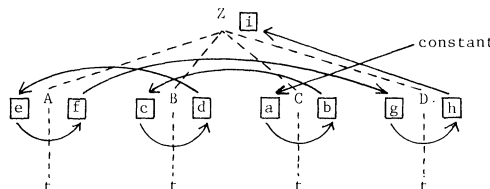


FIG. 5. Attributed derivation tree of grammar AG2.

tttt with attribute instances and their dependencies. The associated precedence graph is given in Fig. 6.

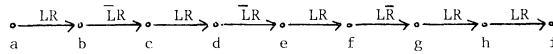


FIG. 6. $P(AG2)$.

For sequence $\langle L, L, R, L \rangle$ the partition associated with maxpass is: $\langle \{ \}, \{ \}, \{a, b, c, d\}, \{e, f, g, h, i\} \rangle$ and the partition associated with minpass is $\langle \{a, b\}, \{c, d\}, \{e, f\}, \{g, h, i\} \rangle$.

Observe that the result of Algorithm 5.1 is sequence $\langle R, L \rangle$ with the associated minpass partition $\langle \{a, b, c, d, e, f\}, \{g, h, i\} \rangle$ and that the result of Algorithm 5.2 is sequence $\langle L, L, L \rangle$ with the associated maxpass partition $\langle \{a, b\}, \{c, d\}, \{e, f, g, h, i\} \rangle$.

Example 5.4 illustrates that it depends on the order in which passes are considered and crossed out, which subsequence of the original sequence of pass directions is found and whether the length of this subsequence is equal to the minimal possible length. Notice that besides the orderings of Algorithms 5.1 and 5.2 other orderings are possible. Hence, in general it is not clear whether a subsequence of minimal length is found. Another drawback of Algorithms 5.1 and 5.2 is that the resulting sequence of pass directions is always a subsequence of the original sequence. For this reason we further try to shorten the number of evaluation passes by stepwise reconsidering and if necessary changing pass directions in order to find empty passes that can be crossed out.

Given a sequence $\langle d_1^{old}, \dots, d_{m_s}^{old} \rangle$ of pass directions and with respect to this sequence the correct complete partition $\langle A_1^{old}, \dots, A_{m_s}^{old} \rangle$ associated with the pass function maxpass, we compute a sequence $\langle d_1^{new}, \dots, d_{m_f}^{new} \rangle$ of pass directions, where $m_f \leq m_s$, and with respect to this sequence the correct complete partition $\langle A_1^{new}, \dots, A_{m_f}^{new} \rangle$ associated with the pass function minpass.

At the $(k+1)$ th step we start with the sequence $\langle d_1^{new}, \dots, d_k^{new}, d'_{k+1}, d'_{k+2}, \dots, d'_p \rangle$ and with respect to this sequence the correct complete partition $\langle A_1^{new}, \dots, A_k^{new}, A'_{k+1}, A'_{k+2}, \dots, A'_p \rangle$ associated with the pass function minmaxpass $_k$. Now, all the pass directions except the $(k+1)$ th one are fixed. If A'_{k+1} is empty, then the $(k+1)$ th pass is crossed out as in Algorithm 5.1. Otherwise, for d'_{k+1} the best of the original pass direction d'_{k+1} (denoted by opd) and the reversed pass direction (denoted by rpd) is chosen.

Clearly for $\langle d_1^{new}, \dots, d_k^{new}, opd, d'_{k+2}, \dots, d'_p \rangle$ a correct pass function exists. Now the question arises whether for the sequence $\langle d_1^{new}, \dots, d_k^{new}, rpd, d'_{k+2}, \dots, d'_p \rangle$ a correct pass function also exists. If for both sequences of pass directions correct pass functions exist, then from the definition of minmaxpass $_k$ it follows that both sequences have the same minmaxpass $_k$ partition. Hence, A'_{k+1} must be the same for both $d'_{k+1} = opd$ and $d'_{k+1} = rpd$. Hence (Theorem 3.1) for each pair of attributes $a, b \in A'_{k+1}$ such that an arc (a, b) exists, its label must be rpd.

From this observation follows our selection criterion for d'_{k+1} . If A'_{k+1} includes attributes a, b such that arc (a, b) is labeled rpd, then opd is selected for d'_{k+1} .

The application of this criterion is illustrated in the following example.

Example 5.5. Consider attribute grammar AG3 with $V_N = \{Z, A, B, C, D\}$, $V_T = \{t\}$ and $P = \{Z \rightarrow AB, B \rightarrow CD, A \rightarrow t, C \rightarrow t, D \rightarrow t\}$.

The only possible sentence is *ttt*. Figure 7 shows the attributed derivation tree of *ttt* with attribute instances and their dependencies. The associated precedence graph is given in Fig. 8.

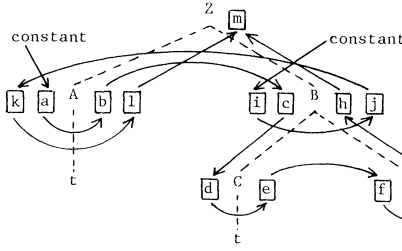


FIG. 7. Attributed derivation tree of grammar AG3.

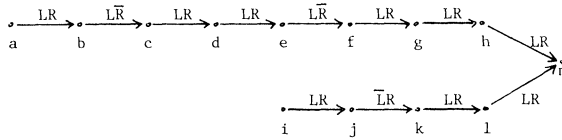


FIG. 8. P(AG3).

For sequence $\langle d_1^{\text{old}}, d_2^{\text{old}} \rangle = \langle L, R \rangle$ the partition associated with minmaxpass_0 is $\langle \{a, b, c, d, e\}, \{f, g, h, i, j, k, l, m\} \rangle$. Notice that arc (b, c) has label \overline{LR} . Hence, for d_1^{new} the original pass direction L is selected.

Now we consider the case where A'_{k+1} does not include arcs labeled \overline{rpd} . In that case we further compare the partitions associated with minmaxpass_{k+1} with respect to $\langle d_1^{\text{new}}, \dots, d_k^{\text{new}}, \text{opd}, d'_{k+2}, \dots, d'_p \rangle$ and $\langle d_1^{\text{new}}, \dots, d_k^{\text{new}}, \text{rpd}, d'_{k+2}, \dots, d'_p \rangle$.

If for $d_{k+1}^{\text{new}} = \text{opd}$ or $d_{k+1}^{\text{new}} = \text{rpd}$ all the attributes of some A'_{k+i} ($2 \leq i \leq p - k$) can be moved to A'_{k+1} , then it is possible to cross out the $(k + i)$ th pass. If passes become empty we select the pass direction for which the largest number of passes becomes empty.

Example 5.6. Consider attribute grammar AG4 with $V_N = \{Z, A, B, C, D\}$, $V_T = \{t\}$ and $P = \{Z \rightarrow ABCA, Z \rightarrow DD, A \rightarrow t, B \rightarrow t, C \rightarrow t, D \rightarrow t\}$.

The possible sentences are $tttt$ and tt . Figure 9 shows the attributed derivation trees for these sentences. The associated precedence graph is given in Fig. 10.

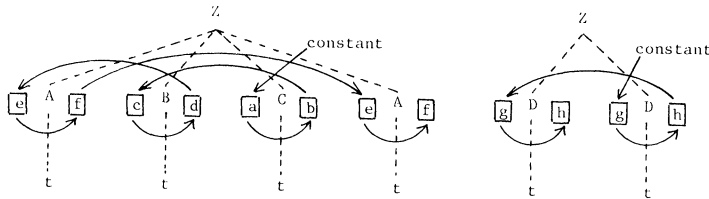


FIG. 9. Attributed derivation trees of grammar AG4.

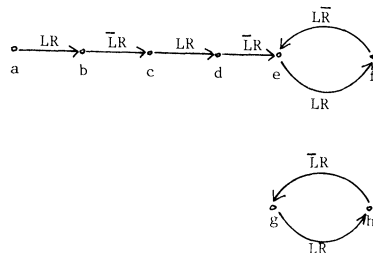


FIG. 10. P(AG4).

For sequence $\langle d_1^{old}, d_2^{old}, d_3^{old}, d_4^{old} \rangle = \langle L, L, L, R \rangle$ the partition associated with $\text{minmaxpass}_0 = \langle \{a, b\}, \{c, d\}, \{e, f\}, \{g, h\} \rangle$. $A_1' = \{a, b\}$ has no \bar{R} -label. Hence, correct pass functions exist for both $d_1^{new} = L$ and $d_1^{new} = R$.

For $d_1^{new} = L$, the partition associated with $\text{minmaxpass}_1 = \langle \{a, b\}, \{c, d\}, \{e, f\}, \{g, h\} \rangle$. Hence no passes become empty. For $d_1^{new} = R$ the partition associated with $\text{minmaxpass}_1 = \langle \{a, b, c, d, g, h\}, \{ \}, \{e, f\}, \{ \} \rangle$. Hence, the second and the fourth pass become empty. So for d_1^{new} direction R is selected.

The following example illustrates that it may happen that $d_{k+1}^{new} = \text{opd}$ and $d_{k+1}^{new} = \text{rpd}$ deliver equal numbers of empty passes. In such a case we need another criterion to select the $(k+1)$ th pass direction.

Example 5.7. Consider attribute grammar AG5 with $V_N = \{Z, A, B, C, D, E, F\}$, $V_T = \{t\}$ and $P = \{Z \rightarrow AA, Z \rightarrow BCCDD, Z \rightarrow EEDD, Z \rightarrow FFDD, A \rightarrow u, B \rightarrow v, C \rightarrow w, D \rightarrow x, E \rightarrow y, F \rightarrow z\}$.

The possible sentences are $uu, vwwxx, yyxx$ and $zzxx$. Figure 11 shows the attributed derivation trees for these sentences. The associated precedence graph is given in Fig. 12.

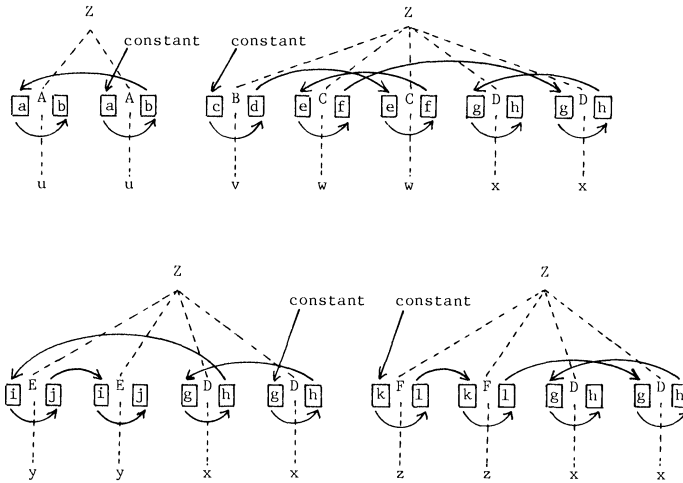


FIG. 11. Attributed derivation trees of grammar AG5.

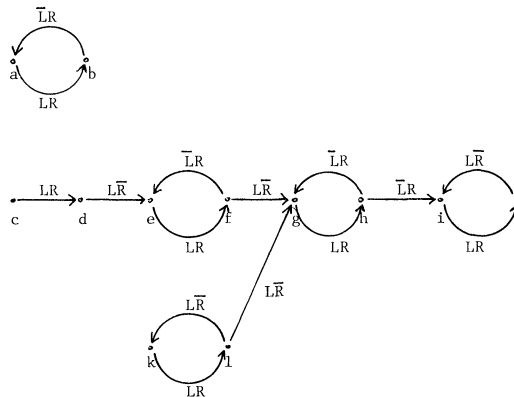


FIG. 12. $P(AG5)$.

For sequence $\langle d_1^{\text{old}}, d_2^{\text{old}}, d_3^{\text{old}}, d_4^{\text{old}}, d_5^{\text{old}}, d_6^{\text{old}} \rangle = \langle R, R, L, R, L, R \rangle$ the partition associated with $\text{minmaxpass}_0 = \langle \{c, d\}, \{e, f\}, \{k, l\}, \{g, h\}, \{i, j\}, \{a, b\} \rangle$. For $d_1^{\text{new}} = L$ the partition associated with $\text{minmaxpass}_1 = \langle \{c, d, k, l\}, \{e, f\}, \{g, h\}, \{i, j\}, \{a, b\} \rangle$, in which the third pass is empty. For $d_1^{\text{new}} = R$ the partition associated with $\text{minmaxpass}_1 = \langle \{a, b, c, d\}, \{e, f\}, \{k, l\}, \{g, h\}, \{i, j\}, \{ \} \rangle$, in which the sixth pass is empty. Hence, in this case it is impossible to select the pass direction for d_1^{new} by counting the number of empty passes.

If no passes become empty or if for both directions of the $(k + 1)$ th pass the same number of empty passes is found, we try to move as many attributes as possible from A_{k+2}^i to A_{k+1}^{new} . The more attributes can be moved to A_{k+1}^{new} the easier it will be to combine the $(k + 2)$ th pass with following passes.

We compare the subsets of A_{k+2}^i that can be moved to A_{k+1}^{new} for pass direction opd and rpd respectively. If one subset is contained in the other the direction for the larger subset is chosen. If the subsets are equal, we compare the subsets of A_{k+3}^i that can possibly be moved to A_{k+1}^{new} , and so on. Observe that the intent of this order of comparing subsets is that first the attributes with highest priority are considered, i.e., the attributes of A_{k+2}^i whose evaluation can be postponed at most one pass.

If however subsets are incomparable we need another selection criterion. Instead of subsets cardinalities of subsets can be compared or the number of arcs labeled \bar{L} or \bar{R} between attributes in subsets. Such criteria are not considered in this paper.

To get more insight in the possibility of selecting a pass direction by comparing subsets, we discuss two examples.

Example 5.8. Consider with respect to sequence $\langle \dots, d'_{k+1} = L, d'_{k+2} = L, \dots \rangle$ of pass directions the partition $\langle \dots, A'_{k+1}, A'_{k+2}, \dots \rangle$ associated with pass function minmaxpass_k for precedence graph $P(\text{AG6})$ as sketched in Fig. 13. The solid lines denote paths composed of arcs without labels \bar{L} and \bar{R} , unless explicitly indicated otherwise.

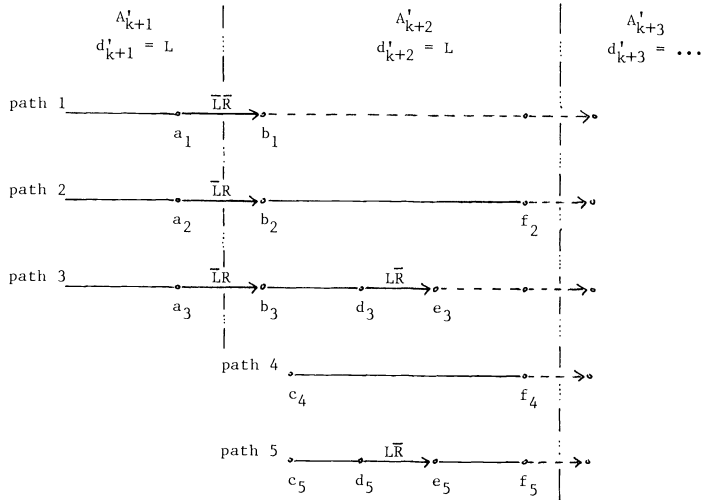


FIG. 13. Part of $P(\text{AG6})$.

Now for the partition associated with minmaxpass_{k+1} we have to choose between $d_{k+1}^{\text{new}} = L$ and $d_{k+1}^{\text{new}} = R$. For $d_{k+1}^{\text{new}} = R$ the subset of attributes from A'_{k+2} moved to A_{k+1}^{new} is $\{b_2, \dots, f_2, b_3, \dots, d_3, c_4, \dots, f_4, c_5, \dots, d_5\}$ and for $d_{k+1}^{\text{new}} = L$ the subset of attributes from A'_{k+2} moved to A_{k+1}^{new} is $\{c_4, \dots, f_4, c_5, \dots, d_5, e_5, \dots, f_5\}$. These subsets

are incomparable. But observe that if paths 2 and 3 are not in the graph, then $d_{k+1}^{new} = L$ is selected and if path 5 does not form part of it, then $d_{k+1}^{new} = R$ is selected.

Example 5.9. Consider with respect to sequence $\langle \dots, d'_{k+1} = L, d'_{k+2} = R, \dots \rangle$ of pass directions the partition $\langle \dots, A'_{k+1}, A'_{k+2}, \dots \rangle$ associated with pass function minmaxpass_k for precedence graph $P(\text{AG7})$ as sketched in Fig. 14.

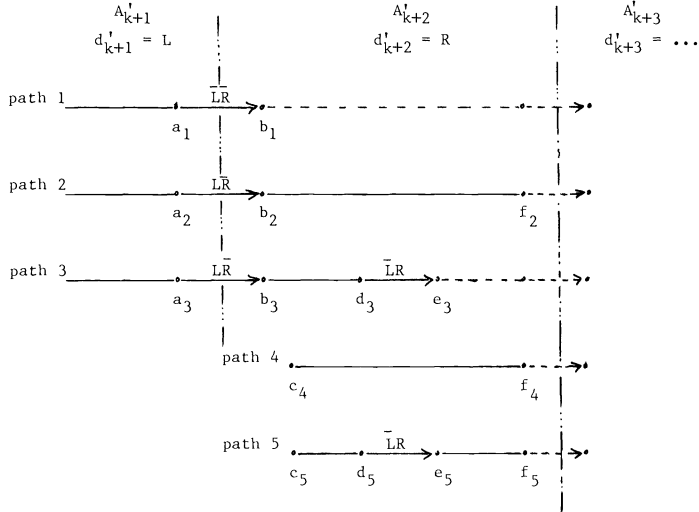


FIG. 14. Part of $P(\text{AG7})$.

For $d_{k+1}^{new} = L$ the subset of attributes from A'_{k+2} moved to A_{k+1}^{new} is $\{b_2, \dots, f_2, b_3, \dots, d_3, c_4, \dots, f_4, c_5, \dots, d_5\}$ and for $d_{k+1}^{new} = R$ the subset of attributes from A'_{k+2} moved to A_{k+1}^{new} is $\{c_4, \dots, f_4, c_5, \dots, d_5, e_5, \dots, f_5\}$. These subsets are incomparable. However, if paths 2 and 3 are not in the graph, then for d_{k+1}^{new} the reversed pass direction is selected and if on the contrary path 5 is not in the graph, then for d_{k+1}^{new} the original pass direction is maintained.

Let $\langle A_1^{new}, \dots, A_k^{new}, A_{k+1}^{opd}, A_{k+2}^{opd}, \dots, A_p^{opd} \rangle$ be the partition associated with the pass function minmaxpass_{k+1} with respect to the sequence $\langle d_1^{new}, \dots, d_k^{new}, \text{opd}, d'_{k+2}, \dots, d'_p \rangle$ of pass directions and let $\langle A_1^{new}, \dots, A_k^{new}, A_{k+1}^{rpd}, A_{k+2}^{rpd}, \dots, A_p^{rpd} \rangle$ be the partition associated with the pass function minmaxpass_{k+1} with respect to the sequence $\langle d_1^{new}, \dots, d_k^{new}, \text{rpd}, d'_{k+2}, \dots, d'_p \rangle$.

Notice that $A_{k+2}^{rpd} \subset A_{k+2}^{opd}$ means that the subset of A'_{k+2} moved to A_{k+1}^{opd} is included in the subset of A'_{k+2} moved to A_{k+1}^{rpd} . Hence, if $A_{k+2}^{rpd} \subset A_{k+2}^{opd}$ then we select rpd as the direction for the $(k+1)$ th pass and if $A_{k+2}^{opd} \subset A_{k+2}^{rpd}$ then we select opd as the $(k+1)$ th pass direction. If A_{k+2}^{rpd} and A_{k+2}^{opd} are incomparable we have to use another criterion to select a direction for the $(k+1)$ th pass. If $A_{k+2}^{rpd} = A_{k+2}^{opd}$ we continue the selection process by comparing A_{k+3}^{opd} and A_{k+3}^{rpd} . The process continues until $A_{k+i}^{opd} \subset A_{k+i}^{rpd}$ or $A_{k+i}^{rpd} \subset A_{k+i}^{opd}$ for some i ($2 \leq i \leq p-k$). If finally the comparison of A_p^{opd} and A_p^{rpd} does not lead to a decision we may arbitrarily select a direction for the $(k+1)$ th pass.

Algorithm 5.3 starts with the partition $\langle A_1^{old}, \dots, A_m^{old} \rangle$ associated with pass function $\text{minmaxpass}_0 = \text{maxpass}$ with respect to sequence $\langle d_1^{old}, \dots, d_m^{old} \rangle$ of pass directions, where $m = m_s$ the initial number of passes. Now for k from 0 up to $m-2$ action 1 or 2 is taken.

1. If in the partition $\langle A_1^{new}, \dots, A_k^{new}, A'_{k+1}, A'_{k+2}, \dots, A'_m \rangle$ associated with minmaxpass_k with respect to sequence $\langle d_1^{new}, \dots, d_k^{new}, d'_{k+1}, d'_{k+2}, \dots, d'_m \rangle$ of pass directions the set A'_{k+1} is empty, then pass direction d'_{k+1} is crossed out from the sequence

of pass directions and the set A'_{k+1} from the partition associated with minmaxpass_k . The remaining pass directions and sets of the partition are renumbered such that again a consecutive sequence of pass numbers results. The resulting partition has the minmaxpass_k property with respect to the shortened sequence of pass directions.

2. If the $(k+1)$ th pass cannot be crossed out, then procedure next minmaxpass is applied. It selects the $(k+1)$ th pass direction and computes the partition associated with the pass function minmaxpass_{k+1} . Starting from a sequence $\langle d_1^{\text{new}}, \dots, d_k^{\text{new}}, d'_{k+1}, d'_{k+2}, \dots, d'_m \rangle$ of pass directions and the partition associated with the pass function minmaxpass_k with respect to this sequence it delivers the sequence $\langle d_1^{\text{new}}, \dots, d_k^{\text{new}}, d'_{k+1}, d'_{k+2}, \dots, d'_m \rangle$ and the partition associated with the pass function minmaxpass_{k+1} with respect to the resulting sequence of pass directions, where d'_{k+1} is the "best" direction.

The repetition ends with the sequence $\langle d_1^{\text{new}}, \dots, d_m^{\text{new}} \rangle$ of pass directions and the partition $\langle A_1^{\text{new}}, \dots, A_m^{\text{new}} \rangle$ associated with the pass function $\text{minmaxpass}_{m-1} = \text{minmaxpass}_m = \text{minpass}$, where $m = m_f$ the final number of passes.

Procedure next minmaxpass makes use of the function nextmin to compute A_{k+1}^{new} , the $(k+1)$ th subset of the new partition.

Function nextmin is useful for both the case where d_{k+1}^{new} has the original value d'_{k+1} and where d_{k+1}^{new} and d'_{k+1} are reversed directions (see also the optimized version of Algorithm 4.1).

Function nextmin makes use of the following sets: A: the set of all attributes of the grammar; B: the attributes that belong to preceding subsets in the partition, i.e., $\cup_{i=1}^k A_i^{\text{new}}$; C: the attributes that certainly belong to A_{k+1}^{new} , i.e., A'_{k+1} ; D: the attributes that possibly belong to A_{k+1}^{new} , i.e., initially $A - (B \cup C)$.

ALGORITHM 5.3. Computation of the partition associated with the pass function minpass with respect to a sequence $\langle d_1^{\text{new}}, \dots, d_{m_f}^{\text{new}} \rangle$ of pass directions from the partition associated with the pass function maxpass with respect to a sequence $\langle d_1^{\text{old}}, \dots, d_{m_s}^{\text{old}} \rangle$ of pass directions, where $m_f \leq m_s$.

Input: $P(\text{AG})$; sequence $\langle d_1^{\text{old}}, \dots, d_{m_s}^{\text{old}} \rangle$ of pass directions; partition $\langle A_1^{\text{old}}, \dots, A_{m_s}^{\text{old}} \rangle$ associated with the pass function maxpass with respect to $\langle d_1^{\text{old}}, \dots, d_{m_s}^{\text{old}} \rangle$.

Output: sequence $\langle d_1^{\text{new}}, \dots, d_{m_f}^{\text{new}} \rangle$ of pass directions; partition $\langle A_1^{\text{new}}, \dots, A_{m_f}^{\text{new}} \rangle$ associated with the pass function minpass with respect to $\langle d_1^{\text{new}}, \dots, d_{m_f}^{\text{new}} \rangle$.

Algorithm:

begin

procedure $\text{maxpass to minpass}$ (PG : precedence graph);

{globals $\langle d_1^{\text{old}}, \dots, d_{m_s}^{\text{old}} \rangle$: initial sequence of pass directions;

$\langle A_1^{\text{old}}, \dots, A_{m_s}^{\text{old}} \rangle$: initial maxpass partition of A ;

$\langle d_1^{\text{new}}, \dots, d_{m_f}^{\text{new}} \rangle$: final sequence of pass directions;

$\langle A_1^{\text{new}}, \dots, A_{m_f}^{\text{new}} \rangle$: final minpass partition of A }

function nextmin (A, B, C : set of attributes; d : pass direction;

PG : precedence graph): set of attributes;

var D : set of attributes;

begin

$D := A - (B \cup C)$;

repeat

delete a vertex a from D

if in PG there exists a vertex b such that

$b \in C \cup D$ and $d\text{-cost}(b, a) > 1$


```

    until no more vertices can be deleted from  $D$ ;
    nextmin :=  $D \cup C$ 
end {of nextmin};
procedure next minmaxpass ( $k$ : integer;  $PG$ : precedence graph);
{global variables:
 $\langle d_1^{\text{new}}, \dots, d_k^{\text{new}}, d'_{k+1}, d'_{k+2}, \dots, d'_m \rangle$ :
    initial sequence of pass directions;
 $\langle A_1^{\text{new}}, \dots, A_k^{\text{new}}, A'_{k+1}, A'_{k+2}, \dots, A'_m \rangle$ :
    initial minmaxpass $_k$  partition of  $A$ ;
 $\langle d_1^{\text{new}}, \dots, d_k^{\text{new}}, d'_{k+1}, d'_{k+2}, \dots, d'_m \rangle$ :
    final sequence of pass directions;
 $\langle A_1^{\text{new}}, \dots, A_k^{\text{new}}, A'_{k+1}, A'_{k+2}, \dots, A'_m \rangle$ :
    final minmaxpass $_{k+1}$  partition of  $A$ }
begin
    opd :=  $d'_{k+1}$ ; rpd := reverse ( $d'_{k+1}$ ); direction := ?;
    if  $A'_{k+1}$  includes rpd-labels
    then direction := opd
    else  $A_{k+1}^{\text{opd}}$  := nextmin ( $A, \cup_{i=1}^k A_i^{\text{new}}, A'_{k+1}, \text{opd}, PG$ );
         $A_{k+1}^{\text{rpd}}$  := nextmin ( $A, \cup_{i=1}^k A_i^{\text{new}}, A'_{k+1}, \text{rpd}, PG$ );
        for  $i$  from  $k+2$  to  $m$ 
        do  $A_i^{\text{opd}}$  :=  $A_i - A_{k+1}^{\text{opd}}$ ;  $A_i^{\text{rpd}}$  :=  $A_i - A_{k+1}^{\text{rpd}}$  od;
        if |empty opd sets| < |empty rpd sets|
        then direction := rpd
        else if |empty rpd sets| < |empty opd sets|
        then direction := opd
        else for  $i$  from  $k+2$  to  $m$  while direction = ?
            do if  $A_i^{\text{opd}} \subset A_i^{\text{rpd}}$ 
                then direction := opd
                else if  $A_i^{\text{rpd}} \subset A_i^{\text{opd}}$ 
                then direction := rpd
                else if  $A_i^{\text{opd}}$  and  $A_i^{\text{rpd}}$  are incomparable
                then direction := other criterion
                fi
            fi
        od
        fi
        if direction = ? then direction := arbitrary fi
    fi;
     $d_{k+1}^{\text{new}}$  := direction;
    if direction = opd
    then  $\langle A_{k+1}^{\text{new}}, A'_{k+2}, \dots, A'_m \rangle$  :=  $\langle A_{k+1}^{\text{opd}}, A_{k+2}^{\text{opd}}, \dots, A_m^{\text{opd}} \rangle$ 
    else  $\langle A_{k+1}^{\text{new}}, A'_{k+2}, \dots, A'_m \rangle$  :=  $\langle A_{k+1}^{\text{rpd}}, A_{k+2}^{\text{rpd}}, \dots, A_m^{\text{rpd}} \rangle$ 
    fi
end {of next minmaxpass};
begin
     $m$  :=  $m_s$ ;  $k$  := 0;
     $\langle A'_1, \dots, A'_m \rangle$  :=  $\langle A_1^{\text{old}}, \dots, A_m^{\text{old}} \rangle$ ;
    {maxpass partition of  $A$  with respect to sequence  $\langle d_1^{\text{old}}, \dots, d_m^{\text{old}} \rangle$  of pass
    directions}

```

```

 $\langle d'_1, \dots, d'_m \rangle := \langle d_1^{\text{old}}, \dots, d_m^{\text{old}} \rangle;$ 
while  $k \leq m - 2$ 
do
  consider partition  $\langle A_1^{\text{new}}, \dots, A_k^{\text{new}}, A'_{k+1}, A'_{k+2}, \dots, A'_m \rangle$ 
  associated with  $\text{minmaxpass}_k$  with respect to sequence
   $\langle d_1^{\text{new}}, \dots, d_k^{\text{new}}, d'_{k+1}, d'_{k+2}, \dots, d'_m \rangle$  of pass directions;
  if  $A'_{k+1} = \emptyset$ 
  then {cross out and renumber}
     $\langle A'_{k+1}, \dots, A'_{m-1} \rangle := \langle A'_{k+2}, \dots, A'_m \rangle;$ 
     $\langle d'_{k+1}, \dots, d'_{m-1} \rangle := \langle d'_{k+2}, \dots, d'_m \rangle;$ 
     $m := m - 1$ 
  else next  $\text{minmaxpass}(k, PG);$ 
     $k := k + 1$ 
  fi
od;
 $m_f := m$ 
end {of maxpass to minpass};
call maxpass to minpass ( $P(AG)$ )
end

```

Notice that Algorithms 5.1 and 5.3 deliver the same sequence of pass directions if not a single pass of the original sequence is replaced by a pass in the opposite direction.

As for Algorithm 5.1 we count the number of times the d -cost function between attributes is examined. The function nextmin is called twice in the body of function next minmaxpass . Hence, Algorithm 5.3 takes time $O(m_f \cdot n^2)$, where n is the number of attributes of the grammar.

A similar approach as in Algorithm 5.3 is possible for the computation of the partition associated with the pass function maxpass from the partition associated with the pass function minpass . As in § 4 we make use of the correspondence between the minpass to maxpass computation and the maxpass to minpass computation. After putting the sequence of pass directions and subsets of the partition of A into the reversed order, the minpass to maxpass computation can be realized by activation of the procedure maxpass to minpass with the reversed precedence graph $\tilde{P}(AG)$ as its argument.

Hence, our algorithm to reduce the number of evaluation passes by a repeated application of Algorithm 5.3 is as follows.

ALGORITHM 5.4. Computation of a minimal (minimal with respect to the sub-sequence ordering) sequence of pass directions and the pass function minpass with respect to this sequence.

Input: attribute grammar AG .

Output: a sequence of pass directions and the minpass partition of the attributes of AG associated with this sequence.

Algorithm:

```

begin
  construct  $P(AG)$  and  $\tilde{P}(AG);$ 
  construct a sequence of pass directions and the  $\text{minpass}$  partition with respect
  to this sequence and  $P(AG);$ 
  apply Algorithm 5.2 to cross out pass directions and to compute the pass
  function  $\text{maxpass}$  with respect to the resulting sequence of pass directions;

```

repeat

old m := length of the sequence of pass directions;

call maxpass to minpass ($P(AG)$);

put the pass directions of the resulting sequence and subsets of the resulting partition into the reversed order;

call maxpass to minpass ($\tilde{P}(AG)$);

put the pass directions of the resulting sequence and subsets of the resulting partition into the reversed order;

new m := length of the sequence of pass directions

until *new m* = *old m*;

apply Algorithm 5.1 to cross out pass directions and to compute the pass function minpass with respect to the resulting sequence of pass directions

end

The resulting sequence of pass directions is minimal with respect to the sub-sequence ordering.

Any sequence of pass directions for which a correct complete partition of the set of attributes exists can be used to start the optimization process. A possible sequence could be $\langle L, R, L, R \cdot \cdot \cdot \rangle$, i.e., the sequence where L - and R -passes strictly alternate. In [1] an algorithm is presented that produces with respect to such a sequence the minimal pass numbers and in case of failure indicates the attributes that cause the rejection of the grammar, i.e., the attributes that are involved in a cycle whose labels are not consistent with one of the possible pass directions.

The importance of this algorithm for strictly alternating simple multi-pass evaluation follows from the fact that an attribute grammar is simple multi-pass with respect to any sequence of pass directions if and only if it is simple multi-pass with respect to the sequence of pass directions where left-to-right and right-to-left passes strictly alternate [1, Thm. 8.1].

6. Discussion and conclusions. Algorithm 5.4 proved to be successful for several small example grammars discussed in the literature, e.g., in [2]–[12]. As starting sequences of pass directions both the strictly alternating sequences $\langle L, R, L, R, \cdot \cdot \cdot \rangle$ and $\langle R, L, R, L, \cdot \cdot \cdot \rangle$ were applied.

For all these examples handchecking showed that:

1. passes were crossed out from the original sequence and there was no need to change the directions of the remaining passes, although in some cases the changing of a pass direction into the opposite one would have given the same result;
2. sequences of pass directions of minimal length were produced. Hence, for the more or less practical examples, referred to in the literature mentioned above, Algorithms 5.1 and 5.2 deliver the same results as Algorithm 5.4. Changing of pass directions seems to be needed only in complicated cases.

The construction algorithm in this paper starts from a given nonoptimal solution and subsequently tries to shorten it by crossing out passes and changing pass directions.

Other algorithms, e.g. in [8], start from an empty sequence of pass directions and successively extend it by selecting the direction for the next pass. These algorithms compare the sets of attributes that can be evaluated during a left-to-right pass and during a right-to-left pass. If one of these sets is contained in the other, the direction of the larger set is chosen. Notice that this criterion is also implicitly included in Algorithm 5.4 in this paper. If the two sets are incomparable, then for the next pass direction the reversed of the previous pass is chosen.

In [10], [11] a better criterion is explained. It is assumed that for the remaining attributes, not yet evaluated, still another left-to-right pass has to be made. The next left-to-right pass will be preceded by a sequence of zero or more right-to-left passes. If the set of attributes evaluated during the left-to-right pass will not grow regardless of the number of preceding right-to-left passes, there is no reason for delaying the left-to-right pass. Doing some passes in the opposite direction will not help. Analogously the completeness of a right-to-left pass immediately after a number of left-to-right passes is investigated. In case these criteria do not lead to a decision other heuristics are added to the selection function.

Notice that the algorithms in this paper start from given sequences of pass directions, especially the strictly alternating sequences, while the algorithm in [10], [11] repeatedly computes subsequences $\langle L^n, R \rangle$ and $\langle R^n, L \rangle$.

Both the algorithms in this paper and the algorithm in [10], [11] take time $O(n^3)$, where n is the number of attributes of the grammar, and seem to deliver the same results for practical example grammars. Artificial examples are needed to find differences.

An interesting point is that for a given sequence of pass directions several different pass functions may exist. The distribution of the attributes over the passes may influence the lifetime of the attributes, i.e., the span of time between the point when an attribute is defined and the point when its value is used for the last time. Hence, the freedom in the distribution of the attributes over the passes can be applied for space efficient storage management in an attribute grammar evaluator, see [4].

Acknowledgments. I am grateful to Joost Engelfriet for his stimulating and critical comments and to Elvira Dijkhuis and Thérèse ter Heide for rapid typing.

REFERENCES

- [1] H. ALBLAS, *A characterization of attribute evaluation in passes*, Acta Inform., 16 (1981), pp. 427–464.
- [2] G. V. BOCHMANN, *Semantic evaluation from left to right*, Comm. ACM, 19 (1976), pp. 55–62.
- [3] R. GIEGERICH AND R. WILHELM, *Counter-one-pass features in one-pass compilation: a formalization using attribute grammars*, Inform. Proc. Lett., 7 (1978), pp. 279–284.
- [4] M. JAZAYERI AND D. POZEFSKI, *Space efficient storage management in an attribute evaluator*, ACM Trans. Programming Languages and Systems, 3 (1981), pp. 388–404.
- [5] M. JAZAYERI AND K. G. WALTER, *Alternating semantic evaluator*, Proc. ACM 1975 Annual Conference, 1975, pp. 230–234.
- [6] D. E. KNUTH, *Semantics of context-free languages*, Math. Systems Theory, 2 (1968), pp. 117–145.
- [7] ———, *Examples of formal semantics*, Symposium on Semantics of Algorithmic Languages, E. Engeler, ed., Lecture Notes in Mathematics 188, Springer-Verlag, Berlin-Heidelberg-New York, 1971, pp. 212–235.
- [8] K.-J. RÄIHÄ, *On attribute grammars and their use in a compiler writing system*, Report A-1977-4, Dept. Computer Science, Univ. of Helsinki, 1977.
- [9] K.-J. RÄIHÄ AND M. SAARINEN, *Developments in compiler writing systems*, GI-6 Jahrestagung, Informatik Fachberichte 5, Springer-Verlag, Berlin-Heidelberg-New York, 1976, pp. 164–178.
- [10] K.-J. RÄIHÄ AND E. UKKONEN, *On the optimal assignment of attributes to passes in multi-pass attribute evaluators*, Proc. 7th Int. Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 85, Springer-Verlag, Berlin-Heidelberg-New York, 1980, pp. 500–511.
- [11] ———, *Minimizing the number of evaluation passes for attribute grammars*, this Journal, 10 (1981), pp. 772–786.
- [12] R. WILHELM, *Attributierte Grammatiken*, Informatik-Spektrum, 2 (1979), pp. 123–130.

OPTIMUM COMMUNICATION SPANNING TREES IN SERIES-PARALLEL NETWORKS*

EHAB S. EL-MALLAH† AND CHARLES J. COLBOURN‡

Abstract. The optimum communication spanning tree problem is to locate a spanning tree which minimizes the sum of the lengths of the shortest routes between all pairs of vertices in a graph, weighted by traffic requirements. Although NP-complete in general, this problem has an efficient solution for series-parallel graphs when all requirements are equal. This problem was introduced by Hu, who gave an efficient solution for the restricted case when the network is complete and the distances are equal.

Key words. spanning tree, network design, series-parallel graph, 2-tree

1. Introduction. A common objective of most network design problems is to construct a network to connect a set of sites while minimizing construction costs and routing costs associated with satisfying the required flow. The optimum communication spanning tree problem (OCSTP) is a special case of noncongested network design problems, in which edge construction costs are all equal and totally dominate the routing costs (see for example [10] for a survey of some network design problems). In these cases the network topology that minimizes the construction costs while maintaining connectedness is a spanning tree. The network under consideration is modelled by an undirected graph $G = (V, E)$, $|V| = n$ and $|E| = m$; the distance and the traffic requirements between any two vertices i and j is denoted by d_{ij} and r_{ij} respectively. The problem asks for a spanning tree T of the graph G that minimizes the total routing cost defined by the function $F(T) = \sum_{i < j, i, j \in V} r_{ij} L_{ij}$, where L_{ij} is the sum of the distances of the edges which form the unique path connecting i and j in T . For example, if G is the r -vertex complete graph K_r , with all possible distances and traffic requirements equal to one, then the criterion function of any tree T isomorphic to the star tree $K_{1, r-1}$ is $(r-1)^2$. In this case, T is an optimum communication tree.

Hu [8] first introduced this problem and identified two interesting cases: the optimum requirement spanning tree problem (ORSTP) where all the distances are restricted to be equal and the optimum distance spanning tree problem (ODSTP) where all the requirements are restricted to be equal. Hu solved the ORSTP and a geometric ODSTP in polynomial time on complete graphs. Johnson, Lenstra, and Rinnooy Kan proved that the OCSTP is NP-complete even in the restricted case where all distances and requirements are equal [9]. Hence, an efficient solution for any arbitrary instance of the problem is unlikely.

In this paper, we devise an algorithm that solves the ODSTP in $O(n^3 + m)$ time on series-parallel networks. Note that the problem can not be characterized by a forbidden set of subgraphs; hence, the divide and conquer algorithm [12] for solving some graph problems on series-parallel networks is not applicable.

Throughout this paper we assume standard graph-theoretic terminology and definitions (see, for example, [1], [6]). In addition, some basic definitions follow. Duffin [3] characterized *series-parallel graphs* as those graphs with no induced subgraph homeomorphic to K_4 . A 2-tree is defined recursively as follows. A triangle, K_3 , is a

* Received by the editors February 7, 1984, and in revised form July 3, 1984.

† Department of Computational Science, University of Saskatchewan, Saskatoon, S7N 0W0, Canada.

‡ Department of Computer Science, University of Waterloo, Waterloo, N2L 3G1, Canada. The work of this author was supported by the Natural Sciences and Engineering Research Council of Canada under grant A5047.

2-tree. Further, given a 2-tree and an edge (x, y) of the 2-tree, we can add a new vertex z adjacent to both x and y to produce a 2-tree. A partial 2-tree is a subgraph of a 2-tree. Finally, it can be shown that the class of partial 2-trees is precisely the class of series-parallel graphs without loops or parallel edges [14].

The rest of this paper is organized as follows. Section 2 describes an algorithm to transform a series-parallel network to a 2-tree such that both graphs have the same optimum distance trees. Section 3 describes a recursive algorithm for generating spanning trees in a 2-tree; this algorithm is a prototype of the main algorithm. Section 4 defines two optimality criterion functions that will be used to identify the subtrees that are considered candidates to appear in the optimum tree. Section 5 outlines the algorithm and proves its correctness and the timing required.

2. Transforming a series-parallel graph to a 2-tree. In this section we outline a method for transforming a series-parallel graph $G = (V, E)$ to a 2-tree $G'' = (V'', E'')$ such that a tree T is optimum in G if and only if it is optimum in G'' . The transformation proceeds as follows. Reduce every set of parallel edges in G to one edge, namely the one with the shortest distance. The resulting graph $G' = (V', E')$ is the underlying graph of the graph G without parallel edges, and hence it is a partial 2-tree (the two classes of graphs are characterized by forbidden subgraphs homeomorphic to K_4). Since graph G is assumed to be connected and removing the parallel edges does not interrupt its connectivity, we may assume that G' is a connected partial 2-tree. Clearly, both G and G' have the same optimum distance spanning trees. The above step can be implemented in $O(m)$ time. If $|E'| = 2|V'| - 3$ then stop; G' is a 2-tree, otherwise add one or more new edges to transform G' to a 2-tree G'' as follows.

In the following steps we associate a large distance with every new added edge to ensure that none of them is included in any optimum distance spanning tree. One such value is the criterion function of any spanning tree of G' . First, add the necessary edges to transform the graph G' to a biconnected partial 2-tree G'' using a standard depth-first technique [13]. This step can be implemented in $O(n + m)$ time. Given a biconnected partial 2-tree G'' , transform it to a 2-tree as follows [14]. Form a queue containing all vertices of degree 2 in G'' . Then repeatedly perform the following operations, on a copy G''' of G'' , until G''' is reduced to K_2 . Remove the first vertex v from the queue; if there is no such queue element and G''' is not a K_2 , declare that G'' is not a partial 2-tree and stop. Otherwise, locate the neighbours x and y of v in G''' . If (x, y) is not an edge, add (x, y) to the graphs G'' and G''' . Delete the vertex v ; if either x or y has its degree decreased to 2 in the graph G''' , add it to the queue. This transformation requires $O(n)$ time. As can be seen the transformation from the connected series-parallel graph G to the 2-tree G'' requires $O(n + m)$ time. We henceforth assume that the input graph G is a 2-tree.

3. Recursive generation of spanning trees in 2-trees. 2-trees possess the following edge separation property: each edge (x, y) of a 2-tree G partitions G into one or more components which pairwise intersect at (x, y) and whose union is the entire 2-tree; moreover, each component so obtained is a 2-tree. A subgraph G_1 of G is called a *side* of (x, y) if it is the union of one or more such components. Note that the edge (x, y) and the entire graph G are the minimal and maximal sides of (x, y) respectively.

The above property enables us to view any spanning tree T of the 2-tree G as the union of certain subgraphs of the sides of any edge (x, y) . In fact, the subgraphs of any side G_i of the edge (x, y) that can appear in any spanning tree T of G may be classified into two types. Members of the first type are spanning trees of this particular side and are referred to as the *CT* trees; we use the notation $CT_i(x, y)$ to refer to a

spanning tree of the i th side of (x, y) . Members of the second type are spanning forests of the side under consideration; each forest contains exactly two disjoint subtrees, one containing x and the other containing y , and are referred to as the DT forests. We use the notation $DT_i(x, y)$ to refer to such a spanning forest of the i th side of (x, y) . Following the above notations we can write any spanning tree of G as the union of some CT and DT forests with respect to any edge (x, y) with k identified sides as

$$\bigcup_{\substack{1 \leq j \leq k \\ j \neq i}} DT_j(x, y) \cup CT_i(x, y)$$

for some side C_i . Note that no other forest of any side G_i (that is not of type CT or DT) can appear in any spanning tree of G since any other type of forests contains at least one vertex v isolated from both x and y . Hence, this vertex v appears in the final spanning subgraph isolated from all other vertices in all other sides.

The general scheme is to reverse the recursive construction process of the 2-tree by repeatedly eliminating vertices of degree 2 until the graph is reduced to a base component K_2 . During this vertex elimination procedure, summarize information about the triangle (x, y, z) , where z has degree 2, on the edge (x, y) prior to deleting z . For our purpose, the summary information associated with every edge (x, y) consists of two sets $S_CT(x, y)$ and $S_DT(x, y)$ whose members are spanning trees of type CT and spanning forests of type DT respectively. At any time, the summary information encodes information about the CT trees and the DT forests of the side which has thus far been reduced onto the edge (x, y) . This technique owes much to similar techniques used to compute the center and diameter of outerplanar graphs [5] and Steiner trees in 2-trees [14].

Initially, no side has been reduced onto the edge (x, y) ; the only member of the set $S_CT(x, y)$ is the edge (x, y) itself. Similarly, the only member of the set $S_DT(x, y)$ is the pair of vertices x and y . Subsequently, prior to eliminating any vertex z of degree 2 in the triangle (x, y, z) the two sets $S_CT(x, y)$ and $S_DT(x, y)$ are updated to include all possible forests that can be constructed from the forests included in the six sets $S_CT(x, y)$, $S_DT(x, y)$, $S_CT(x, z)$, $S_DT(x, z)$, $S_CT(y, z)$ and $S_DT(y, z)$. The update operations are described using the cross product operation defined on graphs as follows: let S_1 and S_2 be two sets of graphs; then $S_1 \times S_2$ is the set containing all possible graphs that can be constructed by applying the graph union operation on every pair of graphs $G_1 \in S_1$ and $G_2 \in S_2$. The following procedure summarizes the main steps of the algorithm.

GENERATE_SPANNING_TREES(G)

Input: A 2-tree G .

Output: All spanning trees of G .

1. For every edge $(x, y) \in E$,
 - 1.1. $S_CT(x, y) \leftarrow$ the edge (x, y) .
 - 1.2. $S_DT(x, y) \leftarrow$ the trivial forest of the two separate vertices x and y .
2. Form a queue of degree 2 vertices in G .
3. Repeat until the graph is reduced to K_2 .
 - 3.1. Remove vertex z from the queue. Locate its neighbours x and y .
 - 3.2. Merge the four sets associated with the edges (x, z) and (y, z) into temporary sets.
 - 3.2.1. $S_CT_{temp}(x, y) \leftarrow S_CT(x, z) \times S_CT(y, z)$.
 - 3.2.2. $S_DT_{temp}(x, y) \leftarrow \{S_CT(x, z) \times S_DT(y, z)\} \cup \{S_DT(x, z) \times S_CT(y, z)\}$.

- 3.3. Update the set of forests associated with the edge (x, y) .
 - 3.3.1. $S_CT(x, y) \leftarrow \{S_CT(x, y) \times S_DT_{temp}(x, y)\} \cup \{S_DT(x, y) \times S_CT_{temp}(x, y)\}$.
 - 3.3.3. $S_DT(x, y) \leftarrow S_DT(x, y) \times S_DT_{temp}(x, y)$.
- 3.4. Delete the vertex z . If either x or y has its degree decreased to 2 add it to the queue.
- 4. The set $S_CT(x, y)$ associated with the last edge (x, y) is the set of all spanning trees of G .

Note that in Step 3.2 the only omitted combination is $S_DT(x, z) \times S_DT(y, z)$. Here every forest contains exactly three subtrees, and hence none of them is of type CT or DT . Similarly, in Step 3.3 the only omitted combination is $S_CT_{temp}(x, y) \times S_CT(x, y)$ in which every graph contains a cycle joining the two vertices x and y . Clearly, such graphs are not forests.

The main algorithm, as described in § 5, generates in Steps 3.2 and 3.3 those forests that are considered candidates to appear in the optimum tree selected according to certain selection functions. In the following section, we introduce the selection functions used.

4. Optimality functions for selecting optimal subtrees. The problem of finding the optimum distance tree on a 2-tree can be regarded as the sequence of decisions concerning the forests of type CT and DT that should be *generated* prior to eliminating a degree 2 vertex and the information associated with the two edges incident with it. To this end, we define two selection functions that are used throughout the algorithm to decide the exact forests to be generated.

First, we extend a method adopted by Hu [8] to compute the criterion function of a communication tree T in the ODSTP. Assume that the removal of the edge $(i, j), (i, j) \in T$, disconnects T into two subtrees of sizes k_{ij} and $(n - k_{ij})$ respectively. The factor $d_{ij}k_{ij}(n - k_{ij})$ is the cost incurred by allowing the flow between the these two subtrees to be routed through the edge (i, j) . The sum of all such edge costs of T yields $F(T)$. For example, the cost of any edge of length one of the star tree $K_{1,r-1}$ is $r - 1$. Another method of computing the criterion function is to view T as a union of one or more subtrees and associating a cost with each subtree; the sum of all such costs yields $F(T)$. For our purpose, we introduce the following notations defined with respect to a tree T_1 containing n_1 vertices, one of them labelled x .

- 1. $k_{ij}(x)$ = the size of a subtree *not containing* x that results from removing the edge (i, j) from T_1 ; $1 \leq k_{ij} < n_1$.
- 2. $S_1(x)$ = the cost of routing the traffic from every vertex to the vertex x in the tree $T_1 = \sum_{(i,j) \in T_1} d_{ij}k_{ij}(x)$.

For example, if x is a vertex of degree 1 in the unit distance tree $K_{1,r-1}$ then $S(x) = 2r - 3$. The cost is computed by associating a cost of $d_{ij}k_{ij}(x)$ with every edge in T_1 and then summing all such edge costs.

Using the above notation, if x is a distinguished vertex in a tree T , T_1 and T_2 are two subtrees of T such that $T = T_1 \cup T_2$, and $V_1 \cap V_2 = \{x\}$ then $F(T)$ can be decomposed as follows.

$$\begin{aligned}
 F(T) &= \sum_{(i,j) \in T} d_{ij}k_{ij}(x)(n - k_{ij}(x)) \\
 &= \left[\sum_{(i,j) \in T_1} d_{ij}k_{ij}(x)(n_1 + n_2 - 1 - k_{ij}(x)) \right] \\
 &\quad + \left[\sum_{(i,j) \in T_2} d_{ij}k_{ij}(x)(n_1 + n_2 - 1 - k_{ij}(x)) \right]
 \end{aligned}$$

$$\begin{aligned}
 &= \left[\sum_{(i,j) \in T_1} d_{ij}k_{ij}(x)(n_1 - k_{ij}(x)) + (n_2 - 1) \sum_{(i,j) \in T_1} d_{ij}k_{ij}(x) \right] \\
 &\quad + \left[\sum_{(i,j) \in T_2} d_{ij}k_{ij}(x)(n_2 - k_{ij}(x)) + (n_1 - 1) \sum_{(i,j) \in T_2} d_{ij}k_{ij}(x) \right] \\
 &= [F(T_1) + (n_2 - 1)S_1(x)] + [F(T_2) + (n_1 - 1)S_2(x)].
 \end{aligned}$$

The term $[F(T_1) + (n_2 - 1)S_1(x)]$ can be interpreted as follows: $F(T_1)$ is the total cost incurred by the internal flow of the tree T_1 , while the term $(n_2 - 1)S_1(x)$ is the cost incurred by allowing the flow between the two subtrees to pass through tree T_1 . In addition, if y is a distinguished vertex in T_1 then:

$$S(y) = S_1(y) + S_2(x) + (n_2 - 1)L_{xy}.$$

Now, assume that (x, y) is an edge of the 2-tree G , G_i and G_j are two sides of (x, y) such that $G = G_i \cup G_j$. Furthermore, assume that $T = CT_i(x, y) \cup DT_j(x, y)$ is a spanning tree of G . Let $T_1 = CT_i(x, y)$ and $(T_2, T_3) = DT_j(x, y)$ where $V_1 \cap V_2 = \{x\}$ and $V_1 \cap V_3 = \{y\}$. Applying the above decomposition scheme to the two subtrees T_1 and T_2 in one step and the two subtrees $T_1 \cup T_2$ and T_3 in a second step, we get

$$\begin{aligned}
 F(T) &= [F(T_1) + (n_2 - 1)S_1(x) + (n_3 - 1)S_1(y) + (n_2 - 1)(n_3 - 1)L_{xy}] \\
 &\quad + [F(T_2) + (n - n_2)S_2(x) + F(T_3) + (n - n_3)S_3(y)].
 \end{aligned}$$

The terms between the first pair of square brackets can be viewed as a function of the subtree T_1 and the two numbers n_2 and n_3 . The terms between the second pair of square brackets can be viewed as a function of the forest (T_2, T_3) . We henceforth denote these two functions by $FCT(G_i, T_1, x; n_2, y; n_3)$ and $FDT(G_j, T_2, x; n_2, T_3, y; n_3)$, respectively, and refer to them as the FCT and the FDT selection functions. The FCT notation specifies the selection function of a spanning tree T_1 of the graph G_i (so $n_1 = n_i$) with two vertices labelled x and y when connected to a spanning forest of the subgraph induced by the vertices $V - V_i + \{x, y\}$ such that one tree in this forest contains n_2 vertices and is attached to vertex x , while the second tree contains n_3 vertices and is attached to vertex y . The FDT notation specifies the selection function of a spanning forest of the subgraph G_j containing two vertices x and y , subtree T_2 includes the vertex x and it contains n_2 vertices and subtree T_3 includes the vertex y and it contains n_3 vertices.

Several notes are in order. First, this notation encodes more information than is actually required since n_1, n_2, n_3 satisfy $n_1 + n_2 + n_3 - 3 = n$ in the FCT function, while $n_2 + n_3 = n_j$ in the FDT function. Note also that because G_i and G_j intersects at the two vertices x and y , we have $n_i + n_j - 2 = n$. Thus we can omit either n_2 or n_3 and substitute for it the mark $\#$. This alternative notation is used to make the equations more readable.

We use the notation $CT_{opt}(G_i, x; \alpha, y; \#)$ and $DT_{opt}(G_j, x; \alpha, y; \#)$ to refer to a spanning tree of type $CT_i(x, y)$ and a spanning forest of type $DT_i(x, y)$ of the subgraph G_i that minimizes the corresponding FCT and FDT functions respectively among all possible spanning trees and forests of G_i .

Moreover, the parameters

$$(F(T_1), S_1(x), S_1(y), L_{xy}, n_1) \quad \text{and} \quad (F(T_2), F(T_3), S_2(x), S_3(y), n_2, n_3))$$

are all we need to know about the subtree T_1 and the forest (T_2, T_3) to compute its selection function. Hence we call them the *summary parameters*. In the next section

we state the necessary equations to compute these parameters recursively for the generated trees and forests. It should be pointed out that maintaining such sets of parameters contributes to the efficiency of computing the selection functions.

The idea of the algorithm is to summarize a subgraph G_i by the following set of CT spanning trees

$$\{T: T = CT_{opt}(G_i, x: \alpha, y: \#), 1 \leq \alpha < n - n_i + 1\}$$

and the following set of DT spanning forests

$$\{H: H = DT_{opt}(G_i, x: \alpha, y: \#), 1 \leq \alpha < n_i - 1\}.$$

5. The algorithm. Let (x, y, z) be a triangle in a 2-tree G , where z is a degree two vertex, G_{xy} , G_{xz} and G_{yz} are three edge disjoint sides of the three edges (x, y) , (x, z) and (y, z) respectively, their union is the subgraph G_{xyz} . We first develop a few lemmas, and subsequently modify the prototype introduced in § 3 to generate the required forests.

LEMMA 1.

$$CT_{opt}(G_{xz} \cup G_{yz}, x: \alpha, y: \beta) = CT_{opt}(G_{xz}, x: \alpha, z: \#) \cup CT_{opt}(G_{yz}, y: \beta, z: \#).$$

Proof. Let T be a spanning tree of the subgraph $G_{xz} \cup G_{yz}$. T is the union of two subtrees T_1 of G_{xz} and T_2 of G_{yz} . It is required to show that $T = CT_{opt}(G_{xz} \cup G_{yz}, x: \alpha, y: \beta)$ if and only if $T_1 = CT_{opt}(G_{xz}, x: \alpha, z: \#)$ and $T_2 = CT_{opt}(G_{yz}, y: \beta, z: \#)$.

Since $T = T_1 \cup T_2$ we can deduce the summary parameters for T in terms of the summary parameters for T_1 and T_2 as follows.

$$(1.1) \quad F(T) = F(T_1) + F(T_2) + (n_2 - 1)S_1(z) + (n_1 - 1)S_2(z),$$

$$(1.2) \quad S(x) = S_1(x) + S_2(z) + (n_2 - 1)L_{xz},$$

$$(1.3) \quad S(y) = S_1(z) + S_2(y) + (n_1 - 1)L_{yz},$$

$$(1.4) \quad L_{xy} = L_{xz} + L_{yz},$$

$$(1.5) \quad n = n_1 + n_2 - 1.$$

We want to minimize:

$$(1.6) \quad \begin{aligned} \text{FCT}(G_{xz} \cup G_{yz}, T, x: \alpha, y: \beta) &= F(T) + (\alpha - 1)S(x) + (\beta - 1)S(y) \\ &\quad + (\alpha - 1)(\beta - 1)L_{xy}. \end{aligned}$$

Substituting for $F(T)$, $S(x)$ and L_{xy} in (1.6) we get:

$$\begin{aligned} \text{FCT}(G_{xz} \cup G_{yz}, T, x: \alpha, y: \beta) &= [F(T_1) + (n_2 + \beta - 2)S_1(z) + (\alpha - 1)S_1(x) + (\alpha - 1)(n_2 + \beta - 2)L_{xz}] \\ &\quad + [F(T_2) + (n_1 + \alpha - 2)S_2(z) + (\beta - 1)S_2(y) + (\beta - 1)(n_1 + \alpha - 2)L_{yz}] \end{aligned}$$

and the requirement of the proof follows immediately. \square

LEMMA 2.

$$\begin{aligned} DT_{opt}(G_{xz} \cup G_{yz}, x: \alpha, y: \beta) &= \begin{cases} CT_{opt}(G_{xz}, x: \#, z: (\alpha - n_{xz} + 1)) \cup DT_{opt}(G_{yz}, y: \beta, z: \#), & \alpha \geq n_{xz}, \\ DT_{opt}(G_{xz}, x: \alpha, z: \#) \cup CT_{opt}(G_{yz}, y: \#, z: (n_{xz} - \alpha)), & \alpha < n_{xz}. \end{cases} \end{aligned}$$

Proof. We prove the lemma for the case where $\alpha \geq n_{xz}$, the other case is similar. since $\alpha \geq n_{xz}$ then a DT spanning forest of the subgraph $G_{xz} \cup G_{yz}$ is a union of two subgraphs: a spanning tree T_1 of G_{xz} and a spanning forest (T_2, T_3) of G_{yz} where $z \in V_2$ and $y \in V_3$. Note that in the forest $(T_1 \cup T_2, T_3)$ we have $\beta = n_3$ and $a = n_1 + n_2 - 1$. It is required to prove that $(T_1 \cup T_2, T_3) = DT_{opt}(G_{xz} \cup G_{yz}, x: \alpha, y: \beta)$ if and only if $T_1 = CT_{opt}(G_{xz}, x: \#, z: (\alpha - n_{xz} + 1))$ and $(T_2, T_3) = DT_{opt}(G_{yz}, y: \beta, z: \#)$.

We first define the summary parameters of the subtree $T_1 \cup T_2$ in terms of the summary parameters of the subtrees T_1 and T_2 as follows.

$$(2.1) \quad F(T_1 \cup T_2) = F(T_1) + F(T_2) + (n_2 - 1)S_1(z) + (n_1 - 1)S_2(z),$$

$$(2.2) \quad S_{1,2}(x) = S_1(x) + S_2(z) + (n_2 - 1)L_{xz},$$

$$(2.3) \quad n_{1,2} = n_1 + n_2 - 1.$$

We want to minimize:

$$(2.4) \quad \begin{aligned} &FDT(G_{xz} \cup G_{yz}, T_1 \cup T_2, x: \alpha, T_3, y: \beta) \\ &= F(T_1 \cup T_2) + F(T_3) + (n - (n_1 + n_2 - 1))S_{1,2}(x) + (n - n_3)S_3(y). \end{aligned}$$

Substituting for $F(T_1 \cup T_2)$, $S_{1,2}(x)$ in (2.4) we get

$$\begin{aligned} &FDT(G_{xz} \cup G_{yz}, T_1 \cup T_2, x: \alpha, T_3, y: \beta) \\ &= [F(T_1) + (n_2 - 1)S_1(z) + (n - (n_1 + n_2 - 1))S_1(x) + (n_2 - 1)(n - (n_1 + n_2 - 1))L_{xz}] \\ &\quad + [F(T_2) + F(T_3) + (n - n_3)S_3(y) + (n - n_2)S_2(z)] \\ &= [F(T_1) + (\alpha - n_{xz})S_1(z) + (n - \alpha)S_1(x) + (\alpha - n_{xz})(n - \alpha)L_{xz}] \\ &\quad + [F(T_2) + F(T_3) + (n - \beta)S_3(y) + (n - (\alpha - n_{xz} + 1))S_2(z)] \end{aligned}$$

and the requirement of the proof follows immediately. \square

Lemmas 1 and 2 provide the basic tools required to modify Step 3.2 of the procedure introduced in § 3 to generate the required set of forests. The modification required is now summarized.

3.2. Merge the four sets associated with the edges (x, z) and (y, z) into temporary sets as follows.

3.2.1. Compute the following numbers.

$in(xz, yz) \leftarrow$ the number of vertices
that are included in the two subgraphs
 G_{xz} and G_{yz} , i.e., $(n_{xz} + n_{yz} - 1)$.

$out(xz, yz) \leftarrow$ the number of vertices
that are not included in the two subgraphs
 G_{xz} and G_{yz} except the two vertices x and y , i.e.,
 $(n - in(xz, yz) + 2)$.

3.2.1. For $\alpha \leftarrow 1, 2, \dots, out(xz, yz) - 1$,

$$\begin{aligned} &S_CT_{temp}(x, y) \leftarrow S_CT_{temp}(x, y) \\ &\cup \{CT_{opt}(G_{xz}, x: \alpha, z: \#) \cup CT_{opt}(G_{yz}, y: \#, z: n_{xz} + \alpha - 1)\} \end{aligned}$$

3.2.2. For $\alpha \leftarrow 1, 2, \dots$, in $(xz, yz) - 1$,

$$S_DT_{\text{temp}}(x, y) \leftarrow S_DT_{\text{temp}}(x, y) \cup \begin{cases} CT_{\text{opt}}(G_{xz}, x: \#, z: (\alpha - n_{xz} + 1)) \\ \cup DT_{\text{opt}}(G_{yz}, y: \#, z: (\alpha - n_{xz} + 1)), & \alpha \geq n_{xz}, \\ DT_{\text{opt}}(G_{xz}, x: \alpha, z: \#) \cup CT_{\text{opt}}(G_{yz}, y: \#, z: (n_{xz} - \alpha)), & \alpha < n_{xz}. \end{cases}$$

LEMMA 3. Let $\text{in}(xz, yz)$ be the number of vertices in the subgraph $G_{xz} \cup G_{yz}$, i.e., $\text{in}(xz, yz) = n_{xz} + n_{yz} - 1$. Then

$$CT_{\text{opt}}(G_{xyz}, x: \alpha, y: \beta) \in \{CT_{\text{opt}}(G_{xy}, x: \alpha + \text{index} - 1, y: \#) \cup DT_{\text{opt}}(G_{xz} \cup G_{yz}, x: \text{index}, y: \#): \text{index} = 1, 2, \dots, \text{in}(xz, yz) - 1\} \cup \{DT_{\text{opt}}(G_{xy}, x: \text{index}, y: \#) \cup CT_{\text{opt}}(G_{xz} \cup G_{yz}, x: \alpha + \text{index} - 1, y: \#): \text{index} = 1, 2, \dots, n_{xy} - 1\}.$$

Proof. Let T be a spanning tree of the subgraph G_{xyz} . We prove the lemma in the case where T is the union of a spanning tree T_1 of G_{zy} and a spanning forest (T_2, T_3) of $G_{xz} \cup G_{yz}$, where $\{x\} \in V_2$ and $\{y\} \in V_3$. The remaining case is similar.

It is sufficient to show that

$$(3.1) \quad \text{FCT}(G_{xyz}, T, x: \alpha, y: \beta) = \text{FCT}(G_{xy}, T_1, x: \alpha + n_2 - 1, y: \#) \cup \text{FDT}(G_{xz} \cup G_{yz}, T_2, x: n_2, T_3, y: \#)$$

and the proof follows by considering all possible values of n_2 .

Since $T = T_1 \cup T_2 \cup T_3$ we can write the parameters of T in terms of the parameters of the three subtrees T_1, T_2 and T_3 as follows.

$$(3.2) \quad F(T) = [F(T_1) + (n_2 - 1)S_1(x) + (n_3 - 1)S_1(y) + (n_2 - 1)(n_3 - 1)L_{xy}] + [F(T_2) + F(T_3) + (n_1 + n_3 - 2)S_2(x) + (n_1 + n_2 - 2)S_3(y)],$$

$$(3.3) \quad S(x) = S_1(x) + S_2(x) + S_3(y) + (n_3 - 1)L_{xy},$$

$$(3.4) \quad S(y) = S_1(y) + S_2(x) + S_3(y) + (n_2 - 1)L_{xy},$$

$$(3.5) \quad n = n_1 + n_2 + n_3 - 2.$$

Substituting for $F(T), S(x)$ and $S(y)$ in (3.1) we get

$$\begin{aligned} &\text{FCT}(G_{xyz}, T, x: \alpha, y: \beta) \\ &= F(T) + (\alpha - 1)S(x) + (\beta - 1)S(y) + (\alpha - 1)(\beta - 1)L_{xy} \\ &= [F(T_1) + (\alpha + n_2 - 2)S_1(x) + (\beta + n_3 - 2)S_1(y) + (\alpha + n_2 - 2)(\beta + n_3 - 2)L_{xy}] \\ &\quad + [F(T_2) + (n_1 + n_3 + \alpha + \beta - 4)S_2(x)] \\ &\quad + [F(T_3) + (n_1 + n_2 + \alpha + \beta - 4)S_3(y)], \end{aligned}$$

and (3.1) follows immediately. \square

Following the same strategy of the proof one can verify the following lemma.

LEMMA 4.

$$DT_{\text{opt}}(G_{xyz}, x: \alpha, y: \#) = \{DT_{\text{opt}}(G_{xy}, x: \text{index}, y: \#) \cup DT_{\text{opt}}(G_{xz} \cup G_{yz}, x: \alpha - \text{index} + 1, y: \#): \text{index} = 1, 2, \dots, \min(n_{xy} - 1, \alpha)\}.$$

Lemmas 4 and 5 provide the basic tools for modifying Step 3.3 of the prototype to generate the required forests as follows.

3.3. Update the set of forests associated with the edge (x, y) . Note that all spanning trees and forests of the subgraph $G_{xz} \cup G_{yz}$ have been previously generated in Step 3.2 and are stored in the two sets $S_CT_{temp}(x, y)$ and $S_DT_{temp}(x, y)$.

3.3.1. Compute the following numbers:

in $(xy, xz, yz) \leftarrow$ the number of vertices
that are included in the three subgraphs
 G_{xy}, G_{xz} and G_{yz} i.e. $(n_{xy} + n_{xz} + n_{yz} - 3)$.

out $(xy, xz, yz) \leftarrow$ the number of vertices
that are not included in the three
subgraphs G_{xy}, G_{xz} and G_{yz} except
the two vertices x and y i.e.
 $(n - \text{in}(xy, xz, yz) + 2)$.

3.3.2. For $\alpha \leftarrow 1, 2, \dots, \text{out}(xy, xz, yz) - 1$,
 $S_CT(x, y) \leftarrow S_CT(x, y) \cup$ a minimum selection cost
tree among the trees in the set

$\{CT_{opt}(G_{xy}, x: \alpha + \text{index} - 1, y: \#)$
 $\cup DT_{opt}(G_{xz} \cup G_{yz}, x: \text{index}, y: \#): \text{index} = 1, 2, \dots, \text{in}(xz, yz) - 1\}$

$\cup \{DT_{opt}(G_{xy}, x: \text{index}, y: \#)$

$\cup CT_{opt}(G_{xz} \cup G_{yz}, x: \alpha + \text{index} - 1, y: \#):$

$\text{index} = 1, 2, \dots, n_{xy} - 1\}$.

3.3.3. For $\alpha \leftarrow 1, 2, \dots, \text{in}(xy, xz, yz) - 1$,

$S_DT(x, y) \leftarrow S_DT(x, y) \cup$ a minimum selection cost
forest among the forest in the set

$\{DT_{opt}(G_{xy}, x: \text{index}, y: \#)$

$\cup DT_{opt}(G_{xz} \cup G_{yz}, x: \alpha - \text{index} + 1, y: \#):$

$\text{index} = 1, 2, \dots, \min(n_{xy} - 1, \alpha)\}$.

LEMMA 5. For an n vertex 2-tree G , the optimum distance spanning tree can be computed in $O(n^3)$ time.

Proof. We derive the worst case time required by the algorithm using the following prototype.

1. Initialization: for every edge (x, y) in G , initialize the two sets $S_CT(x, y)$ and $S_DT(x, y)$. Every set contains one simple subgraph. Hence, the time required by this step is $O(n)$ since a 2-tree contains $2n - 3$ edges.
2. Form a queue of degree 2 vertices in G . This step requires time which is linear in n .
3. Repeat until the graph is reduced to a single edge. The following steps are repeated $O(n)$ times.
 - 3.1. Remove vertex z from the queue. Locate its neighbors x and y . This step requires $O(1)$ time.
 - 3.2. Merge the four sets associated with the edges (x, z) and (y, z) into temporary sets. This operation computes at most n subgraphs. Each subgraph is constructed by performing a union of two forests and computing their

associated summary lists. Performing a union of two forests, each forest containing at most 2 trees, may be accomplished by establishing at most four pointers to them, thus requiring $O(1)$ time. Computing their associated summary list requires $O(1)$ time as described in the proofs of Lemmas 1 to 4. Hence this step requires $O(n)$ time.

- 3.3. Update the set of forests associated with the edge (x, y) . Again, this step computes at most n subgraphs. The computation of any such subgraph requires generating $O(n)$ forests and selecting the one which minimizes the appropriate selection function among those generated forests. Hence, the time required by this step is $O(n^2)$.
- 3.4. Delete the vertex z . If either x or y has its degree decreased to 2 add it to the queue. This step requires $O(1)$ time.
4. Extract the optimum distance tree from the tree of pointers constructed throughout the algorithm. This step requires $O(n)$ time.

It follows that the total time required to find the optimum distance tree on 2-trees is $O(n^3)$ in the worst case. \square

THEOREM 6. *For an n -vertex, m -edge series-parallel network G , the optimum distance spanning tree can be computed in $O(n^3 + m)$ time.*

Proof. We have already established that a series-parallel network can be transformed to a 2-tree such that both networks have the same optimum distance trees in $O(n + m)$ time. This fact together with Lemma 5 establishes the required proof. \square

6. Conclusions. In this paper we presented a polynomial time algorithm to find the optimum distance spanning tree on series-parallel networks and 2-trees. We doubt that a similar scheme would result in a polynomial time algorithm for the general optimum communication spanning tree problem or even for the optimum requirement spanning tree problem. In such problems the size of the set of forests that should be maintained appears to grow exponentially with the size of the problem instance.

Another problem that extends the OCSTP is the *optimal network problem*, discussed by Scott [11] and Boyce et al. [2]. This problem allows different edge construction costs and asks for a network that minimizes the total routing costs subject to the added constraint that the total construction costs cannot exceed a given budget. However, this problem is NP-complete even if the requirements are all equal [9]. In this case the knapsack problem reduces to the problem on a series-parallel network. It is also interesting to note that a similar technique solves the optimum distance problem on k -trees for fixed k in polynomial time [4].

REFERENCES

- [1] J. A. BONDY AND U. S. R. MURTY, *Graph Theory with Applications*, Macmillan, New York, 1976.
- [2] D. E. BOYCE, A. FARHI AND R. WIESCHEDEL, *Optimal network problem: a branch-and-bound algorithm*, Environment and Planning, 5 (1973), pp. 519-533.
- [3] R. J. DUFFIN, *Topology of series-parallel networks*, J. Math. Anal. Appl., 10 (1965), pp. 303-318.
- [4] E. S. EL-MALLAH, *Recursive graph structure and the optimum communication spanning tree problem*, M.Sc. Thesis, Dept. Computational Science, Univ. of Saskatchewan, Canada, 1983.
- [5] A. M. FARLEY AND A. PROSKUROWSKI, *Computation of the center and diameter of outer-planar graphs*, Discr. Appl. Math., 2 (1980), pp. 185-191.
- [6] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1971.
- [7] T. C. HU AND R. E. GOMORY, *Multi-terminal network flows*, SIAM J. Appl. Math., 9 (1961), pp. 551-570.
- [8] T. C. HU, *Optimum Communication Spanning Trees*, this Journal, 3 (1974), pp. 188-195.

- [9] D. S. JOHNSON, J. K. LENSTRA AND A. H. G. RINNOOY KAN, *The complexity of the network design problem*, *Networks*, 8 (1978), pp. 279-285.
- [10] T. L. MAGNANTI AND R. T. WONG, *Network design and transportation planning: models and algorithms*, *Transportation Sci.*, 18 (1984), pp. 1-55.
- [11] A. J. SCOTT, *The optimal network problem: some computational procedures*, *Transportation Res.*, 3 (1969), pp. 201-210.
- [12] T. TAKAMIZAWA, T. NISHIZEKI AND N. SAITO, *Linear time computability of combinatorial problems on series-parallel graphs*, *J. ACM*, 29 (1982), pp. 623-641.
- [13] R. E. TARJAN, *Depth-first search and linear graph algorithms*, *this Journal*, 1 (1972), pp. 146-159.
- [14] J. A. WALD AND C. J. COLBOURN, *Steiner trees, partial 2-trees, and minimum IFI networks*, *Networks*, 13 (1983), pp. 159-167.

A LINEAR RECOGNITION ALGORITHM FOR COGRAPHS*

D. G. CORNEIL†, Y. PERL‡ AND L. K. STEWART†

Abstract. Cographs are the graphs formed from a single vertex under the closure of the operations of union and complement. Another characterization of cographs is that they are the undirected graphs with no induced paths on four vertices. Cographs arise naturally in such application areas as examination scheduling and automatic clustering of index terms. Furthermore, it is known that cographs have a unique tree representation called a cotree. Using the cotree it is possible to design very fast polynomial time algorithms for problems which are intractable for graphs in general. Such problems include chromatic number, clique determination, clustering, minimum weight domination, isomorphism, minimum fill-in and Hamiltonicity. In this paper we present a linear time algorithm for recognizing cographs and constructing their cotree representation.

Key words. cographs, permutation graphs, perfect graphs, linear algorithms, examination scheduling

1. Introduction. Following Cook's pioneering work in establishing the NP-completeness of the satisfiability problem [2], thousands of other problems were also shown to be NP-complete or NP-hard [6]. Since many practical problems are "intractable" in this sense, hopes for producing algorithms which would find optimal solutions in a reasonable amount of time had to be abandoned and instead attention was directed to the development and analysis of heuristic algorithms. It was soon realized that the very pessimistic worst-case analysis included in the NP-complete results did not accurately reflect the success which heuristic algorithms were achieving on real-world combinatorial optimization problems. As Karp noted "one of the great mysteries in the field of combinatorial algorithms is the baffling success of many heuristic algorithms" [9].

The search for a theoretical answer to this question leads to three main approaches. The first allows approximations to be made to the optimal solution. In a very few cases a constant bound on the ratio of the approximation to the optimal may be proven however in most cases such a bound is not known. Furthermore even if such a bound is known, in practice the observed behaviour of the algorithm is often much better than the bound. The second approach substitutes either expected case or average case analysis for worst-case analysis. Typically this type of research involves the probabilistic analysis of a particular heuristic algorithm. Often this is accomplished by analyzing the algorithm's average behaviour on some form of random input, for example random graphs. One shortcoming of this approach is that usually the graphs encountered in practice have some structure which violates the assumption that all edges have the same independent probability of being present. For this reason the algorithm's performance in a real-world environment often is not as good as predicted by the optimistic expected case or average case analysis.

The third approach, and the one we take in this paper, maintains the worst-case analysis and restricts the class of input to be considered. The hope here of course is that the intractable problem will be solvable in polynomial time on this restricted class of input and furthermore that membership in this restricted class can be recognized quickly. Examples of families of graphs which have received this type of study include

* Received by the editors August 16, 1983 and in final revised form July 27, 1984.

† Department of Computer Science, University of Toronto, Toronto, Ontario, M5S 1A1, Canada.

‡ Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08903, and Bar Ilan University, Ramat-Gan, Israel.

comparability graphs [11], permutation graphs [11], interval graphs [1], chordal graphs [10] and planar graphs [8]. The difficulty with this approach is that one can usually provide only an intuitive justification that the restricted class is a good model for the input expected in practice. Furthermore, even if the restricted class does meet this criterion, it may happen that a particular input is very close to being a member of the restricted class but in fact is not. The algorithms developed for such a class must be sufficiently robust to allow adaptation to these “near misses”.

In this paper, we are concerned with combinatorial optimization problems where the input graphs are expected to obey a certain “local density” property. In particular the application suggests that the graphs are unlikely to have more than a very few induced paths on four vertices. Thus if any four vertices a, b, c, d form a path then it is expected that at least one of (a, c) , (b, d) and (a, d) is also in the edge set. Applications where this structure is expected to occur include examination scheduling and automatic clustering of index terms. In examination scheduling, the vertices represent courses and the edge (i, j) signifies the existence of a student taking both courses i and j . The weight of the edge indicates the number of such students. In any colouring of the graph the colour classes represent courses whose examinations may be held concurrently. It is anticipated that such conflict graphs are very unlikely to have long induced paths. In the second application we want to generate clusters (subsets of vertices with a high density of edges) on a graph where edges represent the proximity or self-referencing of index terms. Again it is expected that if four index terms form a path, then at least one of the other edges will also be present (see [7]). As we shall see, the P_4 restricted graphs are precisely cographs.

Cographs (or *complement reducible graphs*) are defined as the class of graphs formed from a single vertex under the closure of the operations of union and complement. Cographs were independently discovered under various names and were shown by Lerchs (see [3]) to have the following two remarkable properties. First they are exactly the P_4 restricted graphs described above and secondly a cograph has a unique tree representation. This tree, called a *cotree*, forms the basis for fast polynomial time algorithms for problems such as isomorphism, colouring, clique detection, clusters, minimum weight dominating sets, minimum fill-in and Hamiltonicity [3], [4], [5].

Naturally before such algorithms can be employed it is necessary to have a fast cograph recognition algorithm which also produces the cotree on recognition of a cograph. In § 3 the outline of such a linear time algorithm is presented. This algorithm follows from a new theorem on the structure of cographs that forms the basis of a heuristic algorithm to deal with graphs which are almost cographs. Before discussing this material we present examples and properties of cographs.

2. Cographs. Cographs are perfect and in fact form a proper subset of the permutation graphs. Furthermore, cographs are precisely the underlying undirected graphs of transitive series-parallel digraphs [12]. Although there is a linear time algorithm for recognizing if a digraph is transitive series-parallel [13], this algorithm does not seem to be amenable to the cograph case. In fact, as noted by Spinrad “recognition of transitive series-parallel graphs seems easier than recognition of cographs” [11]. We will show that this is not the case.

An example of a cograph and its cotree is presented in Fig. 1. In the cotree representation leaves of the cotree represent vertices of the graph. Internal nodes of the cotree are labelled 0 or 1 in such a way that (0) nodes and (1) nodes alternate along every path starting from the root which is always a (1) node. The root will have only one (0) node child if and only if the represented cograph is disconnected. Two

vertices x and y of the graph are adjacent if and only if the unique path from x to the root of the tree meets the unique path from y to the root at a (1) node.

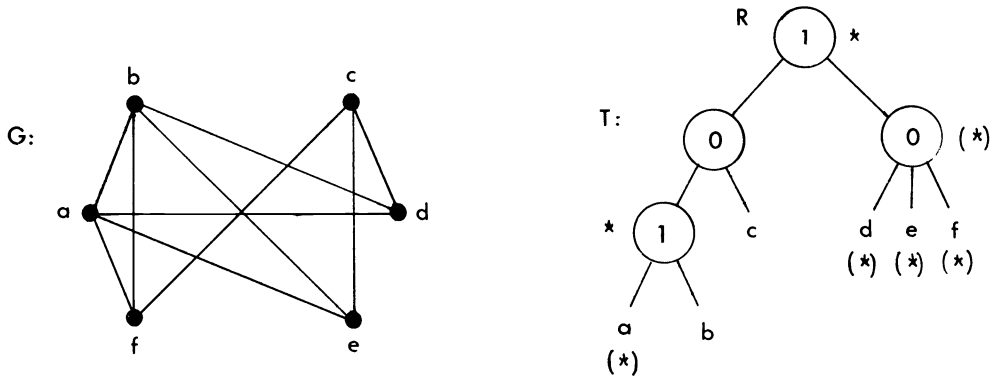


FIG. 1. A cograph G and its cotree T . The $*$ and $(*)$ symbols refer to procedure MARK in § 3.1.

Cograph algorithms typically associate operations with the (1) nodes and the (0) nodes of the cotree and assign weights or numbers to the leaves. The algorithm then uses the operations on the interior nodes to calculate particular values associated with the subgraph rooted at the interior node. The value assigned to the root is the value assigned to the cograph. For example, if the leaves are assigned the value $a = 1$ and the (1) nodes have $a = \prod_{i=1}^k a_i$ and the (0) nodes have $a = \sum_{i=1}^k a_i$ where a_1, \dots, a_k are the values associated with the children, then the a value of the root is the number of maximal complete subgraphs in the given cograph. For the example in Fig. 1, the number of such maximal complete subgraphs is 6 corresponding to the subgraphs $(abd), (abe), (abf), (cd), (ce), (cf)$.

3. Cograph recognition.

3.1. Theoretical foundations of the algorithm. In [12] an $O(n^2)$ algorithm is presented for the recognition of cographs and the construction of the corresponding cotrees. Both that algorithm and our algorithm are incremental in the sense that the vertices are processed one at a time until the entire graph has been handled. Since cographs are *hereditary* (i.e., every induced subgraph is a cograph) we assume we have a cograph G with cotree T and present an algorithm which determines if $G + x$ is also a cograph and, if so, modifies T to represent $G + x$. Given the list of the vertices of G adjacent to x we percolate this adjacency information up the tree from the leaves to the root using a simple marking scheme. Before presenting that algorithm we introduce the following notation. For a node w of T (rooted at R), $d(w)$ denotes the number of children of w in T and $md(w)$ is the current number of children of w which have been both “marked” and “unmarked”. For all nodes w , the value of $md(w)$ is initially 0 and is reset to 0 when w is “unmarked”. The following procedure uses the adjacency information of the new vertex x to “mark” and then “unmark”, where appropriate, the nodes in T .

MARK(x)

Mark all leaves of T which are adjacent to x
 For each marked node u of T with $d(u) = md(u)$
 do unmark (u);
 $md(u) \leftarrow 0$;

if $u \neq R$ then mark (w) where w is the parent of u ;
 $md(w) \leftarrow md(w) + 1$;
 insert u at the head of a linked list of marked
 and unmarked children of w

end

If any vertex remains marked and $d(R) = 1$ then mark R
 end MARK

Let M denote the set of internal nodes of T which remain marked after the procedure MARK has been performed, and let α be a node in M with lowest level in the tree and let β be a node in $M \setminus \{\alpha\}$ with lowest level. We say a marked (1) node γ is *properly marked* if and only if $md(\gamma) = d(\gamma) - 1$. A *legitimate alternating path* in a marked cotree is a path of adjacent alternating properly marked (1) nodes and unmarked (0) nodes, the extreme points of which are (1) nodes.

The procedure MARK is illustrated for adding a vertex x adjacent to the vertices a , d , e and f in the graph of Fig. 1. We associate * with marked nodes and (*) with nodes which are subsequently unmarked. See Fig. 1. In this example $M = \{R, \text{parent}(a)\}$, $\alpha = \text{parent}(a)$ and $\beta = R$.

We now show that the marked set M determines whether or not $G+x$ is a cograph. Later we use this theorem as the basis of our linear time cograph recognition algorithm.

THEOREM 1. *If G is a cograph with cotree T then $G+x$ is a cograph if and only if*

1. M is empty or
2. (i) $M \setminus \{\alpha\}$ consists of exactly the (1) nodes of a (possibly empty) legitimate alternating path which ends at R and
 (ii) α is either a (0) node whose parent is β or α is a (1) node whose grandparent, if it exists, is β .

Proof. Only if. If the conditions of the theorem do not hold then we have at least one of the following:

- (i) $M \setminus \{\alpha\}$ contains a (0) node.
- (ii) \exists a (1) node in $M \setminus \{\alpha\}$ which is not properly marked.
- (iii) $\exists \gamma \neq R$ in $M \setminus \{\alpha\}$ s.t. the grandparent of γ is not in $M \setminus \{\alpha\}$.
- (iv) The vertices of $M \setminus \{\alpha\}$ do not lie on one path to R .
- (v) α is a (0) node whose parent is not β .
- (vi) α is a (1) node which has a grandparent which is not β .

By definition, any vertex in M has been marked but not unmarked, and this implies there is at least one descendant leaf adjacent to x and at least one not adjacent to x . Using this fact, it is fairly straightforward to show that any of the above six conditions implies the existence of an induced P_4 in $G+x$. As an example, we demonstrate an induced P_4 in the graph $G+x$ if condition (i) is found to be true. The following notation is used: for any internal node θ of T , $\text{des}(\theta)$ denotes the set of descendant leaves of θ , that is, the leaves of the subtree of T rooted at θ . Let γ be a (0) node in $M \setminus \{\alpha\}$ and let δ be the lowest common ancestor of α and γ in T . Note the possibility that $\delta = \gamma$. There are four cases to be considered, depending on the labels (0) or (1) of α and δ .

Case 1. α and δ are both (0) nodes.

Proof. There is an induced P_4 on vertices (b, c, x, d) if x and c are adjacent in $G+x$, or on (b, c, a, x) if x and c are not adjacent, where: $a \in \text{des}(\alpha)$ and is adjacent to x ; $b \in \text{des}(\alpha)$ and is not adjacent to x ; $c \in \text{des}(\text{parent}(\alpha)) \setminus \text{des}(\alpha)$; $d \in \text{des}(\gamma)$ and is adjacent to x . If $\delta = \gamma$, we require that $d \in \text{des}(\gamma) \setminus \text{des}(\theta)$, where θ is the child of γ on the $\alpha - \gamma$ path.

Cases 2, 3 and 4 follow similarly.

If. We complete this part of the proof by constructing T' , the cotree representing $G+x$.

1. If M is empty, then x can be added as a child of the root if $G+x$ is connected, or as a child of the only child of the root in the case where $G+x$ and G are both disconnected. If $G+x$ is disconnected but G is connected the root of T and x both become children of the only child of a new root.

2. There is a lowest marked node $\alpha \in M$. Let A be the children of α which were marked and subsequently unmarked by procedure MARK. Similarly, let B be the children of α which were not marked by MARK. The fact that $\alpha \in M$ implies that $|A| \geq 1$ and $|B| \geq 1$. To construct T' there are two cases to consider.

Case (i). α is a (0) node.

In this case, the elements of A and B are either leaves or (1) nodes.

If $|A| = 1$ and $a \in A$ is a leaf then we add a new (1) node in place of a and make a and x children of this node. If $|A| = 1$ and $a \in A$ is a (1) node then we simply add x as a new child of a .

If $|A| > 1$ then we remove all elements of A from α and add a new (1) node in their place. Children of this new node are x and a new (0) node with elements of A as children.

Case (ii). α is a (1) node.

The proof follows exactly as in case (i), except that B is examined instead of A , and the roles of (0) nodes and (1) nodes are reversed.

To see that T' is an accurate representation of $G+x$, we observe that the alterations to T correctly reflect adjacencies of x with vertices in the subtree rooted at α , and the fact that we have a legitimate alternating path from α to R guarantees that all other adjacencies of x are correctly represented. Adjacencies among vertices of G remain unchanged as required. \square

3.2. The cograph recognition algorithm. As stated previously, the algorithm for recognizing cographs and constructing their cotrees is an incremental one. We begin with the cotree for two vertices and incorporate the remaining vertices into the tree one by one. Each iteration essentially consists of an efficient implementation of the preceding theorem. The complete algorithm follows.

ALGORITHM COGRAPH-RECOGNITION. Given a graph $G=(V, E)$ with vertices arbitrarily indexed v_1, \dots, v_n , this algorithm determines whether or not G is a cograph and constructs G 's cotree T if G is a cograph. We assume that *md* is set to zero for all new nodes including leaves as they are added to T .

1. (Initialize.)
 Create a new (1) node R
If $(v_1, v_2) \in E$
 then add v_1, v_2 as children of R
 else create a new (0) node N ;
 add N as a child of R ; add v_1 and v_2 as children of N
2. (Iteratively incorporate v_3, \dots, v_n into T)
For $x \leftarrow v_3, \dots, v_n$ *do*:
 2.1. Call procedure MARK (x)
 2.2. *If* all nodes of T were marked and unmarked
 then add x as a child of R ;
 goto endloop

```

2.3. If no nodes of  $T$  were marked
    then if  $d(R) = 1$ 
        then add  $x$  as a child of the only child of  $R$ 
        else create a new (1) node  $R$  with one child (a new (0) node)
            and two grandchildren:  $x$  and the old root;
        goto endloop
2.4.  $u \leftarrow \text{FIND-LOWEST}$ 
2.5. Let  $A(B)$  denote the set of children of  $u$  which were (were not) marked
    if  $\text{label}(u) = 0 (=1)$ 
        then if  $|A| = 1$  ( $|B| = 1$ )
            then if  $w \in A$  ( $\in B$ ) is a leaf
                then add a new (1) node ((0) node) in place of  $w$ 
                    and make  $w$  and  $x$  children of this node
                else add  $x$  as a new child of  $w$ 
            else remove all elements of  $A$  from  $u$  and add them as children
                of a new node  $y$  with  $\text{label}(y) = \text{label}(u)$ 
                if  $u$  is a (0) node
                    then add a new (1) node as a child of  $u$ ; children of
                        this new (1) node are  $x$  and  $y$ 
                    else remove  $u$  from its parent and add  $y$  in its place;
                        add a new (0) node as a child of  $y$ ; children of
                            this new (0) node are  $x$  and  $u$ 
        endloop
    end COGRAPH-RECOGNITION

```

Function FIND-LOWEST. This function checks whether $G+x$ is a cograph and, if so, returns u , the lowest marked vertex of T . To form the cotree for $G+x$, x must then be added to the subtree of T rooted at u . The following notation is used: u is the lowest marked vertex so far examined; w denotes the lowest marked (1) node examined before u ; y is a marked (1) node which is not properly marked or a marked (0) node, if either exists in T . Whenever the procedure finds that $G+x$ is not a cograph, an accompanying comment indicates which of the conditions (i)–(vi) from the proof of the theorem holds. When this occurs, we assume the entire algorithm is terminated.

1. (Initialize and check root.)


```

 $y \leftarrow \Lambda$ 
If  $R$  is not marked
then  $G+x$  is not a cograph /* condition (iii)
else do if  $md(R) \neq d(R) - 1$ 
    then  $y \leftarrow R$ 
    unmark  $R$ ;  $md(R) \leftarrow 0$ ;
     $u \leftarrow w \leftarrow R$ 
    end

```
2. (Choose an arbitrary marked vertex u and follow the path from u to w , checking for a legitimate alternating path and unmarking vertices along the path.)


```

while there are marked vertices remaining in  $T$ 
do choose an arbitrary marked vertex  $u$ 
    2.1. if  $y \neq \Lambda$ 
        then  $G+x$  is not a cograph /* condition (i) or (ii)
        if  $\text{label}(u) = 1$ 

```

```

then do if  $md(u) \neq d(u) - 1$ 
    then  $y \leftarrow u$ 
    if parent ( $u$ ) is marked
    then  $G+x$  is not a cograph /* conditions (i) and (vi)
    else  $t \leftarrow$  parent (parent ( $u$ ))
    end
else do  $y \leftarrow u$ ;
     $t \leftarrow$  parent ( $u$ )
    end
unmark  $u$ ;  $md(u) \leftarrow 0$ 
2.2. (Now check if the  $u-w$  path is part of the legitimate alternating path
 $u-R$ .)
while  $t \neq w$ 
do
    if  $t = R$ 
    then  $G+x$  is not a cograph /* condition (iv)
    if  $t$  is not marked
    then  $G+x$  is not a cograph /* condition (iii) or (v) or (vi)
    if  $md(t) \neq d(t) - 1$ 
    then  $G+x$  is not a cograph /* condition (ii)
    if parent ( $t$ ) is marked
    then  $G+x$  is not a cograph /* condition (i)
    unmark  $t$ ;  $md(t) \leftarrow 0$ ;
     $t \leftarrow$  parent (parent ( $t$ ))
    end
2.3. (Reset  $w$  for next choice of marked vertex.)
 $w \leftarrow u$ 
end (step 2)
end FIND-LOWEST

```

3.3. Example of the algorithm. To illustrate the algorithm, assume we have the cograph G and cotree T of Fig. 1 and consider the graph $G+x$ where x is adjacent to a, d, e and f . When procedure MARK terminates, the following vertices have been marked: $a, d, e, f, \text{parent}(a), \text{parent}(d), R$; and the following have been unmarked: a, d, e, f and $\text{parent}(d)$; leaving $\text{parent}(a)$ and R marked. See Fig. 1. Function FIND-LOWEST unmarks R and sets $u \leftarrow w \leftarrow R$ in step 1. Thus in step 2 there is only one remaining marked vertex to examine: $u \leftarrow \text{parent}(a)$. Since u is a (1) node with $md(u) = d(u) - 1$ and $\text{parent}(u)$ is not marked, the only actions taken in step 2.1 are $t \leftarrow R$ and unmark ($\text{parent}(a)$). The loop of step 2.2 is never executed because $t = w = R$. In step 2.3 we set $w \leftarrow \text{parent}(a)$ in preparation for the next step 2 loop, but this loop is not repeated in this case since no marked vertices remain. The FIND-LOWEST function terminates with $u = \text{parent}(a)$, indicating that $G+x$ is a cograph. Back in step 2.5 of the main procedure, we identify $A = \{a\}$ and $B = \{b\}$. Since u is a (1) node, $|B| = 1$ and $b \in B$ is a leaf, we need only add a new (0) node in place of b and make b and x children of this node. The resulting cotree T' representing cograph $G+x$ is shown in Fig. 2.

Let us also consider the graph G' in Fig. 2 and add vertex z where z is adjacent to a, c, d, e and f . Procedure MARK will mark the following: $a, c, d, e, f, \text{parent}(a), \text{parent}(c), \text{parent}(d), R$; and unmark a, c, d, e, f and $\text{parent}(d)$; leaving $\text{parent}(a), \text{parent}(c)$ and R marked. In FIND-LOWEST we unmark R and set $u \leftarrow w \leftarrow R$ in step

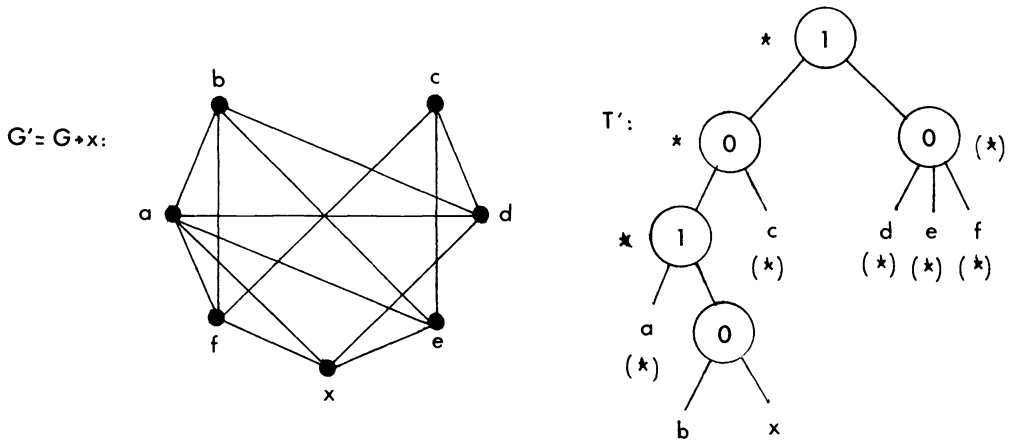


FIG. 2. Cograph $G' = G + x$ and its cotree T' .

1. Assume $u \leftarrow \text{parent}(c)$ is chosen in step 2. As label $(u) = 0$, we end step 2.1 with $y = u$, $t = R$ and parent (c) unmarked. Step 2.2 is not executed because $w = t = R$ and in 2.3 we set $w \leftarrow \text{parent}(c)$. Now $u \leftarrow \text{parent}(a)$ and we see in 2.1 that $G' + z$ is not a cograph because $y \neq \Lambda$, indicating in this case that $M \setminus \{\alpha\}$ contains a (0) node (condition (i) of the proof). If $u \leftarrow \text{parent}(a)$ is chosen first in step 2, we find that $G' + z$ is not a cograph in step 2.1, where we find that label $(u) = 1$ and parent (u) is marked. Fig. 3 shows the graph $G' + z$ with a $P_4(b, a, z, c)$ indicated.

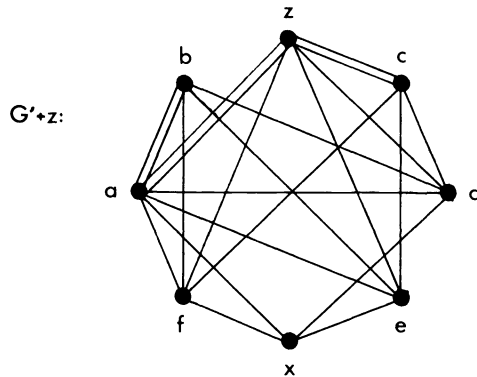


FIG. 3. The graph $G' + z$ is not a cograph—vertices b, a, z, c form a P_4 .

3.4. Timing analysis. The timing analysis for the algorithm relies on a time bound of $O(\text{deg}(x))$ for the iteration adding x to T , where $\text{deg}(x)$ is the degree of x in $G + x$.

Since all internal nodes of T , except possibly the root, have at least two children, we know that the MARK procedure will examine only $O(\text{deg}(x))$ nodes. For each of these nodes, the processing is done in constant time, and thus the time bound for procedure MARK is $O(\text{deg}(x))$. It is clear that $|M|$ is also bounded by $O(\text{deg}(x))$, and since FIND-LOWEST examines each marked node once in constant time, the time for this function is $O(|M|) = O(\text{deg}(x))$.

All but one of the tree alterations can be done in constant time. The only cases which may require more than constant time are those where the lowest marked node is a (0) node ((1) node) which has two or more children which have been marked and unmarked (not been marked). In both cases, we are careful to move the children which

were both marked and unmarked, since the cardinality of this set is $O(\deg(x))$ whereas the cardinality of the set of children which were not marked is not similarly bounded. In procedure MARK we have maintained a linked list of the children which were marked and subsequently unmarked, and hence, they can be accessed in time bounded by $O(\deg(x))$. Therefore, all of the tree modifications can be done in $O(\deg(x))$ time.

Thus, we have the required bound for each iteration, implying an overall time bound of $O(m+n)$ for the entire algorithm.

4. Concluding remarks. As noted in § 1, in most typical applications, the graphs encountered may not be cographs but in fact will be very close to being a cograph. It may be necessary to add or delete a few edges in order to destroy all P_4 s and thus to achieve a cograph. For this purpose one would want to find as large a subcograph as possible. Not surprisingly this problem is intractable, however using Theorem 1 it is possible to get a good heuristic algorithm. We first note that applying the recognition algorithm to the graph and rejecting any vertex whose addition does not yield a cograph does result in a subcograph. This procedure has two drawbacks. First, it greatly depends on the order in which the vertices are presented. One would expect that a non-increasing degree order would help in obtaining a large subcograph. Secondly, some particularly bad vertex may be included in the subcograph and thereafter cause many other vertices to be rejected. Obviously, a complete backtracking scheme is out of the question; however using Theorem 1 it is possible to develop a limited backtracking procedure. Under this scheme any time a vertex is rejected, a note is made of the existing vertices which cause the rejection. Once an existing vertex has accumulated enough such notes it is removed and other rejected vertices are tried again.

Acknowledgments. The authors wish to thank the Natural Sciences and Engineering Research Council of Canada for financial assistance.

REFERENCES

- [1] K. S. BOOTH AND G. S. LUEKER, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms*, J. Comput. System Sci., 13 (1976), pp. 335-379.
- [2] S. A. COOK, *The complexity of theorem-proving procedures*, Proc. 3rd Annual ACM Symposium on Theory of Computing, ACM, New York, 1971, pp. 151-158.
- [3] D. G. CORNEIL, H. LERCHS AND L. STEWART BURLINGHAM, *Complement reducible graphs*, Disc. Appl. Math., 3 (1981), pp. 163-174.
- [4] D. G. CORNEIL AND Y. PERL, *Clustering and domination in perfect graphs*, Disc. Appl. Math., 9 (1984), pp. 27-39.
- [5] D. G. CORNEIL, Y. PERL AND L. K. STEWART, *Cographs: recognition, applications and algorithms*, Proc. 15th Southeastern Conference on Combinatorics, Graph Theory and Computing, LSU, 1984, to appear.
- [6] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman, San Francisco, 1979.
- [7] C. C. GOTLIEB AND S. KUMAR, *Semantic clustering of index terms*, J. Assoc. Comput. Mach., 15 (1968), pp. 493-513.
- [8] J. E. HOPCROFT AND R. E. TARJAN, *Efficient planarity testing*, J. Assoc. Comput. Mach., 21 (1974), pp. 549-568.
- [9] R. M. KARP, *On the complexity of combinatorial problems*, Networks, 5 (1975), pp. 45-68.
- [10] D. J. ROSE, R. E. TARJAN AND G. S. LUEKER, *Algorithmic aspects of vertex elimination graphs*, this Journal, 5 (1976), pp. 266-283.
- [11] J. SPINRAD, *Transitive orientation in $O(n^2)$ time*, Proc. 15th Annual ACM Symposium on Theory of Computing, ACM, New York, 1983, pp. 457-466.
- [12] L. STEWART, *Cographs, a class of tree representable graphs*, TR 126/78, Dept. Computer Science, Univ. Toronto, Toronto, Ontario, 1978.
- [13] J. VALDES, R. E. TARJAN AND E. L. LAWLER, *The recognition of series parallel digraphs*, this Journal, 11, 2 (1982), pp. 298-313.

EQUATIONS BETWEEN REGULAR TERMS AND AN APPLICATION TO PROCESS LOGIC*

ROHIT PARIKH†, ASHOK CHANDRA‡, JOE HALPERN§ AND ALBERT MEYER¶

Abstract. Regular terms with the Kleene operations \cup , $;$ and $*$ can be thought of as operators on languages, generating other languages. An equation $\tau_1 = \tau_2$ between two such terms is said to be *satisfiable* just in case languages exist which make this equation true. We show that the satisfiability problem even for $*$ -free regular terms is undecidable. Similar techniques are used to show that a very natural extension of the Process Logic of Harel, Kozen and Parikh is undecidable.

Key words. regular sets, context free languages, logic of programs

1. Introduction. Process Logic is an outgrowth of Propositional Dynamic Logic (PDL), introduced by Pratt in [Pr] and is essentially an attempt to combine the best features of both PDL and Temporal Logic (TL). Briefly, the difference between PDL and TL is as follows. In PDL, we consider simultaneously the behavior of several programs, and a formula of PDL depends for its truth or falsity on all its component programs. However, PDL talks only about the “before-after” behavior of programs, and program behavior during a computation cannot be referred to by PDL in a direct way. Sometimes it *can* be done in an *indirect* way, since the behavior of a program α during a computation of α can often be reduced to the before-after behavior of some component program β of α . TL on the other hand looks at a single computation of a single program in a detailed way and has constructs which allow us to state what happens during the computation. However, TL normally talks only about *one* program at a time. Thus expressing the interaction of several programs becomes cumbersome in TL, though it *can* be achieved, for example by introducing “ At ” predicates which refer to the labels of statements. In process logic we have program constructs as in PDL *as well as* temporal constructs as in TL. Thus process logic becomes a general purpose, flexible framework for talking about programs at the propositional level.

In [Pa1] there was introduced a very powerful version of process logic called SOAPL. This logic was shown to include both PDL and TL, and it was shown in [H2] that SOAPL was properly stronger than Pratt’s version of process logic in [Pr]. SOAPL was shown to be decidable by reducing it to SnS [R]. However, SOAPL had restrictions on formula formation which were essential to proving decidability, and this made the language unclear to many people. A simplified language, closely related to SOAPL was introduced by Nishimura in [Ni] and a further simplification was given in [HKP]

* Received by the editors August 25, 1982, and in final revised form July 3, 1984. An earlier version of this paper was published in the Thirteenth Annual ACM Symposium on the Theory of Computing, Milwaukee, Wisconsin, May 1981, Copyright 1981, Association for Computing Machinery, Inc. This research was supported in part by the National Science Foundation under grants MCS80-10707, MCS79-10261 and the Natural Sciences and Engineering Research Council of Canada.

† Mathematics Department, Boston University, Boston, Massachusetts 02215 and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. Present address Department of Computer and Information Science, Brooklyn College of the City University of New York, Brooklyn, New York 11210.

‡ IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

§ Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139 and Mathematics Department, Harvard University, Cambridge, Massachusetts 02138. Present address, IBM Research Center, San Jose, California 95193.

¶ Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

where also were proved a completeness result and a decidability result. We now proceed to give a brief description of the [HKP] language and semantics. The operator construct "chop" was not included in the [HKP] version of process logic, but we have included it here. This extension leaves the system decidable provided that no *other* changes are made at the same time.

- (i) P_1, \dots, P_n are *atomic formulae*.
- (ii) a_1, \dots, a_m are *atomic programs*.
- (iii) If α, β are programs, so are $\alpha; \beta, \alpha \cup \beta, \alpha^*$.
- (iv) If A, B are formulae, so are $\neg A, A \vee B$.
- (v) If A is a formula and α is a program, then $\langle \alpha \rangle A$ is a formula.
- (vi) If A, B are formulae, so are $fA, A \text{ suf } B$ and $A \text{ chop } B$.

In (iii) above, we have described program formation under the Kleene connectives. (iv), (v) and (vi) describe, respectively, formula formation under Boolean connectives, program modalities and temporal connectives. Dropping (ii), (iii) and (v) would give us TL and dropping (vi) would give us PDL. We shall use $[\alpha]A$ as an abbreviation for $\neg \langle \alpha \rangle \neg A$. Similarly we shall use $A \vee B$ as an abbreviation for $\neg((\neg A) \vee (\neg B))$.

A *model* of process logic is a triple $M = (W, \rho, R)$. W is a set of *states*. A *path* is a finite or infinite sequence of states, and ρ and R are assignments of sets of paths to the atomic formulae and programs respectively. States will be denoted by the letters s, t, \dots , and paths will be denoted by p, q, \dots . If $p = (s_0, \dots, s_k)$ is a path and $q = (t_0, t_1, \dots)$ is a path, then pq , the *fusion product* of p and q is defined if and only if $s_k = t_0$, and equals $(s_0, \dots, s_k, t_1, \dots)$. The path q need not be finite for pq to be defined, but p must be.

The *length* $l(p)$ of a path $p = (s_0, \dots, s_k)$ is k . A state can be thought of as a path of length 0. $Fp = (s_0)$ is the first state of p . If $p = qr$ and neither q nor r has length 0, then q is a *prefix* of p and r is a *suffix* of p . Thus p is not its own prefix or suffix.

We now define the *semantics* of process logic. First we extend R to a mapping on all programs inductively via:

$$R_{\alpha;\beta} = \{pq \mid p \in R_\alpha \text{ and } q \in R_\beta\},$$

$$R_{\alpha \cup \beta} = R_\alpha \cup R_\beta,$$

$$R_{\alpha^*} = \bigcup (R_{(\alpha^n)}); n \geq 0.$$

We are now ready to define what it means for a path p in M to satisfy a formula A :

- (i) $p \models P_i$ iff $p \in \rho(P_i)$;
- (ii) $p \models \neg A$ iff $p \not\models A$. Similarly for \vee ;
- (iii) $p \models \langle \alpha \rangle A$ iff $\exists q \in R_\alpha$ such that pq is defined and $pq \models A$;
- (iv) $p \models fA$ iff $Fp \models A$ if and only if the first state of p satisfies A ;
- (v) $p \models A \text{ suf } B$ iff $\exists q$ suffix of p such that $q \models B$ and for all suffixes r of p such that q is a suffix of r , $r \models A$;
- (vi) $p \models A \text{ chop } B$ iff $\exists q, r$ such that $p = qr$ and $q \models A$ and $r \models B$.

The following notions can be defined from the ones given above. See [HKP]. In (viii) $p = (s_0, \dots, s_k)$, for some k . Of course, p may be infinite in (ix) and (x) and so s_k might not exist.

- (vii) $p \models Ln_0$ iff $k = 0$; $p \models Ln_1$ iff $k = 1$.
- (viii) $p \models \text{last } A$ iff $(s_k) \models A$.
- (ix) $p \models nA$ iff $(s_1, \dots, s_k) \models A$.
- (x) $p \models A \text{ chop } B$ iff $p \models A \text{ chop } (nB)$; i.e., $\exists j < k$ such that $(s_0, \dots, s_j) \models A$ and $(s_{j+1}, \dots, s_k) \models B$.

The difference between chop and chomp is that in chomp there is no overlap between the pieces into which p is split. In chop the last state of one is the first state of the other.

We will say that a formula A is *local* in a model M if for all paths p , $p \models A$ iff $p \models fA$. Equivalently, every p satisfies $A \Leftrightarrow fA$. Thus a local formula depends only on the *first* state of the path. A model M is local if and only if the basic predicates P_i are local. *Local process logic* is process logic with the semantics restricted to local models. The following theorem is due to Dexter Kozen and is stated here with his permission.

THEOREM. *The validity problem for formulae of local process logic is decidable but not elementary.*

TL with chop can also be shown to be nonelementary by the same sort of argument which involves the coding of Turing machine computations.

The obvious question is what happens if we drop this locality condition. Note that both PDL and TL are embeddable in local process logic. A related question is the following. Note that both programs and predicates have the same semantics, namely states of paths. Do we really need two kinds of objects? What happens, for instance, if we allow the construct Mem, whose semantics is defined by:

$$“(xi) p \models \text{Mem}(\alpha) \text{ iff } p \in R_\alpha” \quad ?$$

We shall show that in either case process logic becomes undecidable. This fact may be thought of as a confirmation of the intuition that even though both programs and properties are represented in process logic as sets of computations, they are very different kinds of animals. A program *produces* computations, a property *evaluates* them. This is why it does not make any sense to talk of the complement of a program, though it is perfectly natural to talk of the negation of a property.

2. Equations between regular terms. Suppose we consider regular terms τ formed from symbols a, b, c, \dots and variables X, Y, Z, \dots by means of the Kleene operations \cup and $;$. The operation $*$ is not needed for the undecidability result. Let Σ be some finite alphabet. Then we can interpret these terms as *operators* which take subsets of Σ^* (languages) as inputs and provide a new subset of Σ^* as output. For example, if the term τ is $(X; a) \cup Y$ and the input languages are L and L' respectively, then the output language $\tau(L, L')$ will be $(La) \cup L'$. (We shall suppress $;$ for the sake of readability.)

DEFINITION 1. An equation $\tau(X_1, \dots, X_k) = \tau'(X_1, \dots, X_k)$ is said to be *satisfiable* over Σ if there exist languages L_1, \dots, L_k over Σ such that

$$\tau(L_1, \dots, L_k) = \tau'(L_1, \dots, L_k).$$

THEOREM 1. *The satisfiability problem for *-free regular terms in the sense above is undecidable.*

The proof of this theorem depends on two auxiliary lemmas. We remark in passing that the validity problem is decidable as the reader can easily verify by applying König's lemma.

LEMMA 1. *Given a context-free language $L \subseteq \Sigma^*$, there exists a finite, satisfiable system Φ of *-free regular equations such that for any alphabet Γ with $\Sigma \subseteq \Gamma$, and for any solution $L_1, \dots, L_k \subseteq \Gamma^*$, L_1 is L .*

Proof. We first assume that $L = L(G)$, where G is a grammar in Greibach normal form, i.e., all the productions of G are of the form $N \rightarrow x$, where x is a string whose

leftmost symbol is a terminal. As an illustration, we take G to be the grammar:

$$\begin{aligned} S &\rightarrow bDE, \\ D &\rightarrow cEc|b, \\ E &\rightarrow bDbEc|c. \end{aligned}$$

Corresponding to G we have the system of equations:

$$\begin{aligned} X_1 &= bX_2X_3, \\ X_2 &= cX_3c \cup b, \\ X_3 &= bX_2bX_3c \cup c. \end{aligned}$$

It is easy to check that $L_1 = L(S)$, $L_2 = L(D)$, $L_3 = L(E)$ is a solution to these equations. Moreover, this solution is unique. For suppose (M_1, M_2, M_3) is another solution, $M_1, M_2, M_3 \subseteq \Gamma^*$. Then we can show by induction on the length of $w \in \Gamma^*$ that $w \in M_1$ (resp. M_2, M_3) iff $w \in L_1$ (resp. L_2, L_3). For example, if $w \in M_1$ and w is of length k , then $w \in bM_2M_3$. Thus there are strings $u \in M_2$ and $v \in M_3$ of length $< k$ such that $w = buv$. By the induction hypothesis $u \in L_1$ and $v \in L_2$, and since (L_1, L_2, L_3) is a solution of Φ , it must be the case that $w \in L_1$. The converse is similar. Q.E.D.

It should be clear that this technique works for an arbitrary grammar H in Greibach normal form. Note that the fact that H is in Greibach normal form assures us that any string of length > 1 can be written as a product of shorter strings to which the induction hypothesis applies.

For the general case, it is well known that given any grammar H we can effectively find a grammar H' in Greibach normal form such that $L(H)$ is $L(H')$ or $L(H') \cup \varepsilon$, depending on whether or not the empty string ε is in $L(H)$ (see, for example, [H3] for a proof). In the latter case we can apply the above techniques to H' to get a system of equations $\Phi_{H'}$. Then we adjoin to the system the equation $X_0 = X_1 \cup \{\varepsilon\}$. If (L_0, L_1, \dots, L_k) is a solution to this new system of equations, then by the above $L_1 = L(H')$, so $L_0 = L(H)$. Q.E.D.

Note that in Lemma 1 the alphabet Γ for the solution is allowed to be strictly greater than the alphabet Σ in which the equations are stated. This is necessary since the proof of Lemma 2 involves the addition of extra symbols and we have to make sure that the uniqueness property is not lost thereby.

LEMMA 2. *Given a finite system of regular equations as in Lemma 1, it can be reduced to a single equation.*

Proof. Let the system Φ be $\tau_1 = \tau'_1, \dots, \tau_m = \tau'_m$. We introduce new symbols b_1, \dots, b_m and consider the equation $(b_1\tau_1) \cup \dots \cup (b_m\tau_m) = (b_1\tau'_1) \cup \dots \cup (b_m\tau'_m)$. It is straightforward to check that a system of languages L_1, \dots, L_k satisfies the system Φ if and only if it satisfies the single equation.

We now prove the main theorem by using the fact that the equality problem for context-free languages is undecidable [H3]. For let $L(H)$ and $L(H')$ be two context-free languages whose equality we wish to decide. Let $\Phi_H, \Phi_{H'}$ be systems of defining equations for H and H' as in Lemma 1. Rename the variables of $\Phi_H, \Phi_{H'}$, so that X_1 is the variable corresponding to H , Y_1 is the variable corresponding to H' and $\Phi_H, \Phi_{H'}$ have no variables in common. Now take the system $\phi_H \cup \phi_{H'} \cup (X_1 = Y_1)$ and use Lemma 2 to compress it to a single equation. It is easy to see that this equation is satisfiable if and only if the unique solutions of $\Phi_H, \Phi_{H'}$ are equal if and only if $L(H) = L(H')$. Q.E.D.

Remark. The theorem also holds if we restrict ourselves to terms τ containing a single variable X_1 . However, we shall need the $*$ operation in that case. We briefly sketch the details.

We reduce the Post correspondence problem (PCP) to a regular equation involving only one variable. Suppose we are given two sets of strings in $\{0, 1\}^+$, say $\{w_1, \dots, w_k\}$ and $\{u_1, \dots, u_k\}$, with $w_i \neq u_i$, (where in general we use Σ^+ to denote $\Sigma \cdot \Sigma^*$).

Let

$$S = \{(x, y) \mid x = w_{j_1} \dots w_{j_m}, y = u_{j_1} \dots u_{j_m}, m \geq 1, j_i \leq k\},$$

and let

$$S_2 = \{(x, y) \mid x = w_{j_1} \dots w_{j_m}, y = u_{j_1} \dots u_{j_m}, m \geq 2, j_i \leq k\}.$$

It is well known (see [H3]) that the question of whether there is a string z such that $(z, z) \in S$ (or equivalently, $(z, z) \in S_2$) is undecidable.

Notation. For $w \in \{0, 1\}^*$, let w^r = the reverse of w . Let $0' = 0b$, $1' = 1b$, $0'' = b0$ and $1'' = b1$. We extend ' and '' to homomorphisms on all of $\{0, 1\}^*$. Let $(w')^-$ denote w' without the trailing b ; i.e., $w' = (w')^-b$. Similarly, let $(w'')^-$ denote w'' without the leading b ; i.e., $w'' = b(w'')^-$. Let $\Sigma = \{0, 1, b, c\}$. Note that for all $u \in \{0, 1\}^+$, $(u')''$ is the same as $(u')^r$.

We define languages R, M, M_1, P and Q as follows:

$$R = \{\text{strings in } \Sigma^* \text{ which have an occurrence of } 00, 01, 10, 11, cc, 0c, c0, 1c, c1 \text{ or } bb\} \\ = \Sigma^*(00 \cup 01 \cup 10 \cup 11 \cup cc \cup 0c \cup c0 \cup 1c \cup c1 \cup bb)\Sigma^*,$$

$$M = \{x'c(y')'' \mid x, y \in \{0, 1\}^*, x \neq y\},$$

$$M_1 = 0'(0' \cup 1')^*c(0'' \cup 1'')^*1'' \cup 1'(0' \cup 1')^*c(0'' \cup 1'')^*0'' \\ \cup (0' \cup 1')^+c \cup c(0'' \cup 1'')^+,$$

$$P = \{w'x'c(y')''(w'')'' \mid (x, y) \in S_2, w \in \{0, 1\}^*\},$$

$$Q = \{bx'c(y')''b \mid (x, y) \in S\}.$$

If the given PCP has no solution, then $P \subseteq M$. On the other hand, if there is a solution $(z, z) \in S_2$, then $z'c(z'')'' \in P - (R \cup M \cup Q)$. Hence $R \cup M \cup Q = R \cup M \cup P \cup Q$ if and only if the PCP has no solutions.

Using methods similar to those in Lemma 1, it is straightforward to check that the equation (A) below has the unique solution $R \cup M \cup Q$, while (B) has the unique solution $R \cup M \cup P \cup Q$:

$$(A) \quad X = R \cup M_1 \cup 0'X0'' \cup 1'X1'' \cup b(w_i')^-X((u_i')'')^-b \\ \cup bw_i'c(u_i')''b \quad (\text{for each } i \leq k);$$

$$X = R \cup M_1 \cup 0'X0'' \cup 1'X1'' \\ (B) \quad \cup b(w_i')^-X((u_i')'')^-b \cup bw_i'c(u_i')''b \\ \cup (w_i')^-X((u_i')'')^- \quad (\text{for each } i \leq k).$$

(In fact, any equation which is of the form $X = \tau(X)$, where $\tau(X)$ is in "Greibach normal form" can be shown to have a unique solution.)

Thus, if (A) is denoted $X = \tau_1(X)$ and (B) is denoted $X = \tau_2(X)$, the final equation is:

$$dX \cup eX = d\tau_1(X) \cup e\tau_2(X).$$

This equation has a solution ($= R \cup M \cup Q$) if and only if the Post correspondence problem has no solution.

We note that we can reduce the total alphabet from $\{0, 1, b, c, d, e\}$ to just $\{0, 1\}$ by appropriately encoding each symbol into 3 bits. Q.E.D.

We now raise the question of what the solutions of these equations look like in case there are solutions.

THEOREM 2. *If a system Φ of regular equations has a unique solution, i.e., if all the languages corresponding to the variables are uniquely determined by the equations, then that solution is recursive. If there is a—not necessarily unique—solution, then at least one solution is recursive in the halting problem.*

Proof. Let L be a language. Then given a number k , we will denote by $L(k)$ the set of all strings in L of length $\leq k$. Given two regular expressions τ_1 and τ_2 as in the previous theorems, and a set of languages L_1, \dots, L_m , we will say that L_1, \dots, L_m are k -consistent (with $\tau_1 = \tau_2$) if and only if finite sets $\tau_1(L_1(k), \dots, L_m(k))(k)$ and $\tau_2(L_1(k), \dots, L_m(k))(k)$ are equal. In other words we get the same result if we chop both before and after plugging into the τ_i . Clearly (L_1, \dots, L_m) is a solution of the equation $\tau_1 = \tau_2$ if and only if it is k -consistent with it for all k . This is because strings of length $\leq k$ in $\tau(L_1, \dots, L_m)$ can only be produced by strings in the L_i of length $\leq k$. A tuple of the form $(L_1(k), \dots, L_m(k))$ will be called a k -piece.

Suppose now that the equation has a unique solution (L_1, \dots, L_m) . Then we will show that we can effectively construct the k -pieces $(L_1(k), \dots, L_m(k))$ of the solution. Let us say that a proposed k -piece is consistent if and only if it is k -consistent. Suppose now that a given k -piece is in fact the correct one. Then it has n -consistent extensions for all $n > k$. Conversely, suppose some k -piece has n -consistent extensions for all $n > k$. Then the tree of its consistent extensions is infinite, and hence, by König's lemma, has an infinite branch, which must be correct, so that the k -piece was correct. Thus a piece is incorrect if and only if the tree of consistent extensions is finite and a piece is correct if and only if the finitely many alternatives of the same length are all incorrect. This gives us a decision procedure for correctness of k -pieces, and hence a decision procedure for a string to be in the solution: A string is in L_i if and only if it is in the i th place of some correct k -piece.

This argument works if the solution is unique. If the solution is not unique, then the set of incorrect pieces is still r.e., though it may no longer be recursive. Hence the set of correct pieces is co-r.e. Since every correct piece has a correct extension, a solution that is recursive in the halting problem exists. For instance, the leftmost solution will be recursive in the halting problem, where "leftmost" is in the tree of k -pieces. Q.E.D.

An interesting application of the results of this section is as follows. Consider two flowcharts made up of some (unspecified) atomic actions and tests, as well as some *black boxes* representing some unknown subroutines. Now consider the question, whether it is possible to find actual (not necessarily regular) subroutines to put in these black boxes so that the resulting flowcharts are equivalent over all interpretations. This question is easily seen to be another form of the problem of Theorem 1 and is therefore undecidable.

3. Undecidability in process logic. We now show that nonlocal process logic as described above is undecidable. This result does not need the full power of the construct *suf*. All we need are *n*, last and chop.

THEOREM 3. *The satisfiability (and hence the validity) problem for nonlocal process logic is undecidable.*

Proof. We proceed much as in Theorem 1. We first show that a context free grammar in Greibach normal form can be “coded up” in process logic. Again we use the grammar G of Lemma 1 for illustration.

We define the formula A_G to be the conjunction of Ln_0 (defined in (vii) in § 1) and the following three formulas (where P, D, E and S are atomic formulae, B abbreviates $P \wedge Ln_0$ and C abbreviates $\neg P \wedge Ln_0$):

$$\begin{aligned} A_1: & [\alpha^*](\text{last}([\alpha]Ln_1)); \\ A_2: & [\alpha^*](\text{last}(\langle \alpha \rangle \text{last}(B) \wedge \langle \alpha \rangle \text{last}(C))); \\ A_3: & [\alpha^*](\text{last}([\alpha^*](S \Leftrightarrow \text{chomp}(B, D, E) \\ & \quad \wedge D \Leftrightarrow (\text{chomp}(C, E, C) \vee B) \\ & \quad \wedge E \Leftrightarrow (\text{chomp}(B, D, B, E, C) \vee C))) \\ & \text{(where chomp}(B, D, E) \text{ is an abbreviation for} \\ & \text{chomp}(B, \text{chomp}(D, E)), \text{ etc.).} \end{aligned}$$

We now explain the intuitive meaning of the formulae A_1, \dots, A_3 . Suppose $M = (W, \rho, R)$ is a model such that for some state s in M we have $(s) \models A_G$. We can assume without loss of generality that every state of W is accessible from s by α^* . (Otherwise we can *restrict* M to this set of states.) A_1 forces all the α paths in M to be of length one. A_2 implies that for any $t \in W$, there exist states u_1 and u_2 in W reachable from t by doing one α such that $(u_1) \models P$ (or equivalently, $(u_1) \models B$) and $(u_2) \models \neg P$.

Let b, c be two symbols. We define the map $\sigma: W \rightarrow \{b, c\}$ via $\sigma(s) = b$ if $(s) \models P$ and $\sigma(s) = c$ if $(s) \models \neg P$. By extending σ to be a homomorphism we can associate with every finite path in M a *unique* string in $\{b, c\}^*$. For example, if $p = (s_0, s_1, s_2)$, where $s_0 \models P, s_1 \models P$ and $s_2 \models P$, then $\sigma(p)$ is $bc b$. Then using the fact that $(s) \models A_3$, we can show that for all paths p in M, p is in $\rho(S)$ (resp. $\rho(D), \rho(E)$) if and only if $\sigma(p) \in L(S)$ (resp. $L(D), L(E)$). The proof is by induction on the length of p , and is similar to the proof of the analogous statement in Lemma 1. We leave the details to the reader. Again it is easy to see that this technique works for arbitrary grammars in Greibach normal form.

Now we can conclude the proof of Theorem 3. Given two grammars H and H' in Greibach normal form, consider the formula $A_H \wedge A_{H'}$. Apart from the common start symbol S, H and H' are assumed to have distinct nonterminals. A_H forces $\rho(S)$ to look like $L(H)$, while $A_{H'}$ forces $\rho(S)$ to look like $L(H')$. Hence the conjunction is satisfiable if and only if $L(H) = L(H')$. Since the equality problem for languages not containing the empty string is undecidable, the satisfiability problem for process logic is also undecidable. Q.E.D.

THEOREM 4. (*Local*) *process logic without the construct chop but with the operation Mem is undecidable.*

The proof of this theorem is quite similar in style to that of Theorem 3. Instead of using primitive propositions to do our encoding, we do it all by means of programs (which is why the question of locality does not arise here). The terminal symbols are represented by primitive program symbols which are associated only with paths of length 1; the nonterminals are represented by other program symbols. Chomp is replaced by a combination of Mem and ;. For example, the grammar G of Lemma 1 is encoded as follows:

The programs β and γ encode the terminals b and c . We force them to have the right properties by the formula A_1 :

$$[(\beta \cup \gamma)^*](\text{last}([\beta]L_1 \wedge [\gamma]L_1 \wedge (\beta) \text{true} \wedge (\gamma) \text{true})).$$

Then we let the programs α , d and η represent the nonterminals S , D and E respectively. The formula A_2 (which corresponds to A_3 in the previous proof) is:

$$\begin{aligned} & [(\beta \cup \gamma)^*](\text{last}([\beta \cup \gamma]^*(\text{mem}(\alpha) \Leftrightarrow \text{mem}(\delta; \eta; \beta) \\ & \quad \wedge \text{mem}(\delta) \Leftrightarrow (\text{mem}(\gamma; \eta; \gamma) \vee \text{mem}(\beta)) \\ & \quad \wedge \text{mem}(\eta) \Leftrightarrow (\text{mem}(\beta; \delta; \beta; \eta; \gamma) \vee \text{mem}(\gamma)))). \end{aligned}$$

As above, if $(s) \models A_1 \wedge A_2$, we can assume without loss of generality that the only states in M are those reachable from s by $(\beta \cup \gamma)^*$. Moreover we can assume (by adding extra copies of states if necessary) that if $(s, t) \in \rho(\beta)$ (resp. $\rho(\gamma)$), then $(s, t) \notin \rho(\gamma)$ (resp. $\rho(\beta)$). Thus to each path p in M we can associate a unique string $\sigma'(p)$ in $\{b, c\}^*$. Then as above we can show that $p \in \rho(\alpha)$ (resp. $\rho(\delta)$, $\rho(\eta)$) if and only if $\sigma'(p) \in L(S)$ (resp. $L(D)$, $L(E)$). The rest of the proof proceeds just as in Theorem 3. Q.E.D.

Conclusion. We have shown that even a slight extension of process logic as defined in [HKP] is undecidable. Thus the language as defined there (with the inclusion of chop, but retaining locality) is as rich as we can hope to have. It is open if local process logic with *suf* and *f*, but without chop is nonelementary.

Postscript. Bob Streett has recently shown that nonlocal process logic with *suf* and *f* is also undecidable. Moreover, David Harel has pointed out to us that using the methods of [HPS], nonlocal process logic with *f* and chop can be shown to be Π_1^1 complete. See also [HMM] for a program free version. We thank Richard Ladner for pointing out an error in our original version of the remark following Theorem 1.

REFERENCES

- [GPSS] G. GABBAY, A. PNUELI, S. SHELAH AND J. STAVI, *The temporal analysis of fairness*, 7th Annual ACM Symposium on Principles of Programming Languages, Jan., 1980, pp. 163–173.
- [H1] J. HALPERN, *Deterministic process logic is elementary*, Proc. 23rd IEEE Symposium on the Foundations of Computer Science, pp. 204–216; also Inform. Control, 57 (1984), pp. 56–89.
- [H2] D. HAREL, *Two results on process logic*, Inform. Proc. Lett., 8 (1979), pp. 195–198.
- [H3] M. HARRISON, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA, 1978.
- [HKP] D. HAREL, D. KOZEN AND R. PARIKH, *Process logic: expressiveness, decidability, completeness*, Proc. 21st IEEE Symposium on the Foundations of Computer Science, pp. 129–142; also in J. Comput. System Sci., 25 (1982), pp. 144–170.
- [HMM] J. HALPERN, Z. MANNA AND B. MOSZKOWSKI, *A hardware semantics based on temporal intervals*, Proc. 10th ICALP, 1983, pp. 278–291.
- [HPS] D. HAREL, A. PNUELI AND J. STAVI, *Propositional dynamic logic on context-free programs*, Proc. 22nd IEEE Symposium on the Foundations of Computer Science, pp. 310–321.
- [M] A. MEYER, *WSIS is not elementary decidable*, Logic Colloquium, R. Parikh, ed., Lecture Notes in Mathematics 453, Springer, New York, 1974, pp. 132–154.
- [Ni] H. NISHIMURA, *Descriptively complete process logic*, Acta Informatica, 14 (1980), pp. 359–369.
- [Pa1] R. PARIKH, *Second order process logic*, 19th IEEE Symposium on Foundations of Computer Science, Oct., 1978, pp. 173–183.
- [Pa2] ———, *Propositional dynamic logics of programs: a survey*, in Logics of Programs, E. Engeler, ed., Lecture Notes in Computer Science 125, Springer, New York, 1981, pp. 102–144.
- [Pr] V. R. PRATT, *Process Logic*, Proc. 6th Annual ACM Symposium on Principles of Programming Languages, Jan., 1979, pp. 93–100.
- [R] M. RABIN, *Decidability of second order theories and automata on infinite trees*, Trans. Amer. Math. Soc., 141 (1969), pp. 1–35.

THE PURE LITERAL RULE AND POLYNOMIAL AVERAGE TIME*

PAUL WALTON PURDOM, JR.† AND CYNTHIA A. BROWN‡

Abstract. For a simple parameterized model of conjunctive normal form predicates, we show that a simplified version of the Davis–Putnam procedure can, for many values of the parameters, solve the satisfiability problem in polynomial average time. Let v be the number of variables, $t(v)$ the number of clauses in a predicate, and $p(v)$ the probability that a given literal appears in a clause ($p(v)$ is the same for all literals). Let ε be any small positive constant and n any large positive integer. Then a version of the Davis–Putnam procedure that uses only backtracking and the pure literal rule uses average time that is polynomial in the problem size when any of the following conditions are true for large v . (1) $t(v) \leq n \ln v$; (2) $t(v) \geq \exp(\varepsilon v)$; (3) $p(v) \geq \varepsilon$; or (4) $p(v) \leq n(\ln v/v)^{3/2}$. Until recently the best previous bounds for cases (1) and (4) were $t(v) \leq (\ln \ln v)/(\ln 3)$ and $p(v) \leq \exp(-v/\ln \ln v)$. These results show that the problem types for which the pure literal rule works well are quite different from those for which backtracking works well. Our present knowledge suggests this random problem sets with $t(v)$ somewhat larger than v and with $p(v)$ somewhat larger than v^{-1} are particularly difficult to solve.

Key words. backtracking, pure literal rule, average time, searching

1. Introduction. Many important and interesting sets of problems are NP-complete [7]. For such problem sets every algorithm that has been analyzed has a worst case time that is an exponential function of the problem size. There are, however, interesting random NP-complete problem sets that can be solved in polynomial *average* time.

The set of satisfiability problems (determining whether an arbitrary predicate in conjunctive normal form (CNF) can be satisfied) is a fundamental NP-complete problem set. Goldberg [8], [9], [10] showed that satisfiability for random CNF predicates with t clauses and probability p that a given literal appears in a clause (p is a constant that is the same for all literals) can be solved in polynomial average time using the pure literal rule (plus splitting) from the Davis–Putnam procedure [3], [4]. Using a different model of random CNF problems, Brown and Purdom [1] showed that for some problem sets backtracking takes average time $\exp O(v^{3/4})$ for problems with v variables, which is a great improvement over the $\exp O(v)$ time required for exhaustive search. However, the type of problems for which backtracking is fast is quite different from the type for which the pure literal rule does best.

This difference in performance led us to investigate the average time for several satisfiability algorithms on a wide range of random problem sets. We used a model of random CNF predicates similar to Goldberg's, parameterized by using arbitrary functions $t(v)$ ($t(v) \geq 1$) for the number of clauses and $p(v)$ ($0 \leq p(v) \leq 1$) for the probability that a literal appears in a clause. We found that by using three simple algorithms random CNF problems can be solved in time polynomial in the problem size whenever any of the following four conditions is satisfied [16]; (1) $t(v) \leq (\ln \ln v)/\ln 3$, (2) $t(v) \geq \exp(\varepsilon v)$ (trivial because the problem has exponential size), (3) $p(v) \geq \varepsilon$, or (4) $p(v) \leq \exp(-v/(\ln \ln v))$ (ε is any small constant). These results show that random CNF problems can be solved in polynomial average time whenever either $t(v)$ or $p(v)$ is extremely large or extremely small.

In this paper, using an improved analysis of the pure literal rule, we show that random CNF problems can be solved in polynomial average time whenever (1)

* Received by the editors July 14, 1982, and in final revised form July 3, 1984. The research reported here was supported in part by the National Science Foundation under grant MCS 7906110.

† Computer Science Department, Indiana University, Bloomington, Indiana 47405.

‡ College of Computer Science, Northeastern University, Boston, Massachusetts 02115.

$t(v) \leq n \ln v$ or (2) $p(v) \leq n(\ln v/v)^{3/2}$, where n is any large positive constant. These limits depend on using only two algorithms, the pure literal rule and backtracking. Figure 1 summarizes our results. The random problem sets corresponding to points away from the center can be solved in polynomial average time. In the center there is a *hard region* where backtracking is known to require exponential average time and where other algorithms are likely to require exponential average time. For random problem sets corresponding to points outside the contour surrounding the hard region, it has been proven that the problems can be solved in polynomial average time. See § 3 for more discussion of the figure.

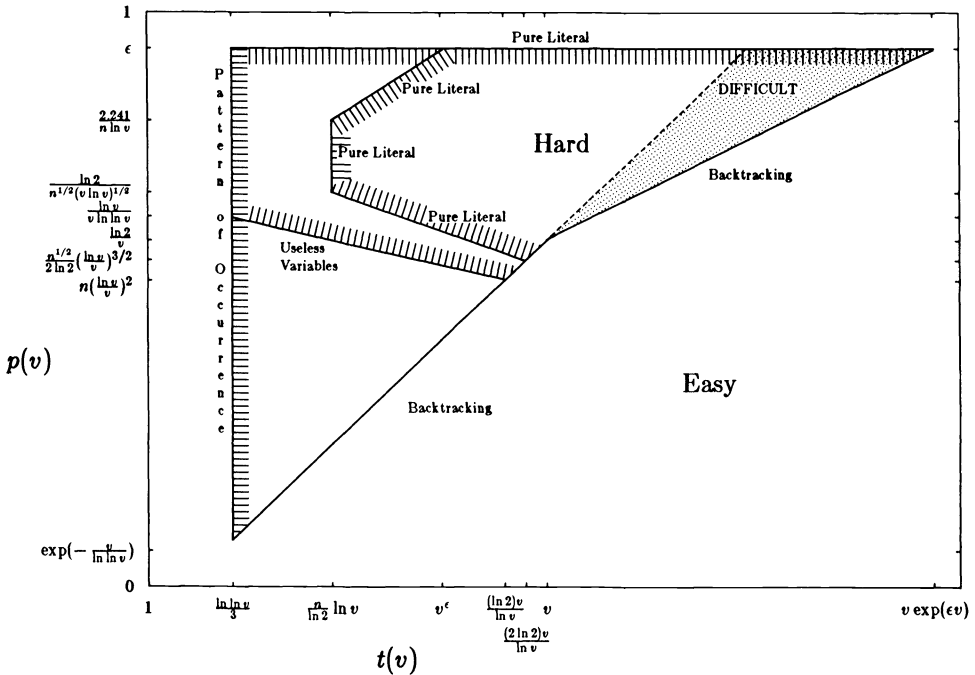


FIG. 1. A diagram showing the regions of $[p(v), t(v)]$ space where random CNF predicates can be solved in polynomial average time. This diagram shows both the results of this paper and of several previous papers. The various boundaries are labeled with the algorithm for producing polynomial average time in that region of the diagram. Along the backtracking boundary, exponential average time is required just inside the boundary, so the boundary is marked with a solid line. All the other boundaries result from upper bound analyses, so they are marked with hashed lines. Outside of the regions marked with hashed lines, random problems can be solved in polynomial average time. Inside such boundaries it is not yet known whether polynomial or exponential average time is required. The dashed line separates the region where the expected number of solutions per problem is exponentially large (upper left side) from where it is exponentially small (lower right). The results given in this paper are that the Davis-Putnam procedure can solve problems outside the polygon in polynomial average time. (The time that it requires inside the polygon has not yet been determined.) The lower right boundary results from an analysis of backtracking. The top, upper left, left, and lower left boundaries come from the analysis of the pure literal rule. See the text and references for the functional form of the boundaries. (The label $1/(n \ln v)$ on the right side is a correction to previously published versions of this figure.)

The running time of a satisfiability algorithm is essentially determined by the number of binary nodes in the search tree it generates. It is clear that the full Davis-Putnam procedure runs at least as fast (to within a factor of v^2 to allow for possible extra overhead) as either the pure literal rule or backtracking, as long as all three algorithms use the same order of selecting variables (whenever the pure literal

rule or the unit clause rule [3], [4] does not force the order of selection, which happens only at unary nodes). This is because the binary nodes of the search tree for the full Davis–Putnam procedure are a subset of the binary nodes for the trees for each of the other algorithms.

We have identified random sets of CNF problems for which a quick (polynomial average time) solution method is known. It is likely that other NP-complete problems sets with similar characteristics can also be solved quickly using similar techniques. Our present knowledge suggests that random CNF problems with $t(v)$ somewhat larger than v and with $p(v)$ somewhat larger than v^{-1} are particularly difficult to solve.

2. Random CNF predicates. To form a random problem set, let v be the number of variables; the variables can take on the values *true* and *false*. There are then $2v$ possible literals, one for each variable and its negation. A random clause is formed by independently selecting each literal with probability $p(v)$, and a random predicate is formed by independently selecting $t(v)$ random clauses. For each variable, the probability that it and its negation both occur in a clause is $p(v)^2$, so trivial clauses are common when $p(v) \geq v^{-1/2}$ and relatively rare when $p(v) \ll v^{-1/2}$.

This model of a set of random predicates differs slightly from the model developed by Goldberg [8], [9], [10], and more significantly from the one used by us [1, 15] in earlier work. The present model has the virtue of simplicity: the predicates that result from setting a few variables and simplifying are essentially the same as the original predicates. This property makes the analysis straightforward. Clauses in Goldberg’s model are not permitted to contain both positive and negative literals, which makes the proportion of trivial problems in a random set smaller. The results of the present paper are the same, however, whether that model or our model is used; this may be seen by replacing the factor $(1-p)^{2t}$ by $(1-2p)^t$ in (1) and the following derivations.

It is not as easy to compare results obtained with this model with the fixed clause size model of [1], [15]. The initial form of the predicates in the fixed clause size model is quite different from that in the present model. As variables are set and predicates simplified, however, the predicates begin to resemble those obtained from the present model. This may explain why the results obtained from the present model are at least qualitatively the same as those that would be obtained from a fixed clause size model.

3. Comparison of algorithms. A fundamental difficulty with average time analyses is that the average depends not only on the algorithm being evaluated, but also on the weights assigned to the problems. Some algorithms will be good on some problem sets, while other algorithms will be good on other problem sets. Our approach to this difficulty is to include two parameters in our models, $t(v)$ and $p(v)$. By varying these two parameters we can generate problems where the typical problem sets have widely different characteristics.

The most complete way to compare two algorithms when using such a model is to compute the average running time of each algorithm for each function $p(v)$ and each function $t(v)$. The analytical techniques we have given here and in [14], [16] can be used to produce such results (see Theorems 3, 4, and 5 of this paper, for example). These results, however, give too much data for easy interpretation. We find that a plot such as Fig. 1, which shows for several algorithms the curve that divides polynomial time from exponential time, is a particularly useful summary of the results of an analysis. Imagine a rectangular space with various functions $p(v)$ labeling a vertical axis along the left edge, and functions $t(v)$ labeling a horizontal axis along the bottom. The functions are arranged so that the most rapidly growing $t(v)$ are at the right and the most slowly decaying $p(v)$ are at the top. This quasigraphical representation should

not be taken too literally; the functions for $t(v)$ and $p(v)$ do not form a complete ordered field.

Each point in Fig. 1 represents a set of random problems characterized by a particular choice of functions $p(v)$ and $t(v)$. When one algorithm gives a curve that is inside the curve produced by a second algorithm, then the first algorithm is clearly a much better algorithm for our model of random problems. It is probably also a much better algorithm in practice. When two algorithms produce contours that intersect, then each algorithm is good for some problems that the other algorithm is not good for.

A problem set that contains an exponentially small fraction of hard problems can still have an exponential average solution time, if those problems are hard enough (e.g. $e^{xv}e^{-yv} = e^{(x-y)v}$ grows exponentially with v if $x - y > 0$). Algorithms that work well on satisfiability problems do so because, after a few variables have been set, most problems reduce to a trivial problem. Since predicates in our model tend to look about the same before and after some variables are set, there is a tendency for problems to be either easy or difficult from the start. More realistic looking models have similar transitions from polynomial to exponential average time, but the complications of the models tend to obscure what is going on.

The lower boundary of the polygon in Fig. 1 divides the region where backtracking takes polynomial average time from that where it takes exponential average time. (See [16] for the derivation of these results.) The lower part of that contour divides the region where problems have, on average, an exponential number of solutions from those that have a polynomial number. This line is extended into the upper part of the figure as a dotted line. Since the backtracking algorithm we studied finds all solutions to a problem, backtracking must take exponential time in that region. The shaded part of the figure shows the region where backtracking takes exponential time on problems that have, on average, a polynomial number of solutions; it is labeled "difficult".

The bounds shown in Fig. 1 for the pure literal rule, reflecting results obtained in this paper, are a big improvement over the bounds obtained by previous analyses. Still, the bounds occur in regions where the typical problems are somewhat strange. At the top boundary the clauses become large as the number of variables becomes large. Each particular clause is extremely easy to satisfy. At the left boundary there are very few clauses compared to the number of variables. Along much of the lower boundary the typical clause contains no variables, so the typical predicate in it is obviously unsatisfiable. Still, for each of these regions, for problem sets just inside the innermost contour line, there are enough difficult problems that no one has yet shown how to solve them in polynomial average time. Consider for a moment the point on the contour for backtracking where $p(v) = 1/v^2$. The typical predicate here contains empty clauses and backtracking usually determines this quickly. Nonetheless, to the left of the boundary the average number of solutions per problem is exponential (because some problems have a very large number of solutions), and backtracking that finds every solution takes exponential average time.

It is more difficult to make any definitive statements about the pure literal rule because no lower bound analysis has been done on its performance. At present all that we can say is that we have an upper bound on its speed, and that for some regions this upper bound is much better than any previous upper bound. Without a lower bound analysis, we cannot tell how tight the upper bound for the pure literal rule is. It may be that the pure literal rule algorithm can only solve trivial problems quickly. However, many of the problems that can be solved quickly by the pure literal rule apparently cannot be solved quickly by other simple algorithms.

Once a lower bound analysis of the pure literal rule is done, so that the location of the boundary of the hard region (for the pure literal rule and backtracking) is determined, problems just inside the boundary should be of particular interest to the developers of new algorithms. Such problems are difficult for current algorithms, but perhaps for trivial reasons. Each time one can find a trivial reason for a problem being difficult, one can hope to develop an improved algorithm.

Another interesting question is whether algorithms can be found to completely eliminate the hard region. In other words, can an algorithm be found that will solve the problems corresponding to each point in Fig. 1 in polynomial average time? At present there is no reason to believe that such an algorithm exists. Random CNF satisfiability appears to be quite difficult for any known algorithm, as long as the parameters in the random distribution are adjusted to avoid generating mainly trivial problems. Of course, if anyone ever proves that random satisfiability is hard for every algorithm, that will be a major achievement equivalent to proving that $NP \neq P$.

4. The algorithm. Goldberg's simplified version of the Davis-Putnam procedure selects variables in a fixed order. The algorithm is:

- (1) If all variables have been selected, stop. The predicate is satisfiable if it contains no clauses; it is unsatisfiable if it contains empty clauses. (If all variables have been selected, then one of these two conditions must hold.)
- (2) Select the next variable. If both it and its negation occur as literals generate two subproblems by setting the selected variable to *true* and to *false* and simplifying the two resulting predicates. (To simplify a predicate drop all clauses that contain a *true* literal and drop any *false* literals.) Otherwise, the variable corresponds to a pure literal. In this case, generate one subproblem by setting the variable to the value that makes the literal *true*. (If the variable does not occur at all, it can be set either way.)
- (3) Solve recursively the subproblems that have been generated. The original predicate is satisfiable if any subproblem is satisfiable; otherwise it is unsatisfiable.

The use of a fixed order for selecting the variables greatly simplifies the analysis. To do the analysis, we assume that the time required for one step of the algorithm is $vt(v)$. The average time is then given by the recurrence

$$\begin{aligned}
 (1) \quad A(t, v) &= vt + 2 \sum_{i \geq 1} \binom{t}{i} p^i (1-p)^{t-i} A(t-i, v-1) + (1-p)^{2t} A(t, v-1), \\
 A(t, 0) &= A(0, v) = 0.
 \end{aligned}$$

Here the summation accounts for the ways in which subproblems can arise when the selected variable occurs in the predicate; the term $(1-p)^{2t} A(t, v-1)$ allows for the case where the selected variable does not occur at all. (We use the convention that $\binom{t}{i} = 0$ for $i > t$ and for $i < 0$. Summation limits are omitted whenever the summand is zero outside of the desired range.) A more detailed derivation is given in [10].

5. Analysis. The basic technique that we use to bound $A(t, v)$ is developed in Theorems 1 and 2. In Theorems 3, 4, and 5 we derive bounds for $A(t, v)$.

THEOREM 1. *Let $B(t, v)$ be any function such that $A(t, v) \leq B(t, v)$ for $v = v_0 - 1, 0 \leq t \leq t_0$. Then*

$$(2) \quad A(t_0, v_0) \leq t_0 v_0 + 2 \sum_{i \leq t_0} \binom{t_0}{i} p^i (1-p)^{t_0-i} B(t_0-i, v_0-1) + (1-p)^{2t_0} B(t_0, v_0-1).$$

Proof. The coefficients on the right sides of (1) and (2) are equal and positive. Since $A(t, v) \leq B(t, v)$ for all the values in question, the right side of (2) is greater than or equal to the right side of (1).

THEOREM 2. *Let $B(t, v)$ be any function such that $A(t, v) \leq B(t, v)$ for $v = v_0 - 1, 0 \leq t \leq t_1$, and for $v_0 \leq v \leq v_1, 0 \leq t \leq t_0$. If*

$$(3) \quad B(t, v) \geq tv + 2 \sum_{1 \leq i} \binom{t}{i} p^i (1-p)^{t-i} B(t-i, v-1) + (1-p)^{2t} B(t, v-1)$$

for $v_0 \leq v \leq v_1, t_0 < t \leq t_1$, then $A(t, v) \leq B(t, v)$ for $v_0 - 1 \leq v \leq v_1, 0 \leq t \leq t_1$.

Proof. The right side of (2) is the same as the right side of (3), so the theorem can be proved by double induction.

For any $B(t, v)$ define

$$(4) \quad X(t, v) = B(t, v) - 2 \sum_{1 \leq i} \binom{t}{i} p^i (1-p)^{t-i} B(t-i, v-1) - (1-p)^{2t} B(t, v-1)$$

$$(5) \quad \begin{aligned} &= B(t, v) - 2 \sum_i \binom{t}{i} p^i (1-p)^{t-i} B(t-i, v-1) \\ &+ (2(1-p)^t - (1-p)^{2t}) B(t, v-1). \end{aligned}$$

Note that $B(t, v)$ satisfies condition (3) in Theorem 2 if and only if $X(t, v) \geq tv$. We are interested in small functions $B(t, v)$ for which $X(t, v)$ satisfies this condition. We proceed in a way similar to the method of repertoire [11]: we select easy to sum functions for $B(t, v)$ and adjust parameters so that the hypothesis of Theorem 2 is satisfied.

One of the more complex functions of t for which it is clear how to do the sum over i is given by

$$(6) \quad B(t, v) = cg! \binom{t}{g} h^t \frac{v(v+1)}{2},$$

where c, g , and h may depend on p , but not on t or v . For this function,

$$(7) \quad \begin{aligned} X(t, v) = \frac{c}{2} g! \binom{t}{g} h^t v \left\{ \left[1 - 2(1-p)^g \left(1 - \frac{h-1}{h} p \right)^{t-g} \right. \right. \\ \left. \left. + 2(1-p)^t - (1-p)^{2t} \right] (v-1) + 2 \right\}. \end{aligned}$$

THEOREM 3. $A(t, v) \leq 2^{t-2} v(v+1)$.

Proof. Let $c = \frac{1}{2}, g = 0$, and $h = 2$ in the above formula, making $B(t, v) = 2^{t-2} v(v+1)$ and

$$(8) \quad X(t, v) = 2^{t-2} v \left\{ \left[1 - 2 \left(1 - \frac{p}{2} \right)^t + 2(1-p)^t - (1-p)^{2t} \right] (v-1) + 2 \right\}.$$

We have $B(0, v) = v(v+1)/4$ and $B(t, 0) = 0$, so Theorem 2 can be applied with $v_0 = 1, t_0 = 0, v_1 = \infty$, and $t_1 = \infty$, provided we can show that $X(t, v) \geq tv$ for all $v \geq 1$ and $t \geq 1$. Now, $X(t, 1) \geq 2^{t-1} \geq t$ for $t \geq 1$, so we have $X(t, v) \geq tv$ for $v \geq 1$ provided the coefficient of $v-1$ in (8) is positive. If we call this coefficient $C(p, t)$, then we need

$$(9) \quad C(p, t) = 1 - 2 \left(1 - \frac{p}{2} \right)^t + 2(1-p)^t - (1-p)^{2t} \geq 0 \quad \text{for } 0 \leq p \leq 1, 1 \leq t.$$

The easiest way to see that $C(p, t) \geq 0$ is to write

$$(10) \quad C(p, t) = 1 - 2 \left(\frac{1 + (1-p)}{2} \right)^t + 2(1-p)^t - (1-p)^{2t}$$

$$(11) \quad \geq 1 - 2 \left(\frac{1^t + (1-p)^t}{2} \right) + 2(1-p)^t - (1-p)^{2t}$$

$$(12) \quad = (1-p)^t - (1-p)^{2t} \geq 0.$$

Thus $A(t, v) \leq B(t, v)$ by Theorem 2.

THEOREM 4.

$$A(t, v) \leq \left[\frac{2^g}{p} \binom{t}{g} + 2^{g-2} \right] v(v+1) \quad \text{for } g = \left\lceil \frac{\ln 2}{-\ln(1-p)} \right\rceil + 1 \text{ and } 0 < p < 1.$$

Proof. Let

$$B(t, v) = \left[\frac{2^g}{p} \binom{t}{g} + 2^{g-2} \right] v(v+1).$$

Note that $(1-p)^g \leq (1-p)/2$ and $2(1-p)^t - (1-p)^{2t} \geq 0$, so by applying (7) to each of the two terms of the formula for $B(t, v)$ we have

$$(13) \quad X(t, v) \geq \frac{2^g}{p} \binom{t}{g} v[p(v-1)+2] + 2^{g-2} v[-(v-1)+2].$$

For $t \leq g-1$, $\binom{t}{g} = 0$ and $B(t, v) = 2^{g-2} v(v+1) \geq A(t, v)$ by Theorem 3. For $t = g$, $\binom{t}{g} = 1$ and

$$B(t, v) = \left[\frac{2^g}{p} + 2^{g-2} \right] v(v+1) \geq 2^g v(v+1) \geq A(t, v).$$

For $v = 0$, $B(t, v) = 0 = A(t, v)$. Now, $X(t, 0) = 0$ (tv for $v = 0$) and

$$X(g+1, 1) \geq \frac{2^{g+1}}{p} (g+1) + 2^{g-1} \geq g+1 \quad (tv \text{ for } t = g+1, v = 1).$$

Since $\binom{t}{g}$ increases more rapidly than linearly with t for $t > g+1$, we will have $X(t, v) \geq tv$ for $v \geq 0$, $t \geq g+1$ if the sum of the coefficients for $v-1$ in (13) is nonnegative for $t \geq g+1$. The coefficient is

$$\left[2^g \binom{t}{g} - 2^{g-2} \right] v \geq 0 \quad \text{for } t \geq g.$$

Thus, by Theorem 2 with $v_0 = 1$, $t_0 = g$, $v_1 = \infty$, $t_1 = \infty$, we have $A(t, v) \leq B(t, v)$.

THEOREM 5.

$$A(t, v) \leq \frac{t[U^{v+1} - (v+1)U + v]}{(1-U)^2}$$

where $U = 2 - 2(1-p)^t + (1-p)^{2t}$.

Proof. Let

$$B(t, v) = t \frac{[U^{v+1} - (v+1)U + v]}{(1-U)^2} = t \sum_{0 \leq i \leq v} (v-i)U^i.$$

Now $U = 1 + (1 - (1-p)^t)^2$ is an increasing function of t , so $B(t, v)$ is also an increasing

function of t . Therefore,

$$\begin{aligned}
 X(t, v) &= B(t, v) - 2 \sum_{i \geq 1} \binom{t}{i} p^i (1-p)^{t-i} B(t-i, v-1) - (1-p)^{2t} B(t, v-1) \\
 (14) \quad &\geq B(t, v) - B(t, v-1) \left[2 \sum_{i \geq 1} \binom{t}{i} p^i (1-p)^{t-i} + (1-p)^{2t} \right] \\
 &= B(t, v) - B(t, v-1) [2 - 2(1-p)^t + (1-p)^{2t}].
 \end{aligned}$$

But $B(t, v)$ is the solution of the recurrence

$$\begin{aligned}
 (15) \quad &B(t, v) = UB(t, v-1) + tv, \\
 &B(t, 0) = 0,
 \end{aligned}$$

so $X(t, v) \geq tv$. Also, $B(t, 0) = 0$, so, by Theorem 2, $A(t, v) \leq B(t, v)$.

6. Asymptotics. In this section we give brief derivations of the asymptotic behavior of the bounds obtained in Theorems 3-5. Figure 1 summarizes these results. The contours in Fig. 1 indicate where the average time has been shown to be a polynomial function of the problem size. In this section we determine where the average time is polynomial in the number of variables. These concepts are equivalent where $t(v)$ is a polynomial function of v ; they differ where $t(v)$ increases more rapidly than polynomially.

For the average time to be polynomial in v , it is sufficient that

$$\begin{aligned}
 (16) \quad &\lim_{v \rightarrow \infty} \frac{A(t(v), v)}{v^n} \leq 1, \quad \text{or} \\
 &\lim_{v \rightarrow \infty} \frac{\ln(A(t(v), v))}{n \ln v} \leq 1
 \end{aligned}$$

for some positive integer n .

Using the bound from Theorem 3, we have polynomial average time when

$$(17) \quad 2^{t(v)-2} v(v+1) \leq v^n, \quad \text{or}$$

$$(18) \quad t(v) \leq \frac{n \ln v - \ln(v(v+1))}{\ln 2} + 2 = \left(\frac{n-2}{\ln 2} \right) \ln v + \text{lower order terms.}$$

Therefore, the time is polynomial when

$$(19) \quad t(v) \leq \frac{n}{\ln 2} \ln v.$$

The best previous bound for low $t(v)$ (for polynomial time when $t(v)$ is below a certain bound) was $t(v) \leq (\ln \ln v) / (\ln 3)$ [16].

Using Stirling's approximation for factorial on Theorem 4 gives

$$\begin{aligned}
 (20) \quad A(t(v), v) &\leq \frac{v^2}{p(v)} \left(\frac{1}{2\pi g(v)} \right)^{1/2} \left(\frac{2t(v)}{g(v)} \right)^{g(v)} \left(1 - \frac{g(v)}{t(v)} \right)^{-t(v)+g(v)} \\
 &\quad \times \left[1 + O\left(\frac{1}{v}\right) + O\left(\frac{1}{g(v)}\right) + O\left(\frac{1}{t(v)-g(v)}\right) + O(p(v)) + O\left(\frac{g(v)}{t(v)}\right) \right]
 \end{aligned}$$

$$(21) \quad \begin{aligned} &\cong \frac{v^2}{p(v)} \left(\frac{1}{2\pi g(v)} \right)^{1/2} \left(\frac{2et(v)}{g(v)} \right)^{g(v)} \\ &\times \left[1 + O\left(\frac{1}{v}\right) + O\left(\frac{1}{g(v)}\right) + O\left(\frac{1}{t(v) - g(v)}\right) + O(p(v)) + O\left(\frac{g(v)}{t(v)}\right) \right]. \end{aligned}$$

The g of Theorem 4 is no more than $\ln 2/p(v) + 2$ for large v . When $t(v) > \frac{3}{2}g(v) + \frac{1}{2}$, the limit in Theorem 4 is an increasing function of g . Assuming that $p(v)$ goes to zero and that $t(v)p(v)$ approaches a number less than $3 \ln 2/2$, using the upper bound of $\ln 2/p(v) + 2$ for g in Theorem 4 gives

$$(22) \quad \begin{aligned} A(t(v), v) &\leq v^2 \left(\frac{p(v)}{2\pi \ln 2} \right)^{1/2} \left(\frac{2et(v)}{\ln 2/p(v) + 2} \right)^{\ln 2/p(v) + 2} \\ &\times \left[1 + O\left(\frac{1}{v}\right) + O(p(v)) + O\left(\frac{1}{p(v)t(v)}\right) \right]. \end{aligned}$$

If the right side of (22) is bounded by v^n for some integer n , then the time $A(t(v), v)$ is polynomial. If $1/p(v)$ is bounded by a polynomial function of v , then all of the right side of (22) is polynomial except for the term raised to the $\ln 2/p(v)$ power. Thus we have polynomial time when

$$(23) \quad t(v)p(v) \leq \frac{\ln 2}{2e} v^{np(v)/\ln 2} [\text{polynomial in } v, t(v), 1/p(v)]^{p(v)} \leq v^{n_1 p(v)}$$

for some integer n_1 . The derivation of (23) assumes that $1/p(v)$ is bounded by a polynomial function of v , but the results given below also imply that (23) is still true when $1/p(v)$ does not have a polynomial bound. For $p(v) = \epsilon$, (23) is the same result as in [10], but it is better for smaller values of $p(v)$. The boundary given by (18) and the one given by (23) intersect at

$$(24) \quad \begin{aligned} p(v) &= \left(\frac{\alpha}{n \ln v} \right) \approx \frac{2.241}{n \ln v}, \\ t(v) &= \frac{n \ln v}{\ln 2}. \end{aligned}$$

where α is the root of the equation

$$\alpha = \ln 2 \left(\ln \alpha + \ln \left(\frac{2e}{(\ln 2)^2} \right) \right) + \text{lower order terms}.$$

The bound of Theorem 5 gives polynomial time (for polynomial $t(v)$) when

$$(25) \quad U^v \leq v^n \quad \text{or} \quad \ln U \leq n(\ln v)/v.$$

This can be true only if $U \approx 1$, so in this case we have

$$(26) \quad \ln U = \ln (1 + [1 - (1 - p(v))^{t(v)}]^2) \approx [1 - (1 - p(v))^{t(v)}]^2.$$

Polynomial time results when

$$(27) \quad 1 - (1 - p(v))^{t(v)} \leq \sqrt{\frac{n \ln v}{v}}, \quad \text{or} \quad \ln \left[1 - \sqrt{\frac{n \ln v}{v}} \right] \leq t(v) \ln (1 - p(v)).$$

For large v and small $p(v)$ this is equivalent to

$$(28) \quad -\sqrt{\frac{n \ln v}{v}} \leq -t(v)p(v) \quad \text{or} \quad t(v) \leq \frac{1}{p(v)} \sqrt{\frac{n \ln v}{v}} \quad (\text{for small } p(v))$$

for any large integer n_1 .

This is a new type of bound for the pure literal rule. Previous bounds showed that the rule is good for large $p(v)$; this one shows that it is good for small $p(v)$. The result is similar to, but better than, a result in [17]. It is interesting that (28) is also the condition for polynomial time for the algorithm that: (1) finds all variables that are initially pure literals; (2) assigns values to make all the pure literals true; and (3) solves the resulting problem using exhaustive search (a similar algorithm is analyzed in [17]). Further analysis may lead to a better bound on the performance of the pure literal rule in this region. The intersection of the boundaries given by (18) and (27) is at

$$(29) \quad \begin{aligned} p(v) &= (\ln 2)n^{-1/2}(v \ln v)^{-1/2}, \\ t(v) &= n \ln v. \end{aligned}$$

In [16] we show that backtracking requires polynomial average time (for $p(v) \leq (\ln 2)/v$) when

$$(30) \quad t(v) \leq \frac{(\ln 2 - \varepsilon)v}{-\ln(1 - \exp(-vp(v)))}.$$

This boundary intersects with (27) (ignoring the ε) when

$$(31) \quad \frac{(\ln 2)v}{-\ln(1 - \exp(-vp(v)))} = \frac{1}{p(v)} \sqrt{\frac{n \ln v}{v}}.$$

Since $vp(v)$ is small at the intersection, for large v we have

$$(32) \quad \begin{aligned} \frac{(\ln 2)v}{-\ln vp(v)} &\approx \frac{1}{p(v)} \sqrt{\frac{n \ln v}{v}} \quad \text{or} \\ p &\approx \frac{\sqrt{n}}{2 \ln 2} \left(\frac{\ln v}{v}\right)^{3/2} \quad \text{and} \quad t \approx \frac{2(\ln 2)v}{\ln v}. \end{aligned}$$

7. Conclusion. The pure literal rule (with backtracking) can solve many sets of problems in polynomial average time. The results of the present analysis represent a major improvement over the previous analysis [10] of the pure literal rule; moreover, the loose nature of the approximations used in the analysis suggests that further improvements are possible, particularly when $p(v)t(v)$ is small. No lower bound analysis has been done for the pure literal rule. Although it is unlikely that all the problem sets in the figure can be solved in polynomial average time using that rule, it would be interesting to see where the region of problems that are hard for the pure literal rule lies.

The algorithms that have been analyzed so far each contain only a subset of the useful features of the Davis-Putnam procedure. The interaction between backtracking and the pure literal rule has not been analyzed ([2] gives the first step of such an analysis). The effect of the pure literal rule and the unit clause rule on the order of selecting variables has been omitted (see [14] for the effect of the unit clause rule on backtracking). No analysis of the effect of stopping the search as soon as one solution is found has been done. (The pure literal rule causes some solutions to be skipped.)

The interactions of all these basic techniques are potentially important and should be analyzed.

There are several more sophisticated techniques that also should be analyzed for their effect on average time. These include selecting variables from short clauses [12], other generalizations of the unit clause rule [18], and two generalizations of the pure literal rule [12, 13]. The algorithm developed by Monien and Speckenmeyer [12] has a good worst-case time for solving CNF predicates.

Another approach for analyzing satisfiability problems was developed by Franco and Paull [6]. They studied the probability that an algorithm completes the search for a solution in polynomial time when doing exhaustive search for the first solution. Franco used the same technique to study the pure literal rule [5]. The results of analyses of satisfiability algorithms help to identify the types of problems that each algorithm does well on, and should provide insights that will lead to better algorithms.

Acknowledgments. We would like to thank Professor George Minty for his help with Theorem 3, and we would like to thank a referee for the proof presented here and for carefully checking the rest of the paper.

REFERENCES

- [1] CYNTHIA A. BROWN AND PAUL WALTON PURDOM, JR., *An average time analysis of backtracking*, this Journal, 10 (1981), pp. 583–593.
- [2] ———, *How to search efficiently*, Proc. Seventh International Joint Conference on Artificial Intelligence, 1981, pp. 588–594.
- [3] MARTIN DAVIS AND HILARY PUTNAM, *A computing procedure for quantification theory*, J. Assoc. Comput. Mach., 7 (1960), pp. 201–215.
- [4] MARTIN DAVIS, GEORGE LOGEMANN, AND DONALD LOVELAND, *A machine program for theorem proving*, Comm. ACM, 5 (1962), pp. 394–397.
- [5] JOHN FRANCO, *Average analysis of the pure literal heuristic*, Case Institute of Technology Report No. CES-81-4, Cleveland, OH.
- [6] JOHN FRANCO AND MARVIN PAULL, *Probabilistic analysis of the Davis–Putnam procedure for solving the satisfiability problem*, Discrete Appl. Math., 5 (1983), pp. 77–87.
- [7] MICHAEL R. GAREY AND DAVID S. JOHNSON, *Computers and Intractability*, W. H. Freeman, San Francisco, 1979.
- [8] ALLEN GOLDBERG, *On the complexity of the satisfiability problem*, Courant Computer Science Report No. 16, New York Univ., New York.
- [9] ———, *Average case complexity of the satisfiability problem*, Proc. Fourth Workshop on Automated Deduction, 1979, pp. 1–6.
- [10] ALLEN GOLDBERG, PAUL WALTON PURDOM, JR. AND CYNTHIA A. BROWN, *Average time analysis of simplified Davis–Putnam procedures*, Inform. Processing Lett., 15 (1982), pp. 72–75.
- [11] DANIEL H. GREENE AND DONALD E. KNUTH, *Mathematics for the Analysis of Algorithms*, Birkhäuser, Boston, 1981.
- [12] BURKHARD MONIEN AND EWALD SPECKENMEYER, *Three-satisfiability is testable in $O(1.62^r)$ steps*, Report No. 3, Theoretical Informatics Series, Univ. Paderborn, 1979.
- [13] PAUL WALTON PURDOM, JR. *Solving satisfiability problems with less searching*, IEEE TPAMI, 4 (1982), pp. 309–316.
- [14] ———, *Search rearrangement backtracking and polynomial average time*, Artificial Intelligence, 21 (1983), pp. 117–133.
- [15] PAUL WALTON PURDOM, JR. AND CYNTHIA A. BROWN, *An analysis of backtracking with search rearrangement*, this Journal, 12 (1983), pp. 717–733.
- [16] ———, *Polynomial average-time satisfiability problems*, Indiana University Computer Science Report No. 118, Bloomington, IN.
- [17] ———, *Evaluating search methods analytically*, National Conference on Artificial Intelligence, 1982, pp. 124–127.
- [18] PAUL PURDOM, CYNTHIA BROWN AND EDWARD ROBERTSON, *Multi-level dynamic search rearrangement*, Acta Inform., 15 (1981), pp. 99–114.

ALGORITHMIC APPROACHES TO SETUP MINIMIZATION*

ULRICH FAIGLE[†], GERHARD GIERZ[‡] AND RAINER SCHRADER[†]

Abstract. Construction of classes of ordered sets are given for which the setup minimization problem can be solved by an efficient algorithm. Those constructions generalize series-parallel connections. Special classes of ordered sets are exhibited for which the greedy algorithm yields an optimal linear extension. In particular, it is shown that the class of N -free ordered sets is both defect optimal and strongly greedy.

Key words. partially ordered sets, scheduling problems, NP-complete problems

1. Introduction. The setup minimization problem arises from a special scheduling problem: n jobs are to be processed on a single machine subject to precedence constraints given by a partial ordering. Every job processed after a job which is not constrained to precede it causes a "setup." We seek to determine the setup number of an optimal schedule.

By Szpilrajn [15] every ordered set P allows a feasible schedule L . However, in general no good algorithms constructing optimal schedules are known. As a matter of fact, Pulleyblank [13] has shown that the problem is NP-complete. Nevertheless it turns out that the problem to find the setup number of optimal linear extensions can be solved for a large variety of ordered sets. Duffus et al. [6] have shown that the setup number of an ordered set containing no "crown" equals its width minus one. Although the width of the ordered set P is easy to compute it appears difficult to recognize whether P contains no crown. Cogis and Habib [4] have demonstrated that an optimal extension for "series-parallel" partially ordered sets can be constructed by a greedy algorithm. Rival [14] has extended this result to the more general class of ordered sets whose Hasse diagrams do not contain N as an induced subgraph. For related graph-theoretic investigations see [3], [16] and [17].

A different approach is taken by the second author and Poguntke [8] who study defect optimal ordered sets, namely sets whose setup number equals the defect minus one of its reduced incidence matrix. They prove that this class of ordered sets is closed under substitution (i.e., lexicographical sums) and series-parallel composition and thus includes the series-parallel ordered sets.

In this paper we broaden the above results by providing general composition techniques, called bipartite sums, for ordered sets. We show that for these bipartite sums the setup number can be computed whenever the setup numbers of the components are known. Moreover, we present efficient algorithms to test whether an order is generated from singletons under bipartite sums, and, if so, simultaneously solve the setup problem.

We introduce the notation we need in § 2 and present some basic results. In § 3 we consider weakly linear sums of ordered sets and show that the properties of being defect optimal or "strongly greedy" are preserved under this construction. N -free ordered sets form an important class of examples which are both defect optimal and strongly greedy. The upper bipartite sums in § 4 also generalize series-parallel compositions. The class of ordered sets generated by upper bipartite sums from singletons properly contains the N -free ordered sets. Moreover, ordered sets which are not necessarily "greedy" may be obtained in this way and we give conditions under which

* Received by the editors July 16, 1984.

[†] Sonderforschungsbereich 21 (DFG), Institut für Operations Research, Universität Bonn, Bonn, West Germany.

[‡] Department of Mathematics, University of California, Riverside, California 92521.

upper bipartite sums yield strongly greedy sets. An efficient recognition algorithm is available. Finally, in § 5 we present a decomposition algorithm for classes of ordered sets generated from arbitrarily given sets of irreducibles under series-parallel compositions. If the setup numbers of the irreducibles are known the setup numbers of the composed order can be computed from its decomposition tree.

2. Definitions and preliminaries. In this section we collect some basic facts about (partially) ordered sets and their linear extensions. We assume the reader to be familiar with the elementary terminology of ordered sets (see, e.g., Birkhoff [1]). Additional notation will be introduced in the sequel. Let $P = (E, \preceq)$ be an ordered set with finite ground set $E, |E| = n$. A linear ordered set $L = (E, \leq)$ with the same ground set E is a *linear extension* or *schedule* of P if for all $a, b \in E, a \preceq b$ in P implies $a \leq b$ in L .

If L is a linear extension of P and if $a, b \in E$ are such that b is an upper neighbor of a in L but $a \not\preceq b$ in P , then the pair (a, b) is called a *setup* (or *jump*) of L . By $s(L)$ we denote the number of a setups of L . The problem consists in determining the *setup number* $s(P)$ of P defined by

$$s(P) = \min \{s(L) : L \text{ is a linear extension of } P\}.$$

The setups divide the linear extension $L = a_1 a_2 \cdots a_n$ into subchains C_j which are also chains in P . We denote such a decomposition of L by $L = C_1 + C_2 + \cdots + C_k$. Clearly, $k = s(L) + 1$. It follows from the definitions that for every $1 \leq i \leq n, L_i = a_1 a_2 \cdots a_i$ forms an (order) ideal in P , i.e., for every $y \in L_i, x \in E, x \preceq y$ in P implies $x \in L_i$.

We now continue with the definition of greedy setups. In some sense, greedy setups are those setups in a given linear extension “which cannot be avoided.” Let $b \in E$ be given. We define

$$N^-(b) = \{a \in E : a \text{ is a lower neighbor of } b \text{ in } P\}$$

and

$$N^+(b) = \{a \in E : a \text{ is an upper neighbor of } b \text{ in } P\}.$$

Note that upper and lower neighbors are defined with respect to P and not with respect to linear extensions of P . We define a setup (A_i, a_{i+1}) in the linear extension L to be *greedy* if a_i does not have an upper neighbor $b \in E \setminus L_i$ such that $N^-(b) \subset L_i$. Let

$$g(L) = |\{(a, b) \in E \times E : (a, b) \text{ is a greedy setup}\}|$$

be the number of greedy setups of L and let

$$ng(L) = s(L) - g(L).$$

A linear extension is *greedy* if $g(L) = s(L)$. The ordered set P is called *greedy* if $ng(L) = 0$ implies that L is optimal and *strongly greedy* if $s(P) = g(L)$ for every linear extension L of P . Thus, in particular, greedy schedules and optimal schedules coincide for strongly greedy ordered sets.

A greedy schedule can always be obtained via the following algorithm: Choose any minimal element $a \in P$. We then construct a linear extension recursively as follows:

If $a = a_1, a_2, \dots, a_i$ have already been chosen, then do one of the following two things:

- (1) There exists an upper neighbor $b \in N^+(a_i)$ such that all lower neighbors of b have already been chosen. In this case let $a_{i+1} = b$.
- (2) No such choice is possible. Then let a_{i+1} be any element minimal in $E \setminus \{a_1, \dots, a_i\}$.

The following result is well known.

LEMMA 2.1. *There always exists an optimal greedy schedule.*

Proof. Let $C_1 + C_2 + \dots + C_k$ be an optimal linear extension. If the setup from C_1 to C_2 is not greedy, then there exists a minimal element b of $P \setminus C_1$ such that $C_1 \cup \{b\}$ is a chain of P and $N^-(b) \subset C_1$. Furthermore, b must be the initial element of some chain C_i , where $i > 1$. Hence $(C_1 \cup \{b\}) + \dots + (C_i \setminus \{b\}) + \dots + C_k$ is also an optimal schedule. Repeating this argument yields the proof. \square

We now fix an arbitrary field K . Let $M(P)$ be the (reduced) incidence matrix of P , i.e., $M(P) = (m_{a,b})_{a,b}$ where

$$m_{a,b} = \begin{cases} 1 & \text{if } a < b, \\ 0 & \text{otherwise.} \end{cases}$$

The defect $\text{def}_K(P)$ of P with respect to the field K was introduced in [8] as

$$\text{def}_K(P) = |E| - \text{rk}_K(M(P)),$$

where $\text{rk}_K(M(P))$ means the rank of the incidence matrix of P . Note that the defect of P is independent of the index order used for the incidence matrix.

The following fundamental relation was proved in [8]:

$$w(P) - 1 \leq \text{def}_K(P) - 1 \leq s(P),$$

where $w(P)$ is the width of P , i.e., the maximum size of a subset of P whose elements are pairwise incomparable. In view of Lemma 2.1, our next result improves this inequality.

THEOREM 2.2. *Let $L = a_1 a_2 \dots a_n$ be a schedule of P . Then*

$$\text{def}_K(P) - 1 \leq g(L).$$

Proof. We assume w.l.o.g. that the rows and columns of $M(P)$ are indexed in the order given by L . Let I_N be those indices i with $a_i < a_{i+1}$, let I_M be those indices for which there is a nongreedy setup from a_i to a_{i+1} and set $I = I_M \cup I_N$. We must show that $\text{rk}_K(M(P)) \geq (n - 1) - g(L) = |I|$. First of all, note that $|I| + g(L) = (n - 1)$: Indeed, if $i \leq (n - 1)$, then passing from a_i to a_{i+1} either is a nonsetup, a nongreedy setup or a greedy setup.

To show $|I| \leq \text{rk}_K(M(P))$, we define a mapping $f: I \rightarrow \{1, \dots, n\}$ by

$$f(i) = \begin{cases} i + 1 & \text{if } i \in I_N, \\ \min \{j: a_i \in N^-(a_j) \subset L_i\} & \text{otherwise.} \end{cases}$$

Then the second expression for f indeed defines f also on I_N showing that $f(i) \geq i + 1$ and that f is injective.

Consider now the $(|I| \times |I|)$ -submatrix B of $M(P)$ induced by the rows corresponding to I and the columns corresponding to $f(I)$. We claim that B is invertible. To see this, we rearrange the columns of B in the order given by f^{-1} . Hence B becomes $B' = (b'_{ij})_{i,j \in I}$ where the indices carry the order given by the natural numbers and where

$$b'_{ij} = \begin{cases} 1 & \text{if } a_i < a_{f(j)}, \\ 0 & \text{otherwise.} \end{cases}$$

By the definition of the mapping f , we have $b'_{ii} = 1$ for every i , since $a_i < a_{f(i)}$. Now assume that $j < i$. We want to show that $b'_{ij} = 0$. Supposing that $a_i < a_{f(j)}$, we consider two cases:

(a) $f(j) = j + 1$. Then $j < i$ implies $j + 1 \leq i$. Now, $a_i < a_{j+1} = a_{f(j)}$ and the fact that $L = a_1 a_2 \dots a_n$ is a linear extension of P yields $i < j + 1$, a contradiction.

(b) $f(j) > j + 1$. Then $j \in I_M$ and therefore we have a nongreedy setup from a_j to a_{j+1} . From the definition of f we conclude that $a_{f(j)}$ is an upper neighbor of a_j and that all lower neighbors of $a_{f(j)}$ are contained in L_j . By assumption, $a_i < a_{f(j)}$. Pick a lower neighbor a_k of $a_{f(j)}$ which dominates a_i . Then we have $a_i \leq a_k \in L_j$. Since the set L_j is an order ideal, $a_i \in L_j$, contradicting $j < i$.

Hence we obtain a contradiction in either case. Therefore, $a_i < a_{f(j)}$ is impossible. Thus B' is an upper triangular matrix of full rank and the proof is complete. \square

A partially ordered set P for which $\text{def}_K(P) - 1 = s(P)$ holds we will call *K-defect optimal*. Defect optimal partially ordered sets were studied in [8].

COROLLARY 2.3. *If P is K-defect optimal then every optimal schedule of P is greedy.*

The ordered sets given by their Hasse diagrams in Fig. 1 show that the notions of defect optimality and greediness generally are different.

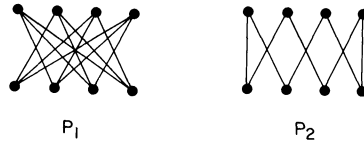


FIG. 1

It is easily verified that P_1 is (strongly) greedy but not defect optimal with respect to any field whereas P_2 is defect optimal but not greedy. In spite of this example the notions of defect optimality and (strong) greediness are closely related. We will clarify this relation in the subsequent sections.

3. Composition. We now investigate two compositions of ordered sets and show that both the class of defect optimal sets and the class of strongly greedy ordered sets are closed under these compositions. The first composition is substitution, the second generalizes a construction in [7].

Let P_1 and P_2 be two ordered sets on disjoint ground sets E_1 and E_2 . Fix an element $a \in E_1$. We define a new partially ordered set (P_1, a, P_2) as follows:

- (I) The ground set of (P_1, a, P_2) is given by $E = (E_1 \setminus \{a\}) \cup E_2$.
- (II) The order relation \leq on (P_1, a, P_2) is given by $x \leq y$ if one of the following cases holds:
 - (i) $x, y \in E_1$ and $x \leq y$ in P_1 ,
 - (ii) $x, y \in E_2$ and $x \leq y$ in P_2 ,
 - (iii) $x \leq a$ in P_1 and $y \in E_2$,
 - (iv) $a \leq y$ in P_1 and $x \in E_2$.

The ordered set (P_1, a, P_2) is called the *substitution* of P_2 in P_1 at a .

Note that (P_1, a, P_2) is a special case of lexicographical sums. As a matter of fact, lexicographical sums can be obtained by successively applying substitutions. Especially, if P_1 and P_2 are defect optimal, then (P_1, a, P_2) is defect optimal (see [8, 4.12]). The next result shows that substitutions behave similarly with respect to the property of being strongly greedy. First of all, however, we need a technical lemma.

Let us assume that L is a linear extension of (P_1, a, P_2) . We then can define linear extensions $L(P_1)$ of P_1 and $L(P_2)$ of P_2 as follows:

$L(P_1)$ is the restriction of L to P_1 , where we replaced the first element of E_2 occurring in L by a .

$L(P_2)$ is the restriction of L to P_2 .

LEMMA 3.1. *Let L be an arbitrary linear extension of (P_1, a, P_2) . Then $g(L) = g(L(P_1)) + g(L(P_2))$.*

Proof. First of all, let (a_i, a_{i+1}) be a greedy setup of L . Then for every upper neighbor b of a_i there is a lower neighbor c of b such that $c \notin L_i = a_1 \cdots a_i$.

Assume that $a_i \in E_1$ and assume that a_i is not a lower neighbor of a . Then every upper neighbor of a_i belongs to E_1 and so does every lower neighbor of every upper neighbor. If also $a_{i+1} \in E_1$, then clearly (a_i, a_{i+1}) is a greedy setup of $L(P_1)$. Let us assume that $a_{i+1} \in E_2$. Let $j > i$ be the smallest index such that $a_j \in E_1$. Let

$$a'_j = \begin{cases} a & \text{if } i+1 \text{ is the smallest index } k \text{ such that } a_k \in E_2, \\ a_j & \text{otherwise.} \end{cases}$$

Then a'_j follows a_i in $L(P_1)$. In order to show that (a_i, a'_j) is a greedy setup of $L(P_1)$ it is enough to show that $a_i \not\leq a'_j$. Assume that we had $a_i < a'_j$. Pick an upper neighbor b of a_i which is bounded by a'_j , i.e., $b \leq a'_j$. If a_k is a lower neighbor of b , then (1) $a_k < a'_j$ and (2) $a_k \in E_1$, thus $k < j$. Since the elements $a_{i+1}, \dots, a_{j-1} \in E_2$, we conclude that $k \leq i$. Hence we have found an upper neighbor of a_i , namely b , such that for every lower neighbor c of b we have $c \in L_i$, contradicting the fact that (a_i, a_{i+1}) is a greedy setup.

Next, assume that $a_i \in E_1$ and that a_i is a lower neighbor of a . Let m be a minimal element of P_2 . Then a_i is a lower neighbor of m in (P_1, a, P_2) . Hence there is a lower neighbor c of m in (P_1, a, P_2) such that $c \notin L_i$. Clearly, c is a lower neighbor of a in P_1 . Hence we conclude: For every upper neighbor b of a_i in P_1 there is a lower neighbor c of b in P_1 such that $c \notin L_i$. Let again $j > i$ be the smallest index such that $a_j \in E_1$. Define a'_j as before. Again, it can be shown that $a_i \not\leq a'_j$ and therefore (a_i, a'_j) is a greedy jump of $L(P_1)$.

We now assume that $a_i \in E_2$. Again, we consider the cases $a_{i+1} \in E_1$ and $a_{i+1} \in E_2$. Let us assume that $a_{i+1} \in E_1$. Let $j > i$ be the smallest index such that $a_j \in E_2$. Then a_j follows a_i in $L(P_2)$. We first establish the fact that there is indeed a setup between a_i and a_j in $L(P_1)$. Assume not. Pick an upper neighbor b of a_i such that $b \leq a_j$. Then $b = a_k$ for a certain k and hence $k \leq j$. We conclude that $k = j$, i.e., a_i is a lower neighbor of a_j . Thus a_j cannot be minimal in P_2 and therefore every lower neighbor of a_j in (P_1, a, P_2) belongs to E_2 . Let a_l be any lower neighbor of a_j . Then $a_l < a_j$ and hence $l < j$. If $i < l$, we would get a contradiction to the choice of j . Hence we obtain $l \leq i$. Hence every lower neighbor c of a_j belongs to L_i , contradicting the fact that we have a greedy setup at a_i . If now b is any upper neighbor of a_i in P_2 , then again b cannot be minimal in P_2 . Hence every lower neighbor of b in (P_1, a, P_2) belongs to E_2 . From the fact that we have a greedy setup at a_i we conclude that there is a lower neighbor $c \in E_2$ of b such that $c \notin L_i$. Hence (a_i, a_j) is a greedy setup of $L(P_2)$.

Let us now assume that $a_i, a_{i+1} \in E_2$. A slight modification of the arguments given in the last paragraph shows that (a_i, a_{i+1}) is also a greedy setup of $L(P_2)$.

These arguments show that

$$g(L) \leq g(L(P_1)) + g(L(P_2)).$$

We now launch a proof of the reverse inequality: Let (x, y) be a greedy setup of $L(P_1)$. We have to consider three cases:

$x \neq a \neq y$. Then $x = a_i$ and $y = a_j$ for certain indices $i < j$. If $j = i + 1$, then $(x, y) = (a_i, a_{i+1})$ is a greedy setup of L : Indeed, let $b \in E_1$ be an upper neighbor of a_i with respect to (P_1, a, P_2) . Then, since (a_i, a_j) is a greedy setup of $L(P_1)$, there is a lower neighbor $c \in E_1$ of b such that $c \notin L_i$. The only complication occurs in the case where c is not a lower neighbor of b in (P_1, a, P_2) . This can only happen if either $b = a$ or $c = a$. If $b = a$, then substitute b by a minimal element of P_2 . This minimal element

will then be an upper neighbor of a_i which has c as a lower neighbor. Assume then that $c = a$. In this case, we substitute c by a maximal element m of P_2 . Then $m \notin L_i$, because otherwise $a = c$ would have to be listed before a_i in $L(P_1)$ by the definition of $L(P_1)$, contradicting the choice of c .

Now assume that $j > i + 1$. In this case $a_{i+1} \in E_2$. We show that (a_i, a_{i+1}) is a setup of L . Assume not. In this case a_i would be a lower neighbor of a_{i+1} in (P_1, a, P_2) . But then a_i is a lower neighbor of a and by the construction of $L(P_1)$, a would have to be listed immediately after a_i in $L(P_1)$. This would imply $a_j = a > a_i$, contradicting the fact that we have a setup at (a_i, a_j) . Now a slight modification of the argument given above for $j = i + 1$ shows that the setup (a_i, a_{i+1}) is in fact greedy.

Consider now the case where $y = a$. Assume that $x = a_i$. In this case a_{i+1} would have to be a minimal element of P_2 , and, since $x \not\leq y$, we have $a_i \not\leq a_{i+1}$. Again we see that (a_i, a_{i+1}) is a greedy setup.

If $x = a$, let $y = a_{i+1}$. In this case the element $a_i \in E_2$. Moreover, since $a = x \not\leq y = a_{i+1}$, we have a setup at (a_i, a_{i+1}) . We verify that this is a greedy setup: First of all, we can pick an upper neighbor b of $x = a$ and a lower neighbor c of b , all belonging to E_1 , such that c is listed after x in $L(P_1)$. We would like to show that c is listed after a_i in L . To this end, let k be the smallest index such that $a_k \in E_2$. Then $L(P_1)$ looks like

$$L(P_1) = a_1 a_2 \cdots a_{k-1} (a = x) (y = a_{i+1}) \cdots$$

Since all the elements a_k, \dots, a_i have to belong to E_2 , and since c is not among a_1, \dots, a_{k-1}, a , we conclude that $c = a_j$ where $j > i$.

Now let (a_i, a_j) be a greedy setup of $L(P_2)$. If $j = i + 1$, then obviously (a_i, a_j) would have to be a greedy setup of L . Hence let us assume that $j > i + 1$. In this case $a_{i+1} \in E_1$. Assume, if possible, that $a_i < a_{i+1}$. Then $a_{j+1} > a$ and therefore a_{i+1} is an upper bound in (P_1, a, P_2) for every $x \in E_2$. Especially, $a_j < a_{i+1}$, contradicting the fact that $i + 1 < j$. Hence we have a setup at (a_i, a_{i+1}) . It is readily verified that this setup is a greedy setup.

We are now allowed to conclude that $g(L) \geq g(L(P_1)) + g(L(P_2))$ and therefore equality holds. \square

THEOREM 3.2. *If P_1 and P_2 are strongly greedy, then so is $P = (P_1, a, P_2)$.*

Proof. Let L_G be an optimal greedy extension of P , and let L be an arbitrary linear extension of P . We use the lemma and the fact that by assumption $s(P_1) = g(L(P_1))$, $s(P_2) = g(L(P_2))$, in order to compute

$$\begin{aligned} s(P) &= g(L_G) \\ &= g(L_G(P_1)) + g(L_G(P_2)) \\ &= s(P_1) + s(P_2) \\ &= g(L(P_1)) + g(L(P_2)) \\ &= g(L). \end{aligned}$$

Hence P is strongly greedy. \square

It is easy to see that in fact the converse of Theorem 3.2 must also be true. Let us note that a substitution decomposition of an ordered set can always be carried out efficiently (cf. Buer and Möhring [2]).

The following corollary deals with lexicographical sums. For the definition of lexicographical sums, we refer the reader to [8].

COROLLARY 3.3. *Let S be a strongly greedy ordered set, and let $(P_s)_{s \in S}$ be a family of strongly greedy ordered sets. Then the lexicographical sum of the family $(P_s)_{s \in S}$ is again strongly greedy.*

We now turn to another construction, which is very closely related to substitution.

Let P and Q be ordered sets such that $P \cap Q = \emptyset$, let $A \subset P$, and let $C \subset Q$. We define the ordered set $P * Q$ with respect to A and C as follows:

The ground set of $P * Q$ is the disjoint union of P and Q .

- (i) $x, y \in P$ and $x \leq y$ in P ,
- (ii) $x, y \in Q$ and $x \leq y$ in Q ,
- (iii) $x \in P, y \in Q$ and $x \leq a, c \leq y$ for certain elements $a \in A, c \in C$.

Several special cases of this construction are already known:

- (a) If $A = \emptyset$ or $C = \emptyset$, then $P * Q$ is the parallel composition of P and Q .
- (b) If A is the set of all maximal elements of P and if C is the set of all minimal elements of Q , then $P * Q$ is the linear sum of P and Q .
- (c) Bipartite sums in the sense of [7] are also a special case of our present construction.

In this paper, we would like to consider two other special cases of $P * Q$:

- (d) If there is an element $a \in P$ such that A is the set of all lower neighbors of a or if A consists only of maximal elements of P while C consists only of minimal elements of Q , then $P * Q$ is called a *weakly linear sum* of P and Q . Weakly linear sums will be denoted as $P \oplus Q$.

For the second case, we need a little bit of preparation: Let us agree to call a set $A \subset P$ to be *submaximal* if $A = B \cup I$ where I consists only of isolated points of P and where either (i) B contains only maximal elements of P or (ii) no element of B is maximal, every upper neighbor of an element of B is maximal and every lower neighbor of an upper neighbor of an element of B belongs to B again.

- (e) If $A \subset P$ is submaximal and if $C \neq \emptyset$ consists of minimal elements of Q only, then $P * Q$ is called an *upper bipartite sum* of P and Q .

LEMMA 3.4. (i) *We always have $s(P) + s(Q) \leq s(P * Q) \leq s(P) + s(Q) + 1$.*

- (ii) *If $P * Q$ is an upper bipartite sum and if A contains a maximal element of P , then $s(P) + s(Q) = s(P * Q)$.*

- (iii) *If $P * Q$ is an upper bipartite sum, and if A contains no maximal element of P , then $s(P * Q) = s(P) + s(Q) + 1$.*

Proof. (i): Let $L(P)$ be an optimal schedule of P and let $L(Q)$ be an optimal schedule of Q . Then $L(P) + L(Q)$ is a schedule of $P * Q$ and we have $s(P * Q) \leq s(L(P) + L(Q)) \leq s(L(P)) + s(L(Q)) + 1 = s(P) + s(Q) + 1$.

For the other inequality, let L be an optimal schedule of $P * Q$ with chain decomposition $C_1 + \dots + C_k$. In this case we have $s(P * Q) = k - 1$. By the definition of $P * Q$ there is at most one chain C_i such that $C_i \cap P \neq \emptyset \neq C_i \cap Q$. Indeed, let $x, y \in C_i$ be such that x is a lower neighbor of $y, x \in P$ and $y \in Q$. Then x must belong to A and y must belong to C . Moreover, since y is an upper bound of all elements of A , all the other elements of A must occur in one of C_1, \dots, C_{i-1} , or, in case A is not an antichain, in the part of C_i before x . Clearly, this can happen only once. If there is a chain C_i which contains elements of P and of Q , split this chain in halves: $C'_i = C_i \cap P$ and $C''_i = C_i \cap Q$. From these (new) chains, let D_1, \dots, D_m be the chains contained in P and let E_1, \dots, E_k be the chains contained in Q . Then $m + k \leq n + 1$ and, since $D_1 + \dots + D_m$ is a linear extension of $P, s(P) \leq m - 1$. Similarly, $s(Q) \leq k - 1$. This gives $s(P) + s(Q) \leq m - 1 + k - 1 \leq n - 1 = s(P * Q)$.

- (ii) Observe that there is an optimal linear extension $L(P)$ of P such that the last element \bar{a} of A occurring in $L(P)$ is maximal in P . This is obvious in the case

where A contains an isolated element, since isolated elements give rise to singleton chains in the chain decomposition of $L(P)$ which may be switched to every arbitrary position. On the other hand, if A does not contain any isolated point, then A consists of maximal elements only and the assertion is clear in this case, too. We now decompose $L(P)$ into $L_1(P)$ and $L_2(P)$ such that the last element of $L_1(Q)$ is \bar{a} . Similarly every optimal linear extension $L(Q)$ of Q splits into $L_1(Q)$ and $L_2(Q)$ where the first element of $L_2(Q)$ is the first element of C occurring in $L(Q)$. Then $L = L_1(Q)L_1(P)L_2(Q)L_2(P)$.

(iii) To see (iii) we claim that there exists an optimal linear extension L of $P * Q$ such that every chain induced by the setups either lies completely in P or completely in Q . Since in this case the induced chains yield linear extensions $L(P)$ of P and $L(Q)$ such that $s(P * Q) = s(L(P)) + s(L(Q)) + 1$. Now (i) implies the first inequality in $s(L(P)) + s(L(Q)) + 1 \leq s(P) + s(Q) + 1 \leq s(L(P)) + s(L(Q)) + 1$, hence (iii) will follow.

So let L be any optimal schedule of $P * Q$ with chain decomposition $C_1 + \dots + C_k$. As we already remarked in the proof of (i), there is at most one chain C_i which intersects both P and Q . Let $a \in A$ be the last element of P in C_i and split C_i into C'_i and C''_i as before. Now note that every upper neighbor of a in P must form a chain C_j in the chain decomposition of L . Hence combining C'_i and C_j to a new chain yields a linear extension of $P * Q$ with the desired properties. \square

PROPOSITION 3.5. *Let K be any field and let $P \oplus Q$ be a weakly linear sum of P and Q with respect to A and C .*

(i) *If $A \neq \emptyset \neq C$ and if A contains only maximal elements of P , C contains only minimal elements of Q , then $\text{def}_K(P \oplus Q) = \text{def}_K(P) + \text{def}_K(Q) - 1$.*

(ii) *If $A = \emptyset$ or if $C = \emptyset$ or if there is an element $b \in P$ such that $A = N^-(b)$, then $\text{def}_K(P \oplus Q) = \text{def}_K(P) + \text{def}_K(Q)$.*

Proof. Let us show (i) first. It suffices to verify that $\text{rk}(P \oplus Q) = \text{rk}(P) + \text{rk}(Q) + 1$. To this end, we index the incidence matrix $M(P \oplus Q)$ by a linear extension $p_1 p_2 \dots p_n q_1 q_2 \dots q_m$ of $P \oplus Q$, where $p_1 \dots p_n$ is a linear extension of P , $q_1 \dots q_m$ is a linear extension of Q and $p_n \in A, q_1 \in C$.

Let S be the $(m+1) \times (m+1)$ -submatrix of the incidence matrix $M(P \oplus Q)$ determined by the rows and columns indexed p_n, q_1, \dots, q_m . We claim $\text{rk}(S) = \text{rk}(M(Q)) + 1$. But this is easily verified by making use of the fact that the p_n th column is zero and the q_1 st column is just the unit vector.

Furthermore, let T be the $(n-1) \times (n-1)$ -submatrix given by the rows and columns with indices p_1, \dots, p_{n-1} . Since p_n is maximal in P , we obtain $\text{rk}(T) = \text{rk}(M(P))$.

Hence it remains to show that $\text{rk}(M(P \oplus Q)) = \text{rk}(S) + \text{rk}(T)$.

The matrix $M(P \oplus Q)$ has the following form:

$$M(P \oplus Q) = \begin{pmatrix} T & R \\ 0 & S \end{pmatrix}.$$

Note that the p_i th row of R is either 0 (in case p_i is not bounded by any element of A) or it has 1's exactly at those columns q_j for which q_j is an upper bound of an element of C . Hence all the nonzero rows of R agree with the p_n th row of $M(P \oplus Q)$. Subtracting this p_n th row from all the nonzero rows of R does not change the rank of $M(P \oplus Q)$ and transforms this matrix into

$$\begin{pmatrix} T & 0 \\ 0 & S \end{pmatrix}.$$

This shows that $\text{rk}(M(P \oplus Q)) = \text{rk}(S) + \text{rk}(T)$.

(ii) The case where $A = \emptyset$ or $C = \emptyset$ is straightforward. We only consider the case where $A = N^-(b)$ for a certain $b \in P$.

We define two linear mappings $f, g: K^{P \oplus Q} \rightarrow K^{P \oplus Q}$ in the following way: The function f is the “incidence function” of $P \oplus Q$, i.e.,

$$f(v)(y) = \sum_{x < y} v(x).$$

The function g is the sum of the “incidence function” g_1 of P and the “incidence function” g_2 of Q . Here we have to identify $K^{P \oplus Q}$ with the direct sum $K^P \oplus K^Q$ of the vector spaces K^P and K^Q :

$$g(v)(y) = \begin{cases} \sum_{x < y, x \in P} v(x) & \text{if } y \in P, \\ \sum_{x < y, x \in Q} v(x) & \text{if } y \in Q. \end{cases}$$

We will show that $\text{im}(f) = \text{im}(g)$. Let $z \in P \oplus Q$. Then the characteristic function $v_z \in K^{P \oplus Q}$ of z is a typical “unit vector.” We have to consider three cases:

(1) $z \in P$. Then $f(v_z) = g(v_z)$.

(2) $z \in Q$ and $c \not\leq z$ for all $c \in C$. In this case we also have $f(v_z) = g(v_z)$.

(3) $z \in Q$ and $c \leq z$ for a certain $c \in C$. Let b be such that $A = N^-(b)$ and let v_b be its characteristic function. Then for any $y \in P$ we have that $y < z$ if and only if $y < b$. This yields

$$f(v_z) = g(v_z) + g(v_b)$$

and, since we already know from case (1) that $g(v_b) = f(v_b)$,

$$g(v_z) = f(v_z) - f(v_b).$$

Since the unit vectors span the whole vector space, we conclude from these three cases that $\text{im}(f) = \text{im}(g)$. Thus $\dim(\text{im}(f)) = \dim(\text{im}(g)) = \dim(\text{im}(g_1)) + \dim(\text{im}(g_2))$ or $\text{rk}(P \oplus Q) = \text{rk}(P) + \text{rk}(Q)$, and the corresponding equality for the defect follows. \square

COROLLARY 3.6. *The class of defect optimal ordered sets is closed under forming weakly linear sums.*

Proof. Let P and Q be defect optimal ordered sets and let $P \oplus Q$ be a weakly linear sum with respect to $A \subset P$ and $C \subset Q$.

If $A = \emptyset$ or $C = \emptyset$, then the claim follows from [8, Prop. 4.12].

If A contains only maximal elements and if C contains only minimal elements, then from Lemma 3.4 we know that $s(P \oplus Q) = s(P) + s(Q)$. Since by (3.5(i)) we also have $\text{def}_K(P \oplus Q) - 1 = \text{def}_K(P) - 1 + \text{def}_K(Q) - 1$, it follows that $s(P \oplus Q) = \text{def}_K(P \oplus Q) - 1$.

Finally, let us assume that there is an element $b \in P$ such that $A = N^-(b)$. Then, by (3.5(ii)) we have

$$\begin{aligned} \text{def}_K(P \oplus Q) - 1 &= (\text{def}_K(P) - 1) + (\text{def}_K(Q) - 1) + 1 \\ &= s(P) + s(Q) + 1. \end{aligned}$$

Since we always have $\text{def}_K(P \oplus Q) - 1 \leq s(P \oplus Q) \leq s(P) + s(Q) + 1$, the statement is also true in this case. \square

Let us remark a few interesting facts: The proof of Theorem 2 in [7] can without problems be adjusted to our present situation of weakly linear sums. In this case, we obtain the result that *a partially ordered set P is N -free if and only if P can be constructed from the empty set by a sequence of weakly linear sums with singletons* (allowing only

series or parallel compositions gives us the class of series-parallel ordered sets). If we connect this result with our last corollary, we obtain:

COROLLARY 3.7. *Every N -free ordered set is defect optimal.*

Note that the description of N -free ordered sets as a weakly linear sum of singletons gives an easy procedure to algorithmically test whether P is N -free: we successively take away maximal elements which must have been added by a weakly linear sum. We will return to this idea in the next paragraph.

The next proposition in connection with the previous results again exhibits the close analogous behavior of the defect of a partially ordered set and the property of being strongly greedy.

PROPOSITION 3.8. *Let P and Q be partially ordered sets. Then a weakly linear sum $P \oplus Q$ is strongly greedy if and only if both P and Q are strongly greedy.*

Proof. The proof is analogous to the proof of [7, Thm. 1] and is a modification of the proof of Theorem 3.2. \square

4. An algorithm for upper bipartite sums. The preceding section shows that the class of ordered sets generated by singletons under weakly linear sums can be efficiently recognized by an algorithm (see the remark following (3.7)). It will follow from the results in this section that this algorithm can be modified to simultaneously construct an optimal linear extension if the ordered set is N -free. However, to obtain more generality we will discuss upper bipartite sums instead of weakly linear sums.

Let \mathcal{B} be the class of finite ordered sets constructed by sequentially forming upper bipartite sums with singletons. We will see that members of this class can also be recognized algorithmically and the optimization problem can be solved efficiently for every ordered set in this class.

THEOREM 4.1. *Let $P \in \mathcal{B}$. If P is an upper bipartite sum $P = P_1 * y$ of P_1 and a singleton, then $P_1 \in \mathcal{B}$.*

Proof. The proof is by induction on $|P|$. By hypothesis there exists $\bar{P} \in \mathcal{B}$ and a (maximal) element $x \in P$ such that P is the upper bipartite sum $\bar{P} * x$ of \bar{P} and x . If $x = y$ there is nothing to prove. Assume $x \neq y$ and let $A_x = N^-(x)$ and $A_y = N^-(y)$.

We claim that either $A_x = A_y$ or $A_x \cap A_y = \emptyset$. Suppose $A_x \cap A_y \neq \emptyset$, then $y \in N^+(A_x)$ and $x \in N^+(A_y)$ hence $A_x = A_y$ by the definition of upper bipartite sums. But now in either case $\bar{P} = P' * y$ where by induction hypothesis $P' \in \mathcal{B}$, and where $*$ represents an upper bipartite sum. Clearly, this implies $P = P' * x * y$, i.e., $P_1 = P' * x \in \mathcal{B}$. \square

Based on the above theorem and Lemma 3.4 we now describe an algorithm which tests membership in the class \mathcal{B} and which for $P \in \mathcal{B}$ computes the setup number $s(P)$.

UBS ALGORITHM 4.2.

Input: A finite ordered set P .

Output: The setup number $s(P)$ of P or the information $P \notin \mathcal{B}$.

1. $s \leftarrow 0$;
2. If $P = \emptyset$, STOP. In this case, $P \in \mathcal{B}$ and $s = s(P)$;
3. Choose a maximal element $a \in P$ such that $P = \bar{P} * a$ is an upper bipartite sum. If no such a exists, STOP. In this case, $P \notin \mathcal{B}$;
4. If $N^-(a)$ contains no maximal element of \bar{P} , then $s \leftarrow s + 1$;
5. $P \leftarrow \bar{P}$;
6. GOTO 2.

Let $\mathcal{B}_0 \subset \mathcal{B}$ consist of those ordered sets P such that in the construction of P the set A contains no isolated elements. We call those sums *nonsingular upper bipartite sums*.

PROPOSITION 4.3. For every $P \in \mathcal{B}_0$, P is strongly greedy.

Proof. In view of Lemma (3.4) the proof may be carried out similarly to the proof of Theorem (3.2) by a straightforward induction on $|P|$. \square

PROPOSITION 4.4. If P is N -free, then $P \in \mathcal{B}_0$.

Proof. Choose $a \in P$ such that $B = N^+(a)$ contains only maximal elements and observe that for every $b \in B$ and $a' \in N^-(b) = A$, $N^+(a')$ contains only maximal elements (otherwise P would not be N -free). Moreover $N^-(b') = A$ for every $b' \in B$ and $N^+(a') = B$ for every $a' \in A$. Hence $P = (P - b) * b$ and this sum is a nonsingular upper bipartite sum. Noting that A contains maximal elements of $P - b$ if and only if $|B| = 1$ the proposition follows by induction on $|P|$. \square

We remark that \mathcal{B} strictly contains \mathcal{B}_0 and that there are strongly greedy ordered sets in \mathcal{B}_0 which are not N -free.

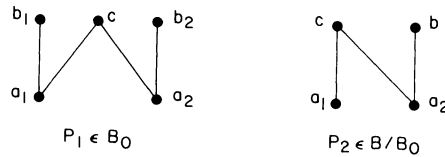


FIG. 2

Note that Algorithm UBS above can be modified as to simultaneously construct an optimal linear extension if P is N -free: always choose, if possible, the element a such that a is a lower neighbor of a' , where a' was the preceding element chosen.

We leave it to the reader to verify that the class \mathcal{B} is closed under upper bipartite sums and that the class \mathcal{B}_0 is closed under nonsingular upper bipartite sums.

5. Series-parallel classes. Our investigation so far has concentrated on classes of ordered sets which can be decomposed to singletons and for which the setup problem can be solved. We will now give a construction principle for classes of ordered sets generated by “irreducibles” with respect to series-parallel composition such that the setup optimization problem can efficiently be dealt with provided the setup numbers of the irreducibles are known. Recall that series-parallel compositions are special weakly linear sums where A is either empty or A contains all maximal elements and C contains all minimal elements. The cycle-series-parallel ordered sets of [8] constitute an example of a special such class.

An ordered set P is (series-parallel) *irreducible* if P cannot be expressed as a nontrivial series or parallel composition of two other ordered sets. Let I be a class of irreducible ordered sets and let $\mathcal{SP} = \mathcal{SP}(I)$ be the class of finite ordered sets generated by taking successively series or parallel composition with members in I .

The first observation is immediate; \mathcal{SP} is closed under series and parallel composition. The following result is fundamental.

THEOREM 5.1. If $P \in \mathcal{SP}$ is either a series or a parallel composition of P_1 and P_2 , then both P_1 and P_2 belong to \mathcal{SP} .

Proof. We discuss only the case where P is a series composition of P_1 and P_2 . The case where P is a parallel composition is handled similarly. The proof is by induction on $|P|$: Since $P \in \mathcal{SP}$ and since every series composition leads to a connected ordered set, P is a series composition of a certain $\bar{P} \in \mathcal{SP}$ and an irreducible $Q \in I$. Since Q is irreducible, we have either $Q \subset P_1$ or $Q \subset P_2$, since otherwise we could write Q as a series composition of $Q \cap P_1$ and $Q \cap P_2$. W.l.o.g. we assume that $Q \subset P_1$. In this case, we can write P_1 as series composition of Q and a certain P' . Hence \bar{P} is a series composition of P_2 and P' . Since $|P| > |\bar{P}|$, the induction hypothesis yields that

P_2 and P' belong to \mathcal{SP} . Hence the series composition P_1 of P' and Q belongs to \mathcal{SP} , too. \square

Theorem 5.1 suggests the following algorithmic procedure to test whether the ordered set P belongs to $\mathcal{SP}(I)$:

1. If the Hasse diagram of P is not connected then we obtain a parallel decomposition from the connected components.

2. If the Hasse diagram of P is connected, choose a maximal chain $K = a_1 < a_2 < \dots < a_n$ in P . For each $a \in K$ choose $b \in N^+(a)$ and test whether $N^-(b)$ and $N^+(a)$ induce a complete bipartite subgraph in the Hasse diagram of P . Moreover, test whether for each $x \in P$ either $x < b$ or $x > a$. If no such a exists, P is irreducible and $P \in \mathcal{SP}(I)$ if and only if $P \in I$. Otherwise $N^-(b)$ and $N^+(a)$ give rise to a series decomposition of P .

Note that the algorithm is valid. Indeed, if $P = P_1 + P_2$ is a series composition, then every maximal element of P_1 must have the properties required in Step 2 of the algorithm.

A more direct decomposition algorithm exists for special cases I . If no maximal element of any irreducible set properly dominates all minimal elements we let in step 2 the ordered set P_2 consist of all elements of P which dominate all minimal elements of P . This includes the cases considered in [8] and [17].

In general, the algorithm above decomposes every $P \in \mathcal{SP}(I)$ into irreducibles in I . By Lemma 3.4 the setup number $s(P)$ can be computed efficiently from this decomposition tree.

REFERENCES

- [1] G. BIRKHOFF (1967), *Lattice theory*, AMS Colloquium Publications, vol. 25, 3rd ed., American Mathematical Society, Providence, RI.
- [2] H. BUER AND R. H. MÖHRING (1983), *A fast algorithm for the decomposition of graphs and posets*, Math. Oper. Res., 8, pp. 170-184.
- [3] G. CHÂTY AND M. CHEIN (1979), *Ordered matchings and matchings without alternating cycles in bipartite graphs*, Util. Math., 16, pp. 183-187.
- [4] O. COGIS AND M. HABIB (1979), *Nombre de sauts et graphes série-parallèles*, RAIRO Inform. Théor., 13, pp. 3-18.
- [5] R. P. DILWORTH (1950), *A decomposition theory for partially ordered sets*, Ann. Math., 51, pp. 161-166.
- [6] D. DUFFUS, I. RIVAL AND P. WINKLER (1982), *Minimizing setups for cycle-free ordered sets*, Proc. Amer. Math. Soc., to appear.
- [7] U. FAIGLE AND G. GIERZ (1983), *A construction for strongly greedy ordered sets*, preprint, Proc. 8th Symposium and Operations Research, Karlsruhe, to appear.
- [8] G. GIERZ AND W. POGUNTKE (1983), *Minimizing setups for ordered sets: a linear algebraic approach*, SIAM J. Alg. Disc. Meth., 4, pp. 132-144.
- [9] M. C. GOLUBIC (1980), *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York.
- [10] P. GRILLET (1969), *Maximal chains and antichains*, Fund. Math., 65, pp. 157-167.
- [11] C. HUCHENNE (1964), *Sur une certaine correspondance entre graphes*, Bull. Soc. Roy. Sci. Liège, 33, pp. 743-753.
- [12] B. LECLERC AND B. MONJARDET (1973), *Orders "C.A.C."*, Fund. Math., 79, pp. 11-22.
- [13] W. R. PULLEYBLANK (1981), *On minimizing setups in precedence constrained scheduling*, preprint, Discr. Appl. Math., to appear.
- [14] I. RIVAL (1982), *Optimal linear extensions by interchanging chains*, preprint; Proc. Amer. Math. Soc., 89 (1983), pp. 387-394.
- [15] E. SZPILRAJN (1930), *Sur l'extension de l'ordre partiel*, Fund. Math., 16, pp. 386-389.
- [16] M. M. SYSLO (1983), *Minimizing the jump number for ordered sets: a graph theoretic approach*, preprint, Order, to appear.
- [17] I. VALDES, R. E. TARIAN AND E. L. LAWLER (1979), *The recognition of series-parallel digraphs*, Proc. 11th Annual ACM Symposium on the Theory of Computing, pp. 1-12; this Journal, 11 (1982), pp. 298-313.

AN EXTENSION OF LIOUVILLE'S THEOREM ON INTEGRATION IN FINITE TERMS*

M. F. SINGER†, B. D. SAUNDERS‡ AND B. F. CAVINESS§

Abstract. In Part I of this paper, we give an extension of Liouville's Theorem and give a number of examples which show that integration with special functions involves some phenomena that do not occur in integration with the elementary functions alone. Our main result generalizes Liouville's Theorem by allowing, in addition to the elementary functions, special functions such as the error function, Fresnel integrals and the logarithmic integral (but not the dilogarithm or exponential integral) to appear in the integral of an elementary function. The basic conclusion is that these functions, if they appear, appear linearly. We give an algorithm which decides if an elementary function, built up using only exponential functions and rational operations has an integral which can be expressed in terms of elementary functions and error functions.

Key words. Liouville's theorem, integration in finite terms, special functions, error function

Introduction. In 1969 Moses [MOSE69] first raised the possibility of extending the Risch decision procedure for indefinite integration to include a certain class of special functions. Some of his ideas have been incorporated as heuristic methods in MACSYMA and REDUCE. However, little progress has been made on the theory necessary to extend the Risch algorithm. One step in this direction was the paper by Moses and Zippel [MOZI79] in which a weak Liouville Theorem was given for special functions (this result also appears in [SING77]).

In Part I of this paper, we give an extension of Liouville's Theorem [RISC69, p. 169] and give a number of examples which show that integration with special functions involves some phenomena that do not occur in integration with the elementary functions alone. Our main result generalizes Liouville's Theorem by allowing, in addition to the elementary functions, special functions such as the error function, Fresnel integrals and the logarithmic integral (but not the dilogarithm or exponential integral) to appear in the integral of an elementary function. The basic conclusion is that these functions, if they appear, appear linearly.

In Part II of this paper, we use the results of Part I to examine the question of when the integral of an elementary function can be expressed in terms of elementary functions and error functions. We give an algorithm which decides if an elementary function, built up using only exponential functions and rational operations has an integral which can be expressed in terms of elementary functions and error functions.

Some of the results of this paper have been announced in [SSC81]. We wish to thank Barry Trager for drawing our attention to Example 2.1 in § 2.

Finally, all fields in this paper are assumed to be of characteristic 0. \mathbb{C} , \mathbb{Q} and \mathbb{Z} stand for the complex numbers, rational numbers, and integers respectively.

I. An extension of Liouville's Theorem.

1. Statement and discussion of results. We begin by defining a generalization of the elementary functions. Let F be a differential field of characteristic 0 with derivation

* Received by the editors March 19, 1982, and in final revised form July 5, 1984.

† Department of Mathematics, North Carolina State University, Raleigh, North Carolina 27650.

‡ Department of Mathematical Sciences, Rensselaer Polytechnic Institute, Troy, New York, 12181. The work of this author was supported in part by the National Science Foundation under grant MCS79-09158.

§ Department of Computer and Information Sciences, University of Delaware, Newark, Delaware 19711. The work of this author was supported in part by the National Science Foundation under grant MCS79-09158.

and constants C . Let A and B be finite indexing sets and let

$$\mathcal{E} = \{G_\alpha(\exp R_\alpha(Y))\}_{\alpha \in A},$$

$$\mathcal{L} = \{H_\beta(\log S_\beta(Y))\}_{\beta \in B},$$

be sets of expressions where:

- (1) $G_\alpha, R_\alpha, H_\beta, S_\beta$ are in $C(Y)$ for all $\alpha \in A, \beta \in B$, i.e. they are all rational functions with constant coefficients;
- (2) for all $\beta \in B$, if $H_\beta(Y) = P_\beta(Y)/Q_\beta(Y)$ with P_β, Q_β in $C[Y]$, then $\deg P_\beta \leq \deg Q_\beta + 1$.

We say that a differential extension E of F is an \mathcal{EL} -elementary extension of F if there exists a tower of fields $F = F_0 \subset F_1 \subset \dots \subset F_n = E$ such that $F_i = F_{i-1}(\theta_i)$ where for each $i, 1 \leq i \leq n$, one of the following holds:

- (i) θ_i is algebraic over F_{i-1} ;
- (ii) $\theta_i' = u' \theta_i$ for some $u \in F_{i-1}$;
- (iii) $\theta_i' = u'/u$ for some nonzero $u \in F_{i-1}$;
- (1.1) (iv) for some $\alpha \in A$, there are $u, v \in F_{i-1}$ such that $\theta_i' = u' G_\alpha(v)$ where $v' = (R_\alpha(u))' v$;
- (v) for some $\beta \in B$, there are u, v in F_{i-1} such that $\theta_i' = u' H_\beta(v)$ where $v' = (S_\beta(u))'/S_\beta(u)$ and $S_\beta(u) \neq 0$.

Informally, we could write (1.1) cases (ii)-(v) as

- (ii') $\theta_i = \exp u$;
- (iii') $\theta_i = \log u$;
- (iv') $\theta_i = \int u' G_\alpha(\exp R_\alpha(u)) dx$;
- (v') $\theta_i = \int u' H_\beta(\log S_\beta(u)) dx$.

Cases (ii)-(iv) and (ii')-(iv') are not equivalent since, for example, (ii) determines θ_i up to a multiplicative constant while (ii') refers to a specific function, \exp . Although this distinction is not usually emphasized in the standard Liouville Theorem, it is not a pedantry here. The distinction between (iv)-(v) and (iv')-(v') is crucial to prevent transcendental constants from being introduced by integration. This will be discussed in detail in § 2.

The definition of \mathcal{EL} -elementary functions is broad enough to include such functions as the error function, the Fresnel integrals and the logarithmic integral. Let $F = C(x), C$ the complex numbers. The error function is defined by

$$\operatorname{erf}(u) = \int u' e^{-u^2} dx$$

where $G_\alpha(\exp R_\alpha(Y)) = \exp(-Y^2)$ with $G_\alpha(W) = W$ and $R_\alpha(Y) = -Y^2$.

The Fresnel integrals are defined by

$$S(u) = \int u' \sin \left[\frac{\pi}{2} u^2 \right] dx,$$

$$C(u) = \int u' \cos \left[\frac{\pi}{2} u^2 \right] dx.$$

For $S(u)$ we have that

$$G_\alpha(\exp R_\alpha(Y)) = \frac{[e^{i\pi/2 Y^2}]^2 - 1}{2i e^{i\pi/2 Y^2}}$$

where $G_\alpha(W) = (W^2 - 1)/2iW$ and $R_\alpha(Y) = i\pi Y^2/2$. For $C(u)$ we have a similar expression.

The logarithmic integral is defined by

$$\text{li}(u) = \int \frac{u'}{\log u} dx$$

with $H_\beta(W) = 1/W$ and $S_\beta(Y) = Y$.

\mathcal{EL} -elementary functions do not include the dilogarithm (or Spence function) defined by

$$\text{Li}_2(u) = - \int \frac{u' \log u}{u - 1} dx$$

nor the exponential integral

$$\text{Ei}(u) = \int \frac{u' e^u}{u} dx$$

since they both violate condition (1) of the definition. Of course, $\text{Ei}(u) = \text{li}(e^u)$, so the exponential integral is implicitly covered by our analysis. One would like a theory that explicitly includes these functions but this remains an open problem.

We can now state the generalization of Liouville's Theorem.

THEOREM 1.1. *Let F be a differential field of characteristic zero with an algebraically closed subfield of constants C . Let γ be in F and assume there exist an \mathcal{EL} -elementary extension E of F and an element y in E such that $y' = \gamma$. Then there exist constants $a_i, b_{i\alpha}, c_{i\beta}$ in C , w_i in F , and $u_{i\alpha}, u_{i\beta}, v_{i\alpha}, v_{i\beta}$, algebraic over F , such that*

$$(1.2) \quad \gamma = w'_0 + \sum_{i=1}^n a_i \frac{w'_i}{w_i} + \sum_{\alpha \in A} \sum_{i \in I_\alpha} b_{i\alpha} u'_{i\alpha} G_\alpha(v_{i\alpha}) + \sum_{\beta \in B} \sum_{i \in J_\beta} c_{i\beta} u'_{i\beta} H_\beta(v_{i\beta})$$

where I_α and J_β are finite sets of integers for all α and β and

$$v'_{i\alpha} = (R_\alpha(u_{i\alpha}))' v_{i\alpha}, \quad v'_{i\beta} = \frac{(S_\beta(u_{i\beta}))'}{S_\beta(u_{i\beta})}, \quad S_\beta(u_{i\beta}) \neq 0$$

for all α, β and i .

The proof of Theorem 1.1 will be given in § 3. Now some comments about the hypotheses and conclusion of the theorem.

Condition (2) in the first paragraph of this section seems artificial, but the theorem is false without it. Consider the following example.

Example 1.1. Let $F = \mathbf{C}(x, \log x)$, where \mathbf{C} is the field of complex numbers, $\mathcal{E} = \emptyset$ and $\mathcal{L} = \{(\log Y(Y+1))^2\}$. In this case the index set B is a singleton and $H = Y^2$. This is excluded by condition (2) since $\text{deg}(\text{numerator of } H) = 2 > \text{deg}(\text{denominator of } H) + 1$.

Claim. (a) $\int \log x/(x+1)$ lies in an \mathcal{EL} -elementary extension of F but (b) $\log x/(x+1) \neq w'_0 + \sum c_i w'_i/w_i + \sum d_i u'_i v_i^2$ for any w_i, u_i, v_i algebraic over F with $v'_i = (u_i(u_i+1))'/u_i(u_i+1)$ and constants c_i, d_i in \mathbf{C} .

To verify (a), compute $\int (\log x(x+1))^2 dx$ by parts. First we have that

$$\int (\log x)^2 dx = x(\log x)^2 - 2x \log x + 2x,$$

$$\int (\log(x+1))^2 dx = (x+1)(\log(x+1))^2 - 2(x+1) \log(x+1) + 2(x+1),$$

and

$$\int (\log x)(\log (x+1)) dx = x(\log x) \log (x+1) - (x+1) \log (x+1) - \log x + 2x + \int \frac{\log x}{x+1} dx.$$

Hence

$$\begin{aligned} \int (\log x(x+1))^2 dx &= \int (\log x + \log (x+1))^2 dx \\ &= \int (\log x)^2 + 2 \int (\log x)(\log (x+1)) dx + \int (\log (x+1))^2 dx \\ &= \text{elementary function} + 2 \int \frac{\log x}{x+1} dx. \end{aligned}$$

To verify (b) assume that $\log x/(x+1) = w'_0 + \sum_{i=1}^n c_i w'_i/w_i + \sum_{i=1}^m d_i u'_i v_i^2$ with w_i, u_i, v_i algebraic over F and $v'_i = (u_i(u_i+1))'/u_i(u_i+1)$. From the structure theorem ([ROCA79, p. 359]), we have for each $i, 1 \leq i \leq m$, that $u_i(u_i+1) = c_i x^{r_i}$ for some rational number r_i and $c_i \in \mathbb{C}$. We can assume that neither c_i nor r_i is zero. We also have $v_i = r_i \log x + k_i$ for some $k_i \in \mathbb{C}$. Furthermore, each u_i is algebraic over $K = \mathbb{C}(x, \log x, x^{r_1}, \dots, x^{r_m})$ and satisfies the irreducible equation $u(u+1) - c_i x^{r_i} = 0$. Letting Tr be the trace function from $K(u_1, \dots, u_m)$ to K , we see from this equation that $\text{Tr}(u_i)$ is an integer. Therefore, $\text{Tr}(u'_i) = (\text{Tr } u_i)' = 0$. Apply the trace to both sides of

$$\frac{\log x}{x+1} = w'_0 + \sum c_i \frac{w'_i}{w_i} + \sum d_i u'_i (r_i \log x + k_i)^2$$

to obtain

$$\mu \frac{\log x}{x+1} = (\text{Tr } w_0)' + \sum c_i \frac{(Nw_i)'}{Nw_i}$$

where μ is a positive integer and Nw_i is the norm of w_i . This contradicts the fact that $\int \log x/(x+1)$ is not elementary and hence (b) is verified.

Unlike the standard Liouville Theorem, the above theorem only guarantees that there exist $w_i, u_{i\alpha}, u_{i\beta}, v_{i\alpha}, v_{i\beta}$, algebraic over F such that (1.2) holds. One would have hoped that these elements could be chosen to lie in F but this is not the case in general.

Example 1.2. Let $F = \mathbb{C}(x, \exp x, \exp(-\exp x + x/2))$, $\mathcal{E} = \{\exp(-Y^2)\}$, $\mathcal{L} = \emptyset$. Note that F is a purely transcendental extension of \mathbb{C} .

Claim. (a) $\int \exp(-\exp x + x/2) dx$ lies in an \mathcal{EL} -elementary extension of F . (b) $\exp(-\exp x + x/2) \neq w'_0 + \sum c_i w'_i/w_i + \sum d_i u'_i v_i$, where $v'_i = (-2u_i u'_i) v_i$, for any w_i, u_i, v_i in F .

To verify (a), we see that

$$\int \exp\left(-\exp x + \frac{x}{2}\right) dx = \int \exp(-\exp x) \exp\left(\frac{x}{2}\right) dx = \sqrt{\pi} \operatorname{erf}\left(\exp \frac{x}{2}\right).$$

Note that $\exp(x/2) \notin F$.

To verify (b), assume such an expression existed. By the structure theorem in [ROCA79], we have $u_i^2 = r_i(\exp x + x/2) + s_i x + a_i$ where r_i and s_i are rational numbers and $a_i \in \mathbb{C}$. Since F is a purely transcendental extension of \mathbb{C} , this is only possible if

$r_i = s_i = 0$ and $u_i \in \mathbb{C}$. Therefore we would have $\exp(-\exp x + x/2) = w'_0 + \sum c_i w'_i/w_i$, contradicting the fact that the error function is not elementary.

2. The question of constants. In this section we will discuss the question of transcendental constants appearing in our integral when we express this integral in terms of \mathcal{EL} -elementary functions. We will rely heavily on the notion of a constrained extension of a differential field and other concepts from differential algebra. We refer the reader to [KOL73] as a general reference for differential algebra and explicitly to page 142 for an exposition of the concept of constrained extension.

We quote two facts from [KOL73]: 1) Let F be a differential field of characteristic 0, P a differential ideal in the ring of differential polynomials $F\{y_1, \dots, y_n\}$ and B a differential polynomial in $F\{y_1, \dots, y_n\}$ such that $B \notin P$. There exist elements η_1, \dots, η_n in some extension of F such that (η_1, \dots, η_n) is a zero of P , $B(\eta_1, \dots, \eta_n) \neq 0$ and (η_1, \dots, η_n) is constrained over F ; 2) Let F be as before. If (η_1, \dots, η_n) is constrained over F , then the constants of $F\langle \eta_1, \dots, \eta_n \rangle$ are algebraic over the constants of F .

PROPOSITION 2.1. *Let F be a differential field of characteristic 0, I a differential ideal in $F\{y_1, \dots, y_m\}$ and $D \in F\{y_1, \dots, y_n\}$ such that $D \notin I$. If there exist an \mathcal{EL} -elementary extension E of F and elements ζ_1, \dots, ζ_m in E such that $(\zeta_1, \dots, \zeta_m)$ is a zero of I with $D(\zeta_1, \dots, \zeta_m) \neq 0$, then there exists an \mathcal{EL} -elementary extension \bar{E} of F , whose constants are algebraic over the constants of F , and $(\bar{\zeta}_1, \dots, \bar{\zeta}_m) \neq 0$.*

Proof. Let $E = F(\theta_1, \dots, \theta_n)$ where each θ_i satisfies (i), (ii), (iii), (iv) or (v) of (1.1). Each of these conditions defines θ_i in terms of differential equations involving elements of $F(\theta_1, \dots, \theta_{i-1})$. These elements can be written as quotients of elements in $F[\theta_1, \dots, \theta_{i-1}]$. Let C_i be the product of the denominators of all elements of $F(\theta_1, \dots, \theta_{i-1})$ appearing in the definition of θ_i . Similarly each ζ_i can be written as $\zeta_i = A_i(\theta_1, \dots, \theta_n)/B_i(\theta_1, \dots, \theta_n)$. Let $G(y_1, \dots, y_n) = D(\prod_{i=1}^m B_i)(\prod_{i=1}^n C_i)$. We can write $F\{\theta_1, \dots, \theta_n\}$ as $F\{y_1, \dots, y_n\}/P$ for some prime differential ideal P . Note that $G \notin P$ and $I \subset P$. Using 1) above, we can find η_1, \dots, η_n , constrained over F such that (η_1, \dots, η_n) is a zero of P and $G(\eta_1, \dots, \eta_n) \neq 0$. One can easily check that $F\langle \eta_1, \dots, \eta_n \rangle$ is an \mathcal{EL} -elementary extension of F which, by fact 2) above, has constants which are at worst algebraic over the constants of F . Furthermore, letting $\bar{\zeta}_i = A_i(\eta_1, \dots, \eta_n)/B_i(\eta_1, \dots, \eta_n)$, we have that $(\bar{\zeta}_1, \dots, \bar{\zeta}_m)$ is a zero of I and $D(\bar{\zeta}_1, \dots, \bar{\zeta}_m) \neq 0$. \square

COROLLARY 2.2. *Let F be a differential field of characteristic 0 and $\gamma \in F$. If $y' = \gamma$ has a solution in some \mathcal{EL} -elementary extension of F , then $y' = \gamma$ has a solution in some \mathcal{EL} -elementary extension of F whose constants are algebraic over the constants of F .*

Proof. Let ζ be a solution of $y' = \gamma$ lying in an \mathcal{EL} -elementary extension of F and let $F\{\zeta\} = F\{y\}/I$ for some prime differential ideal I . Let $D = 1$ and apply Proposition 2.1. \square

As mentioned in § 1, we took care to define \mathcal{EL} -elementary functions in terms of differential equations without explicitly mentioning the functions \exp and \log . This is to prevent the appearance of constants that are generated transcendentially, e.g., as values of \exp or \log . If we insist upon using the functions \exp and \log , i.e. those functions satisfying $y' = y$, $y(0) = 1$ and $y' = 1/x$, $y(1) = 0$ respectively, we are forced to deal with this kind of constant as the following example shows.

Example 2.1. Let \mathbb{Q} be the rational numbers and let $F = \mathbb{Q}(x, \exp(-x^2 + 1))$, $\mathcal{E} = \{\exp(-Y^2)\}$, $\mathcal{L} = \emptyset$, and $\gamma = \exp(-x^2 + 1)$.

Claim. (1) There exist u, v in F such that $\gamma = u'v$ where $v' = (-u^2)'v$ and so, a fortiori, there is an \mathcal{EL} -elementary extension E of F , with the same constants as F , and a y in E such that $t' = \gamma$.

(2) γ cannot be written as $\gamma = w'_0 + \sum c_i (w'_i/w_i) + \sum d_i u'_i \exp(-u_i^2)$ for any elements $w_i, u_i, \exp(-u_i^2)$ algebraic over F and constants c_i, d_i algebraic over \mathbf{Q} .

To prove claim (1), let $u = x, v = \exp(-x^2 + 1)$. Then $u' = 1$ and $v' = (-x^2 + 1)'v = (-x^2)'v = (-u^2)'v$. Let θ be defined by $\theta' = u'v$. One can show that θ is transcendental over F and that $F(\theta)$ has the same field of constants as F . $E = F(\theta)$ is then an \mathcal{EL} -elementary extension of F and $y = \theta$ satisfies $y' = \gamma$.

To prove claim (2), assume that we could write

$$(2.1) \quad \exp(-x^2 + 1) = w'_0 + \sum_{i=1}^n c_i \frac{w'_i}{w_i} + \sum_{i=1}^m d_i u'_i \exp(-u_i^2),$$

with $w_i, u_i, \exp(-u_i^2)$ algebraic over F and constants c_i, d_i algebraic over \mathbf{Q} , and m as small as possible. Since u_i^2 and $\exp(-u_i^2)$ are algebraic over $\mathbf{Q}(x, \exp(-x^2 + 1))$ we have, by [ROS76, Thm. 2], each u_i is algebraic over $\mathbf{Q}(x)$. We now apply an old result of Liouville (see [RITT48, p. 49] or [ROS75, p. 295] for a modern proof): If $f_1, \dots, f_k, g_1, \dots, g_k$ are algebraic functions, such that no two of the g_i differ by a constant, then $f_1 \exp(g_1) + \dots + f_k \exp(g_k)$ is the derivative of an elementary function if and only if each $f_i \exp(g_i)$ is. To apply this result rewrite (2.1) as

$$\exp(-x^2 + 1) + \sum d_i u'_i \exp(-u_i^2) = w'_0 + \sum c_i \frac{w'_i}{w_i}.$$

Since $\int \exp(-x^2 + 1)$ is not elementary, we have either: (i) $-u_i^2$ and $-u_j^2$ differ by a constant for some $i \neq j$, or (ii) $-x^2 + 1$ and $-u_i^2$ differ by a constant for some i . In case (i), we see that the constant (which is algebraic over \mathbf{Q}) must be 0, otherwise $\exp(-u_i^2) (\exp(-u_j^2))^{-1}$ would be a transcendental constant lying in an algebraic extension of F , a contradiction. We must therefore have $-u_i^2 = -u_j^2$ so $u_i = \pm u_j$. This implies that we could combine terms in (2.1) to yield an expression with smaller m . In case (ii), the constant again must be zero., Therefore $-x^2 + 1 = -u_i^2$ for some i . Letting $I = \{i | -u_i^2 = -x^2 + 1\}$ and $J = \{i | -u_i^2 \neq -x^2 + 1\}$ we have

$$\left(1 + \sum_{i \in I} d_i u'_i\right) \exp(-x^2 + 1) + \sum_{i \in J} d_i u'_i \exp(-u_i^2) = w'_0 + \sum c_i w'_i/w_i.$$

Applying the result of Liouville and the previous argument, we must get $J = \emptyset$ and so

$$\left(1 + \sum_{i \in I} d_i u'_i\right) \exp(-x^2 + 1) = w'_0 + \sum c_i \frac{w'_i}{w_i}.$$

Since $\int \exp(-x^2 + 1) dx$ is not elementary we must have $1 + \sum d_i u'_i = 0$. Since $-u_i^2 = x^2 + 1$, we have $\text{Tr}(u_i) = 0$, where Tr is the trace with respect to the extension $\mathbf{Q}(x, u_1, \dots, u_m)$ of $\mathbf{Q}(x)$. Therefore, $0 = \text{Tr}(1 + \sum d_i u'_i) = 1 + \sum d_i (\text{Tr}(u_i))' = 1$, a contradiction. \square

3. Proof of Theorem 1.1. We will need the following three easy lemmas.

LEMMA 3.1. *Let k be a field containing the algebraic closure of the rationals and let X and Y be indeterminants. Let $A(Y)$ and $B(Y)$ be relatively prime elements of $k[Y]$. Furthermore, assume A/B is not an n th power in $k(Y)$ for any positive integer n . Then the polynomial $B(Y)X^m - A(Y)$ is irreducible in $k(X)[Y]$ for any positive integer m .*

Proof. By Gauss's Lemma $B(Y)X^m - A(Y)$ factors in $k(X)[Y]$ if and only if it factors in $k[X, Y]$ if and only if $X^m - A(Y)/B(Y)$ factors in $k(Y)[X]$. Now apply [LANG65, Thm. 16, p. 221]. \square

LEMMA 3.2. *Let k be a field, X and Y indeterminants, and $A(Y)$ and $B(Y)$ relatively prime elements of $k[Y]$. If a and b are elements of k with $a \neq 0$, then $A(Y) - (aX + b)B(Y)$ is irreducible in $k(X)[Y]$.*

Proof. This again follows from two applications of Gauss’s Lemma and the fact that $aX + b - A(Y)/B(Y)$ is irreducible in $k(Y)[X]$. \square

LEMMA 3.3. *Let k be a differential field with algebraically closed field of constants C . For any $S(Y)$ in $C(Y)$, any u, v in k such that $v' = (S(u))'/S(u)$ and for any $a, b \in C$, there exist w_0, \dots, w_n in k, c_1, \dots, c_n in C such that $u'(av + b) = w'_0 + \sum c_i w'_i w_i$.*

Proof. It is enough to show that $u'(av + b)$ has an antiderivative in some elementary extension of k and then apply Liouville’s Theorem. If we write $S(Y) = \beta \prod (Y - \alpha_i)^{n_i}$ where the α_i are in C and n_i are integers, then we can write $v' = \sum n_i(u - \alpha_i)'/(u - \alpha_i)$. Thus $v = \sum n_i v_i$ for v_i in some elementary extension of K such that $v'_i = (u - \alpha_i)'/(u - \alpha_i)$. One can then check that $u'(av_i + b) = (a(u - \alpha_i)(v_i - 1) + bu)'$. \square

Proof of Theorem 1.1. First of all, we may assume that for all β in $B, S_\beta(Y)$ is not an m th power for any positive integer m . If some $S_\beta(Y) = (\bar{S}_\beta(Y))^m$ then in the definition of \mathcal{L} and in condition (v) of (1.1) we could replace $S_\beta(Y)$ by $\bar{S}_\beta(Y)$ and $H_\beta(Y)$ by $\bar{H}_\beta(Y) = H_\beta(mY)$, so that $\bar{H}_\beta(\log \bar{S}_\beta(Y)) = H_\beta(\log S_\beta(Y))$. In this way we get a new set $\bar{\mathcal{L}}$, prove our theorem for $\mathcal{E}\bar{\mathcal{L}}$ -elementary extensions and then switch back.

Furthermore, assuming the hypothesis of the theorem, Corollary 2.2 states that we can assume that $y' = \gamma$ has a solution in an $\mathcal{E}\mathcal{L}$ -elementary extension of F , with no new constants.

We first assume F is algebraically closed. In this case, we proceed by induction on the transcendence degree of E over F . When the transcendence degree is zero, the result is trivial. When it is positive we apply induction and the problem is reduced to showing:

Let E be an algebraic extension of $F(\theta)$ where θ is transcendental over F and satisfies conditions (ii), (iii), (iv) or (v) of (1.1). Let $\gamma \in F$ and assume that E has no new constants and that there exist $w_i, u_{i\alpha}, u_{i\beta}, v_{i\alpha}, v_{i\beta}$ in E and constants $a_i, b_{i\alpha}, c_{i\beta}$ such that

$$(3.1) \quad \gamma = w'_0 + \sum a_i \frac{w'_i}{w_i} + \sum \sum b_{i\alpha} u'_{i\alpha} G_\alpha(v_{i\alpha}) + \sum \sum c_{i\beta} u'_{i\beta} H_\beta(v_{i\beta}),$$

where

$$v'_{i\alpha} = (R_\alpha(u_{i\alpha}))' v_{i\alpha} \quad \text{and} \quad v'_{i\beta} = \frac{(S_\beta(u_{i\beta}))'}{S_\beta(u_{i\beta})}.$$

Then there exist $\bar{w}_i, \bar{u}_{i\alpha}, \bar{u}_{i\beta}, \bar{v}_{i\alpha}, \bar{v}_{i\beta}$ in F and constants $\bar{a}_i, \bar{b}_{i\alpha}, \bar{c}_{i\beta}$ in F such that

$$\gamma = \bar{w}'_0 + \sum \bar{a}_i \frac{\bar{w}'_i}{\bar{w}_i} + \sum \sum \bar{b}_{i\alpha} \bar{u}'_{i\alpha} G_\alpha(\bar{v}_{i\alpha}) + \sum \sum \bar{c}_{i\beta} \bar{u}'_{i\beta} H_\beta(\bar{v}_{i\beta}),$$

where

$$\bar{v}'_{i\alpha} = (R_\alpha(\bar{u}_{i\alpha}))' \bar{v}_{i\alpha} \quad \text{and} \quad \bar{v}'_{i\beta} = \frac{(S_\beta(\bar{u}_{i\beta}))'}{S_\beta(\bar{u}_{i\beta})}.$$

We shall deal with each of the cases (ii)–(v) separately. The main idea is to take the trace of both sides of (3.1) to force everything to belong to $F(\theta)$. We then will equate terms in the partial fraction decomposition with respect to θ and show that the term not depending on θ on the right-hand side can be put in the prescribed form.

Case (ii). $\theta' = u'\theta$ for some u in F .

For each α, β, i we have $v'_{i\alpha} = (R_\alpha(u_{i\alpha}))'v_{i\alpha}$, and $(S_\beta(u_{i\beta}))' = v'_{i\beta}S_\beta(u_{i\beta})$, then by [ROS76, Theorem 2] we have that

$$(3.2) \quad \begin{aligned} v_{i\alpha} &= f_{i\alpha}\theta^{r_{i\alpha}}, \\ S_\beta(u_{i\beta}) &= f_{i\beta}\theta^{r_{i\beta}}, \end{aligned}$$

for some rational numbers $r_{i\alpha}, r_{i\beta}$ and elements $f_{i\alpha}, f_{i\beta}$ of F . Furthermore we have

$$(3.3) \quad \begin{aligned} R_\alpha(u_{i\alpha}) &= r_{i\alpha}u + g_{i\alpha}, \\ v_{i\beta} &= r_{i\beta}u + g_{i\beta}, \end{aligned}$$

with $g_{i\alpha}$ and $g_{i\beta}$ in F . Note that we can arrange that $r_{i\alpha}$ and $r_{i\beta}$ are actually integers. To see this, let $r_{i\alpha} = s_{i\alpha}/n$ and $r_{i\beta} = s_{i\beta}/n$, where $s_{i\alpha}, s_{i\beta}$ and n are integers. Let $\bar{\theta} = \theta^{1/n}$. We then have $\bar{\theta}' = 1/n u'\bar{\theta}$ and $F \subset E \subset E(\bar{\theta})$. If we replace E by $E(\bar{\theta})$ and θ by $\bar{\theta}$, we still have fields of the appropriate form and furthermore, $v_{i\alpha} = f_{i\alpha}\bar{\theta}^{s_{i\alpha}}$, and $S_\beta(u_{i\beta}) = f_{i\beta}\bar{\theta}^{s_{i\beta}}$, where $s_{i\alpha}$ and $s_{i\beta}$ are integers. We shall use the old notation but from now on assume that $r_{i\alpha}$ and $r_{i\beta}$ are integers.

We want to take the trace of both sides of (3.1) over $F(\theta)$. Note that from (3.2) and (3.3), the $v_{i\alpha}$ and $S_\beta(u_{i\beta})$ are in $F(\theta)$ and the $R_\alpha(u_{i\alpha})$ and $v_{i\beta}$ are in F (which implies that $u_{i\alpha}$ is in F). The only elements which may give us trouble when we take the trace are the $u_{i\beta}$ which, a priori, are only algebraic over $F(\theta)$.

To calculate the trace of the $u_{i\beta}$, write

$$S_\beta(Y) = \frac{A_\beta(Y)}{B_\beta(Y)}$$

where A_β and B_β are relatively prime polynomials. Then $u_{i\beta}$ satisfies

$$A_\beta(Y) - f_{i\beta}\theta^{r_{i\beta}}B_\beta(Y) = 0$$

which, by Lemma 3.1, is irreducible over $F(\theta)$. Therefore the trace of $u_{i\beta}$ can be calculated from the coefficients of this polynomial. The coefficients are all of the form $\delta(f_{i\beta}\theta^{r_{i\beta}}) + \varepsilon$ where δ and ε are constants. Dividing by the leading coefficient, we get

$$\text{Tr } u_{i\beta} = m \left(\frac{\delta(f_{i\beta}\theta^{r_{i\beta}}) + \varepsilon}{\mu(f_{i\beta}\theta^{r_{i\beta}}) + \nu} \right)$$

where m is an integer and $\delta, \varepsilon, \mu, \nu$ are constants. We then have

$$(\text{Tr } u_{i\beta})' = m \frac{(\delta\nu - \varepsilon\mu)(f'_{i\beta} + f_{i\beta}r_{i\beta}u')\theta^{r_{i\beta}}}{(\mu(f_{i\beta}\theta^{r_{i\beta}}) + \nu)^2}.$$

Note that the coefficient of θ^0 in the partial fraction decomposition of this expression is 0, assuming that $r_{i\beta} \neq 0$.

We are now ready to take traces in equation (3.1). Doing this we get

$$(3.4) \quad M\gamma = (\text{Tr } w_0)' + \sum a_i \frac{(Nw_i)'}{Nw_i} + \sum \sum b_{i\alpha} u'_{i\alpha} G_\alpha(v_{i\alpha}) + \sum \sum c_{i\beta} (\text{Tr } u_{i\beta})' H_\beta(v_{i\beta})$$

where M is some integer and, abusing notation, the $a_i, b_{i\alpha}, c_{i\beta}$ are possibly different constants. Let us collect the coefficient of θ^0 on the right-hand side of this equation. If we write $\text{Tr } w_0 = \sum \sum (a_{ij}/(\theta - \mu_i)^j) + P(\theta)$, the standard calculations (as in [RISC69, p. 169]) show that the coefficient of θ^0 in $(\text{Tr } w_0)'$ is

$$(3.5) \quad \tilde{w}'_0$$

where \tilde{w}_0 is the coefficient of θ^0 in $P(\theta)$. Considering the next expression in (3.4), we write

$$\sum_i a_i \frac{(Nw_i)'}{Nw_i} = \sum_i \sum_j a_i \left(\frac{l'_i}{l_i} + n_{ij} \frac{(\theta + \mu_j)'}{(\theta - \mu_j)} \right)$$

where $Nw_i = l_i \Pi(\theta - \mu_j)^{n_{ij}}$ for some l_i, μ_j in F and integers n_{ij} . The coefficient of θ^0 here is

$$(3.6) \quad \sum a_i \frac{l'_i}{l_i} + \sum_i \sum_j a_i n_{ij} u'_j.$$

Next we consider the expression

$$\begin{aligned} \sum \sum b_{i\alpha} u'_{i\alpha} G_\alpha(v_{i\alpha}) &= \sum \sum b_{i\alpha} u'_{i\alpha} G_\alpha(f_{i\alpha} \theta^{r_{i\alpha}}) \\ &= \sum \sum_{r_{i\alpha} \neq 0} b_{i\alpha} u'_{i\alpha} G_\alpha(f_{i\alpha} \theta^{r_{i\alpha}}) + \sum \sum_{r_{i\alpha} = 0} b_{i\alpha} u'_{i\alpha} G_\alpha(f_{i\alpha}). \end{aligned}$$

The coefficient of θ^0 in the expression corresponding to the sum over those i, α with $r_{i\alpha} \neq 0$ is $\sum \sum b_{i\alpha} d_{\alpha 0} u'_{i\alpha}$ where $d_{\alpha 0}$ is the coefficient of Y^0 in $G_\alpha(Y)$, which is a constant. The expression corresponding to the sum over i, α with $r_{i\alpha} = 0$ has no occurrence of θ , so the coefficient of θ^0 in $\sum \sum b_{i\alpha} u'_{i\alpha} G_\alpha(v_{i\alpha})$ is of the form

$$(3.7) \quad v' + \sum \sum b_{i\alpha} u'_{i\alpha} G_\alpha(v_{i\alpha})$$

where $v, u_{i\alpha}, v_{i\alpha}$ are in F and $v'_{i\alpha} = (R_\alpha(u_{i\alpha}))' v_{i\alpha}$. Finally, we consider the expression

$$(3.8) \quad \begin{aligned} \sum \sum c_{i\beta} (\text{Tr } u_{i\beta})' H_\beta(v_{i\beta}) \\ = \sum \sum_{r_{i\beta} \neq 0} c_{i\beta} (\text{Tr } u_{i\beta})' H_\beta(v_{i\beta}) + \sum \sum_{r_{i\beta} = 0} c_{i\beta} (\text{Tr } u_{i\beta})' H_\beta(v_{i\beta}) \end{aligned}$$

where $r_{i\beta}$ is defined in (3.2). Note that by (3.3) $H_\beta(v_{i\beta})$ is in F and that if $r_{i\beta} = 0$ then $u_{i\beta}$ is in F so that $\text{Tr } u_{i\beta}$ is in F . Therefore the sum corresponding to $r_{i\beta} = 0$ has no occurrence of θ . If $r_{i\beta} \neq 0$, we showed that the coefficient of θ^0 in $(\text{Tr } u_{i\beta})'$ is zero, so the coefficient of θ^0 in the term corresponding to $r_{i\beta} \neq 0$ is 0. Therefore the coefficient of θ^0 in $\sum \sum c_{i\beta} (\text{Tr } u_{i\beta})' H_\beta(v_{i\beta})$ is

$$\sum \sum_{r_{i\beta} = 0} c_{i\beta} (\text{Tr } u_{i\beta})' H_\beta(v_{i\beta})$$

where $v'_{i\beta} = (S_\beta(u_{i\beta}))' / S_\beta(u_{i\beta})$ and $u_{i\beta}, v_{i\beta} \in F$. Combining (3.5), (3.6), (3.7) and (3.8), we see that the coefficient of θ^0 on the right-hand side of (3.4) is of the prescribed form and, since, for $i \neq 0$, θ^i does not occur on the left hand side, we have that $M\gamma$ equals this prescribed form.

Case (iii). $\theta' = u'/u$ for some $u \in F$. Again [ROS76, Thm. 2] implies that

$$(3.9) \quad R_\alpha(u_{i\alpha}) = d_{i\alpha} \theta + g_{i\alpha}, \quad v_{i\beta} = d_{i\beta} \theta + g_{i\beta},$$

for some constants $d_{i\alpha}, d_{i\beta}$ and elements $g_{i\alpha}, g_{i\beta}$ in F and that the $v_{i\alpha}$ and the $S_\beta(u_{i\beta})$ are in F . So in particular, we have that the $v_{i\alpha}, v_{i\beta}$, and the $u_{i\beta}$ are in $F(\theta)$. We only know that the $u_{i\alpha}$ are algebraic over $F(\theta)$ and so must calculate their trace.

Let

$$R_\alpha(Y) = \frac{A_\alpha(Y)}{B_\alpha(Y)}$$

where A and B are relatively prime polynomials with constant coefficients. Each $u_{i\alpha}$ satisfies $A_\alpha(u_{i\alpha}) - (d_{i\alpha} \theta + g_{i\alpha}) B_\alpha(u_{i\alpha}) = 0$. By Lemma 3.2, the polynomial $A_\alpha(Y) - (d_{i\alpha} \theta + g_{i\alpha}) B_\alpha(Y)$ is irreducible over $F(\theta)$ so the trace can be read off from its

coefficients. As before, we see that

$$\text{Tr } u_{i\alpha} = m \left(\frac{\delta(d_{i\alpha}\theta + g_{i\alpha}) + \varepsilon}{\mu(d_{i\alpha}\theta + g_{i\alpha}) + \nu} \right)$$

where $\delta, \varepsilon, \mu, \nu$ are constants. Therefore

$$(\text{Tr } u_{i\alpha})' = \frac{\text{element of } F}{(\mu(d_{i\alpha}\theta + g_{i\alpha}) + \nu)^2}$$

Note that if $\mu d_{i\alpha} \neq 0$, then the coefficient of θ^0 in this expression is 0. If $\mu = 0$ and $d_{i\alpha} \neq 0$, then

$$(\text{Tr } u_{i\alpha})' = \frac{m}{\nu} \left(\delta \left(d_{i\alpha} \frac{u'}{u} + g'_{i\alpha} \right) \right) = \frac{\delta}{\nu} m R_\alpha(u_{i\alpha})'$$

Now let us take the trace of both sides of (3.1):

$$M\gamma = (\text{Tr } w_0)' + \sum a_i \frac{(Nw_i)'}{Nw_i} + \sum \sum b_{i\alpha} (\text{Tr } u_{i\alpha})' G_\alpha(v_{i\alpha}) + \sum \sum c_{i\beta} u'_{i\beta} H_\beta(v_{i\beta})$$

and let us consider each of the terms on the right separately.

Recalling from (3.9), that each $v_{i\beta} = d_{i\beta}\theta + g_{i\beta}$ we can write the last sum as

$$(3.10) \quad \sum \sum c_{i\beta} u'_{i\beta} H_\beta(v_{i\beta}) = \sum_{d_{i\beta}=0} \sum c_{i\beta} u'_{i\beta} H_\beta(v_{i\beta}) + \sum_{d_{i\beta} \neq 0} \sum c_{i\beta} u'_{i\beta} H_\beta(v_{i\beta})$$

The sum corresponding to $d_{i\beta} = 0$ has $u_{i\beta}$ and $v_{i\beta}$ in F and is of the desired form. To deal with the sum corresponding to $d_{i\beta} \neq 0$, recall that we have assumed that $\text{deg}(\text{numerator } H_\beta) \leq \text{deg}(\text{denominator } H_\beta) + 1$ so the partial fraction decomposition of H_β is

$$\sum \sum \frac{a_{ij}}{(Y - \alpha_i)^j} + P_\beta(Y)$$

where P_β is a polynomial of degree ≤ 1 . We can therefore write

$$\sum_{d_{i\beta} \neq 0} \sum c_{i\beta} u'_{i\beta} H_\beta(v_{i\beta}) = \sum \sum c_{i\beta} u'_{i\beta} (H_\beta(v_{i\beta}) - P_\beta(v_{i\beta})) + \sum \sum c_{i\beta} u'_{i\beta} P_\beta(v_{i\beta}).$$

The first term is a proper rational function of θ (i.e. the degree of the numerator is less than the degree of the denominator). By Lemma 3.3, the second term is of the form $v' + \sum d_i v'_i / v_i$. Therefore we can write (3.10) as

$$(3.11) \quad \begin{aligned} & \sum \sum c_{i\beta} u'_{i\beta} H_\beta(v_{i\beta}) \\ & = \text{an expression whose } \theta^0 \text{ term is } \sum_{d_{i\beta}=0} \sum c_{i\beta} u'_{i\beta} H_\beta(v_{i\beta}), \\ & \quad \text{with no terms containing } \theta^i \text{ for } i > 0 \\ & \quad + \text{an expression of the form } v'_0 + \sum d_i \frac{v'_i}{v_i} \end{aligned}$$

where $u_{i\beta}$ and $v_{i\beta}$ are in F , v_i are in $F(\theta)$ and the d_i and $c_{i\beta}$ are constants. We shall deal with the θ^0 term of $v' + \sum d_i v'_i / v_i$ later.

We now look at the next term which we write as

$$\sum \sum b_{i\alpha} (\text{Tr } u_{i\alpha})' G_\alpha(v_{i\alpha}) = m_d \sum_{d_{i\alpha}=0} \sum b_{i\alpha} u'_{i\alpha} G_\alpha(v_{i\alpha}) + \sum_{d_{i\alpha} \neq 0} \sum b_{i\alpha} (\text{Tr } u_{i\alpha})' G_\alpha(v_{i\alpha}).$$

Note that if $d_{i\alpha} = 0$, then $u_{i\alpha}$ and $v_{i\alpha}$ are in F so $\text{Tr } u_{i\alpha}$ is an integer multiple of $u_{i\alpha}$.

This integer is designated by m . If $d_{i\alpha} \neq 0$ we have shown that the θ^0 term of $(\text{Tr } u_{i\alpha})'$ is zero or a constant times $R(u_{i\alpha})'$. Therefore the θ^0 term of the sum corresponding to $d_{i\alpha} \neq 0$ is of the form

$$\sum_{d_{i\alpha} \neq 0} \sum e_{i\alpha} R(u_{i\alpha})' G_\alpha(v_{i\alpha}) = \sum_{d_{i\alpha} \neq 0} \sum e_{i\alpha} \frac{v'_{i\alpha}}{v_{i\alpha}} G_\alpha(v_{i\alpha}) = v'_0 + \sum d_i \frac{v'_i}{v_i}$$

where $e_{i\alpha}$ and d_i are constants and the v_i are in $F(\theta)$. This last equality follows from the fact that $G_\alpha(v_{i\alpha})/v_{i\alpha}$ is a rational function of $v_{i\alpha}$ with constant coefficients. Therefore, we have

$$\begin{aligned} & \sum \sum b_{i\alpha} (\text{Tr } u_{i\alpha})' G_\alpha(v_{i\alpha}) \\ & = \text{an expression whose } \theta^0 \text{ term is } \sum_{d_{i\alpha}=0} \sum b_{i\alpha} u'_{i\alpha} G_\alpha(v_{i\alpha}), \\ (3.12) \quad & \text{with no terms containing } \theta^i \text{ for } i > 0 \\ & + \text{an expression of the form } v'_0 + \sum d_i \frac{v'_i}{v_i} \end{aligned}$$

where $u_{i\alpha}$ and $v_{i\alpha}$ are in F , v_i are in $F(\theta)$ and the $b_{i\alpha}$ and d_i are constants. From (3.11) and (3.12) we can conclude that

$$\begin{aligned} \gamma = v'_0 + \sum d_i \frac{v'_i}{v_i} + \text{an expression whose } \theta^0 \text{ term is a constant multiple of} \\ (3.13) \quad \sum_{d_{i\alpha}=0} \sum b_{i\alpha} u'_{i\alpha} G_\alpha(v_{i\alpha}) + \sum_{d_{i\beta}=0} \sum c_{i\beta} u'_{i\beta} H_\beta(v_{i\beta}) \text{ and with no terms} \\ \text{containing } \theta^i \text{ for } i > 0 \end{aligned}$$

where $u_{i\alpha}, u_{i\beta}, v_{i\alpha}, v_{i\beta}$ are in F , v_i are in $F(\theta)$ and $b_{i\alpha}, c_{i\alpha}, d_i$ are constants. We now want to calculate the θ^0 term of $v'_0 + \sum d_i v'_i/v_i$. If we write $v_i = \xi_i \prod_j (\theta - \mu_j)^{n_{ij}}$, $i \neq 0$, where ξ_i, μ_j are in F , then the θ^0 term of v'_i/v_i is ξ'_i/ξ_i . Letting $v_0 = \sum_{j=0}^l k_j \theta^j + \text{terms of degree } < 0 \text{ in } \theta$, we have that the θ^0 term of v'_0 is $k'_0 + k_1 u'/u$. If $l > 1$ or k_1 is not a constant, we would have that the right-hand side of (3.13) would contain an expression of the form θ^i with $i \geq 1$. Therefore we have that $l = 1$, k_1 is a constant and the θ^0 term of $v'_0 + \sum d_i v'_i/v_i$ is of the form $u_0 + \sum a_i u'_i/u_i$ with the $u_i \in F$ and a_i constants. This and (3.13) shows that γ has the correct form.

Case (iv) and (v). $\theta' = u'G_\alpha(v)$ or $\theta' = u'H_\beta(v)$ where $v' = (R_\alpha(u))'v$ or $v' = (S_\beta(v))'/S_\beta(v)$ with $u, v \in F$.

In this case we can assume that θ is not elementary over F , otherwise the problem could be reduced to the above considerations. Since $\theta' \in F$, we have that the $S(u_{i\beta})$ and $v_{i\alpha}$ are in F and that $R(u_{i\alpha}) = d_{i\alpha}\theta + g_{i\alpha}$ and $v_{i\beta} = d_{i\beta}\theta + g_{i\beta}$ with the $d_{i\alpha}, d_{i\beta}$ constants and $g_{i\alpha}, g_{i\beta}$ in F . We must have $d_{i\alpha} = d_{i\beta} = 0$, otherwise θ would be elementary over F . Therefore, we can write (3.1) as

$$(3.14) \quad \gamma - \sum \sum b_{i\alpha} u'_{i\alpha} G_\alpha(v_{i\alpha}) - \sum \sum c_{i\beta} u'_{i\beta} H_\beta(v_{i\beta}) = u'_0 + \sum a_i \frac{u'_i}{u_i}$$

with all terms on the left in F . Liouville's Theorem now applies and tells us that the expression on the right must equal $\tilde{u}'_0 + \sum \tilde{a}_i \tilde{u}'_i/\tilde{u}_i$ for some $\tilde{u}_i \in F$ and constants \tilde{a}_i . This completes the proof of Theorem 1.1 in the case that F is algebraically closed.

Now we remove the assumption that F is algebraically closed. The above argument shows that (1.2) holds with $a_i, b_{i\alpha}, c_{i\beta}$ in C and $w_i, u_{i\alpha}, u_{i\beta}, v_{i\alpha}, v_{i\beta}$ algebraic over F . Let K be a finite normal extension of F containing $w_i, u_{i\alpha}, u_{i\beta}, v_{i\alpha}, v_{i\beta}$ and let σ be an

element of the Galois group of K over F . Then

$$\sigma(\gamma) = \gamma = (\sigma w_0)' + \sum a_i \frac{(\sigma w_i)'}{\sigma w_i} + \sum \sum b_{i\alpha} \sigma(u_{i\alpha})' G_\alpha(\sigma v_{i\alpha}) + \sum \sum c_{i\beta} \sigma(u_{i\beta})' H_\beta(\sigma v_{i\beta})$$

where $(\sigma v_{i\alpha})' = (R_\alpha(\sigma u_{i\alpha}))' \sigma v_{i\alpha}$ and $(\sigma v_{i\beta})' = (S_\beta(\sigma u_{i\beta}))' / S_\beta(\sigma u_{i\beta})$, $S_\beta(\sigma u_{i\beta}) \neq 0$.

Summing over all σ in the Galois group of K over F yields, for some M in \mathbf{Z} ,

$$M\gamma = (\text{Tr } w_0)' + \sum a_i \frac{(Nw_i)'}{Nw_i} + \sum_\sigma \sum \sum b_{i\alpha} \sigma(u_{i\alpha})' G_\alpha(\sigma v_{i\alpha}) + \sum_\sigma \sum \sum c_{i\beta} \sigma(u_{i\beta})' H_\beta(\sigma v_{i\beta}).$$

Since $\text{Tr } w_0$ and the norms Nw_i are in F , this yields the final conclusion of the theorem. \square

II. The error function.

4. Statement and discussion of results. In this section we shall specialize the results of the previous sections to the case when $\mathcal{E} = \{\exp(-Y^2)\}$ and $\mathcal{L} = \emptyset$, that is, to integration in terms of error functions and elementary functions. To be more explicit, we say that a differential field E is an *erf-elementary extension* of F if there exists a tower of fields $F = F_0 \subset \dots \subset F_n = E$ such that $F_i = F_{i-1}(\theta_i)$ where for each i , $1 \leq i \leq n$, one of the following holds:

- (i) θ_i is algebraic over F_{i-1} ;
- (ii) $\theta_i' = u_i' \theta_i$ for some u_i in F_{i-1} ;
- (iii) $\theta_i' = u_i' / u_i$ for some $u_i \neq 0$ in F_{i-1} ;
- (iv) $\theta_i' = u_i' v_i$ for some u_i, v_i in F_{i-1} with $v_i' = (-u_i^2)' v_i = -2u_i u_i' v_i$.

Recall that a differential field F is a Liouvillian extension of a differential field k if there exists a tower $k = k_0 \subset \dots \subset k_m = F$ such that $k_i = k_{i-1}(\xi_i)$ where for each i , $1 \leq i \leq m$, we have either:

- (i) ξ_i is algebraic over k_{i-1} , or
- (ii) $\xi_i' \in k_{i-1}$, or
- (iii) $\xi_i' / \xi_i \in k_{i-1}$.

We then have the following result.

THEOREM 4.1. *Let F be a Liouvillian extension of its field of constants C . Assume C is of characteristic zero and algebraically closed and let γ be an element of F . If γ has an antiderivative in some erf-elementary extension of F , then there exist constants a_i and b_i in C , elements w_i in F , and elements u_i and v_i algebraic over F such that*

$$(4.1) \quad \gamma = w_0' + \sum a_i \frac{w_i'}{w_i} + \sum b_i u_i' v_i$$

where $v_i' = (-u_i^2)' v_i$ and u_i^2, v_i^2 and $u_i' v_i$ are in F .

Proof. By Theorem 1.1, we know that there exist constants a_i and b_i in C and elements w_i in F , u_i and v_i algebraic over F satisfying (4.1). We want to show that these can be chosen such that u_i^2, v_i^2 and $u_i' v_i$ are in F . The lemma of [ROS177, p. 338] implies that each u_i^2 is in F and some power of v_i is in F . Let E be a normal extension of F containing all the w_i, u_i and v_i and let σ be an automorphism of E over F . We then have $\sigma u_i = \pm u_i$ and $\sigma v_i = \zeta_{\sigma i} v_i$ where $\zeta_{\sigma i}$ is a root of unity. Therefore,

$$\gamma = \sigma\gamma = (\sigma w_0)' + \sum_i a_i \frac{(\sigma w_i)'}{\sigma w_i} + \sum_i b_i (\pm \zeta_{\sigma i}) u_i' v_i.$$

If we sum over all automorphisms of E over F we get:

$$m\gamma = mw'_0 + m \sum_i a_i \frac{w'_i}{w_i} + \sum_i b_i \left(\sum_{\sigma} \pm \zeta_{\sigma i} \right) u'_i v_i$$

for some integer m . In the expression $\sum_i b_i (\sum_{\sigma} \pm \zeta_{\sigma i}) u'_i v_i$ we shall implicitly assume that we are only summing over those i for which $\sum_{\sigma} \pm \zeta_{\sigma i} \neq 0$. For such an i , we have

$$u'_i v_i = \left(\sum_{\sigma} \pm \zeta_{\sigma i} \right)^{-1} \sum_{\sigma} \sigma(u'_i v_i)$$

so $u'_i v_i$ is left fixed by all automorphisms of E over F and so must lie in F . Furthermore, $(u'_i)^2 = \frac{1}{4}((u'_i)')^2 / u_i^2$, so $(u'_i)^2 \in F$. Since $v_i^2 = (u'_i v_i)^2 / (u'_i)^2$ we have $v_i^2 \in F$. \square

The example at the end of § 1 shows that the u_i and v_i cannot be guaranteed to lie in F . Despite this fact we are able to obtain a decision procedure (presented in § 7) when γ is built up using only exponential functions and rational operations.

Let F and k be differential fields of characteristic zero. We say that F is a *purely exponential extension of k* if $F = k(\theta_1, \dots, \theta_n)$ where θ_i is transcendental over $k(\theta_1, \dots, \theta_{i-1})$ and $\theta'_i = u'_i \theta_i$ for some u_i in $k(\theta_1, \dots, \theta_{i-1})$. The main result of § 7 is the following. Here, we use the term computable field to mean a field in which one can effectively carry out the field theoretic operations and over which one can effectively factor polynomials. Any finitely generated extension of \mathbf{Q} is computable as is the algebraic closure of \mathbf{Q} .

THEOREM 4.2. *Let C be a computable field, $C(x)$ a differential field with derivation ' defined by $x' = 1$ and $c' = 0$ for all c in C , and let F be a purely exponential extension of $C(x)$. Given γ in F , one can decide in a finite number of steps if γ has an antiderivative in an erf-elementary extension of F and if so find $a_i, b_i, u_i, v_i,$ and w_i satisfying (4.1).*

The rest of this paper is devoted to proving this result.

5. Purely exponential extensions. In practice, when we are asked to integrate a function γ , we are not given a differential field F containing γ . In this section we shall show how to make a good choice for a field of definition containing γ . This field will be chosen so that the exponentials appearing in this field satisfy as few relations as possible and so that the u_i and v_i which could possibly appear in (4.1) are already in F . To do this we need some facts about purely exponential extensions.

Let F be a purely exponential extension of k . When we refer to such a field, we shall always consider it as being given by a *fixed* set of generators $\theta_1, \dots, \theta_n$ over k , so $F = k(\theta_1, \dots, \theta_n)$. Renumbering the θ_i , we may write $k = F_0 \subset \dots \subset F_r = F$ where $F_i = F_{i-1}(\theta_{i1}, \dots, \theta_{im_i})$ for $i = 1, \dots, r$ and where the θ_{ij} 's are algebraically independent over k and satisfy $\theta'_{ij} = u'_{ij} \theta_{ij}$ for some u_{ij} in F_{i-1} with u_{ij} not in F_{i-2} . Note that, one can always *uniquely* arrange the θ_i 's in groups to satisfy the above. We define the *rank of $F = k(\theta_1, \dots, \theta_n)$ over k* to be the r -tuple (m_r, \dots, m_1) and we designate this by $\text{rank}_k F$.

Let us consider two examples.

Example 5.1. Let $k = \mathbf{C}(x)$, $F = k(\exp x^2, \exp(\exp x^2), \exp(\exp(x^2) + x))$. $k = \mathbf{C}(x) = F_0 \subset F_1 = F_0(\exp(x^2)) \subset F_2 = F_1(\exp(\exp x^2), \exp(\exp(x^2) + x))$. We have $\text{rank}_k F = (2, 1)$.

Example 5.2. Let k be as above and $\bar{F} = k(\exp x^2, \exp x, \exp(\exp x^2))$. We can write $k = \bar{F}_0 \subset \bar{F}_1 = \bar{F}_0(\exp x^2, \exp x) \subset \bar{F}_2 = \bar{F}_1(\exp(\exp x^2)) = \bar{F}$. We have $\text{rank}_k \bar{F} = (1, 2)$.

Note that $F \cong \bar{F}$; only the generating sets are different. This underlines the important fact that the rank depends on the particular $\theta_1, \dots, \theta_n$ we chose to generate

F over k . Note that if $\text{rank}_k F = (m_r, \dots, m_1)$ then $m_1 + \dots + m_r$ is the transcendence degree of F over k .

We can also define the rank of an exponential in F . Let F be a purely exponential extension of k and let F_0, \dots, F_r be as above. Let u, v be elements of F such that $v' = u'v$. We define the rank of v ($\text{rank}_k v$) to be the smallest i such that $v \in F_i$. Note that if $\text{rank}_k v = i$, then $u \in F_{i-1}$. Also note that in Example 5.1, $\text{rank}_k \exp(x) = 2$ while in Example 5.2 $\text{rank}_k \exp(x) = 1$.

Given two sequences (m_r, \dots, m_1) and $(\bar{m}_s, \dots, \bar{m}_1)$ we say $(m_r, \dots, m_1) < (\bar{m}_s, \dots, \bar{m}_1)$ if $r < s$ or if $r = s$ and (m_r, \dots, m_1) is less than $(\bar{m}_s, \dots, \bar{m}_1)$ in the usual lexicographical ordering. We say that a purely exponential extension F of k is of minimal rank over k if for any algebraic extension \bar{F} of F , where \bar{F} is also a purely exponential extension of k , we have $\text{rank}_k F \leq \text{rank}_k \bar{F}$. For example $C(x, \exp(x^2), \exp(\exp x^2), \exp(\exp x^2 + x))$ is not of minimal rank over $C(x)$, since it is contained in (in fact equal to) $C(x, \exp(x^2), \exp(x), \exp(\exp x^2))$ which is of smaller rank. We will show later that $C(x, \exp(x^2), \exp(x), \exp(\exp x^2))$ is of minimal rank over $C(x)$.

We will need the following technical lemma in Proposition 5.2.

LEMMA 5.1. Let $E = F(\theta_1, \dots, \theta_m)$ where $\theta'_i = u'_i \theta_i$ with u_i in $F(\theta_1, \dots, \theta_{i-1})$. Assume that E and F have the same field of constants and that $\theta_1, \dots, \theta_m$ are algebraically independent over F . If ζ is an element of E such that ζ' is in F , then ζ is in F .

Proof. Proceeding by induction on m , we can assume that $m = 1$. In this case write $E = F(\theta)$ where $\theta'/\theta \in F$. Since θ and ζ are algebraically dependent over F , we have, by [ROS76, Thm. 2], that ζ is algebraic over F . Since $F(\theta)$ is a transcendental extension of F , we must have $\zeta \in F$. \square

PROPOSITION 5.2. Let F be a purely exponential extension of $k = C(x)$, where $x' = 1$ and $c' = 0$ for all c in C , and let $k = F_0 \subset \dots \subset F_r = F$ where $F_i = F_{i-1}(\theta_{i1}, \dots, \theta_{im_i})$ with $\theta'_{ij} = u'_{ij} \theta_{ij}$ for some $u_{ij} \in F_{i-1}$, $u_{ij} \notin F_{i-2}$. Then F is of minimal rank over k if and only if, for each $i = 2, \dots, r$ the following holds:

$$(5.1) \quad \sum_{j=1}^{m_i} n_j u_{ij} \in F_{i-2} \text{ for some integers } n_j \text{ implies } n_j = 0 \text{ for all } n_j.$$

Proof. Assume that F is of minimal rank over k and that for some i , there exist n_1, \dots, n_{m_i} , not all zero, such that $\sum_{j=1}^{m_i} n_j u_{ij} \in F_{i-2}$. Without loss of generality, we can assume $n_1 \neq 0$. Let

$$\theta = \prod_{j=1}^{m_i} \theta_{ij}^{n_j/n_1} \quad \text{and} \quad v = \sum_{j=1}^{m_i} \frac{n_j}{n_1} u_{ij}.$$

We then have $\theta' = v'\theta$. Let

$$\begin{aligned} \bar{F}_0 &= F_0, \\ \bar{F}_1 &= F_1, \\ &\vdots \\ \bar{F}_{i-2} &= F_{i-2}, \\ \bar{F}_{i-1} &= F_{i-1}(\theta), \\ \bar{F}_i &= \bar{F}_{i-1}(\theta_{i2}^{1/n_1}, \dots, \theta_{im_i}^{1/n_1}), \\ \bar{F}_{i+1} &= \bar{F}_i \cdot F_{i+1}, \\ &\vdots \\ \bar{F}_r &= \bar{F}_{r-1} \cdot F_r \end{aligned}$$

where $\bar{F}_k \cdot F_{k+1}$ is the compositum of these two fields. Note that \bar{F}_r is an algebraic

extension of F_r . The rank of \bar{F}_r is $(m_r, \dots, m_{i+1}, m_i - 1, m_{i-1} + 1, \dots, m_1)$ which is smaller than $\text{rank}_k F_r = (m_r, \dots, m_i, m_{i-1}, \dots, m_1)$. This contradicts the fact that $F_r = F$ is of minimal rank over k and so (5.1) must hold.

Now assume that (5.1) holds. We wish to show that F is of minimal rank over k . Let \bar{F} be a purely exponential extension, algebraic over F , such that $\text{rank}_k \bar{F} = (\bar{m}_s, \dots, \bar{m}_1) \leq (m_r, \dots, m_1) = \text{rank}_k F$. Let $k = \bar{F}_0 \subset \bar{F}_1 \subset \dots \subset \bar{F}_s = \bar{F}$ where $\bar{F}_i = \bar{F}_{i-1}(\bar{\theta}_{i1}, \dots, \bar{\theta}_{im_i})$ where $\bar{\theta}'_{ij} = \bar{u}_{ij}\bar{\theta}_{ij}$ for some $\bar{u}_{ij} \in \bar{F}_{i-1}$ and $\bar{u}_{ij} \notin \bar{F}_{i-2}$. We will show that for each i , \bar{F}_i is algebraic over F_i and therefore that $\bar{m}_i \leq m_i$ for each i and $s \leq r$. Since $\text{tr. deg}_k F = \text{tr. deg}_k \bar{F}$, we have $\sum_{i=1}^r m_i = \sum_{i=1}^s \bar{m}_i$ and so $m_i = \bar{m}_i$ for each i . Therefore $\text{rank}_k F = \text{rank}_k \bar{F}$.

To prove that \bar{F}_i is algebraic over F_i , we proceed by induction on i . If $i=0$, $F_i = k = \bar{F}_i$, so we are done. Now assume that \bar{F}_j is algebraic over F_j for $j < i$. Since \bar{F} is algebraic over F , we have that $\bar{\theta}_{i1}, \dots, \bar{\theta}_{im_i}$ are algebraic over F . By the lemma of [ROSI77, p. 338], we have that $\bar{\theta}_{ij}^N \in F$ for some nonzero integer N . Furthermore,

$$\frac{(\bar{\theta}_{ij}^N)'}{\bar{\theta}_{ij}^N} = N\bar{u}'_{ij} \in \bar{F}_{i-1} \cap F = F_{i-1}$$

since by induction \bar{F}_{i-1} is algebraic over F_{i-1} and F_{i-1} is relatively algebraically closed in F . If we write $F = C(x)(\theta_{11}, \dots, \theta_{1m_1}, \dots, \theta_{r1}, \dots, \theta_{rm})$ where $\theta'_{ij} = u'_{ij}\theta_{ij}$ then by [ROCA79, Thm. 3.1] we have that

$$\bar{u}_{ij} = \sum_{\substack{1 \leq b \leq m_a \\ 1 \leq a \leq r}} r_{ab}^{ij} u_{ab} + c$$

for some rational numbers r_{ab}^{ij} and constant c . Since $\bar{u}'_{ij} \in F_{i-1}$, we have by Lemma 5.1 that $\bar{u}_{ij} = \sum r_{ab}^{ij} u_{ab} + c \in F_{i-1}$. If some $r_{ab}^{ij} \neq 0$ with $a > i$, we would contradict (5.1). Therefore $r_{ab}^{ij} = 0$ if $a > i$ and

$$\bar{\theta}_{ij} = d \prod_{a=1}^i \prod_{b=1}^{m_a} \theta_{ab}^{r_{ab}^{ij}}.$$

This implies that $\bar{\theta}_{ij}$ is algebraic over F_i and so \bar{F}_i is algebraic over F_i . \square

Using Proposition 5.2, we now can show that $C(x, \exp x^2, \exp x, \exp(\exp x^2))$ is of minimal rank over $C(x)$. Here $F_0 = C(x) \subset F_1 = F_0(\exp x, \exp x^2) \subset F_2 = F_1(\exp(\exp x^2))$. We must check if $n_0 \exp x^2 \in F_0$ has a nonzero solution n_0 in the integers (which it clearly does not). Similarly we can reaffirm that $C(x, \exp(x^2), \exp(\exp x^2), \exp(\exp x^2 + x))$ is not of minimal rank. Here $F_0 = C(x) \subset F_1 = F_0(\exp x^2) \subset F_2 = F_1(\exp(\exp x^2), \exp(x + \exp x^2))$. Here $\theta_{11} = \exp x^2$, $u_{11} = x^2$, $\theta_{21} = \exp(\exp x^2)$, $u_{21} = \exp x^2$, $\theta_{22} = \exp(x + \exp x^2)$, $u_{22} = x + \exp x^2$. Note that $u_{21} - u_{22} = -x \in F_0$.

PROPOSITION 5.3. *Let C be a computable field and F a purely exponential extension of $C(x)$. We can construct an algebraic extension \bar{F} of F such that \bar{F} is a purely exponential extension of $C(x)$ and such that \bar{F} is of minimal rank over $C(x)$.*

Proof. We will use the criterion (5.1) of proposition 5.2. Let $C(x) = F_0 \subset \dots \subset F_r = F$ with $F_i = F_{i-1}(\theta_{i1}, \dots, \theta_{im_i})$ as before. For each i , we check if there exist n_{ij} , not all zero such that $\sum_{j=1}^{m_i} n_{ij} u_{ij} \in F_{i-2}$. If, for some i , such a set of n_{ij} exists, say with $n_{i1} \neq 0$, let

$$\theta = \prod_{j=1}^{m_i} \theta_{ij}^{n_{ij}/n_{i1}}$$

Let

$$\begin{aligned} \bar{F}_j &= F_j \text{ for } j < i - 1, \\ \bar{F}_{i-1} &= F_{i-1}(\theta), \\ \bar{F}_i &= \bar{F}_{i-1}(\theta_{i_2}^{1/n_{i_1}}, \dots, \theta_{i_{m_i}}^{1/n_{i_1}}), \\ \bar{F}_j &= F_j \cdot \bar{F}_i \text{ for } j > i. \end{aligned}$$

We then have that \bar{F}_r is a purely exponential extension of k , algebraic over F and of smaller rank than F over k . We claim that if we continue this process, it will end after at most N^2 steps where $N = \text{tr. deg}_k F$. This is because each time we repeat this process we decrease one of the integers in (m_r, \dots, m_1) by one and increase its neighbor to the right by one. This can be done at most $rm_r + (r-1)m_{r-1} + \dots + m_1 \leq rN \leq N^2$. \square

PROPOSITION 5.4. *Let C be an algebraically closed field and F a purely exponential extension of $C(x)$, where $x' = 1$ and $c' = 0$ for all c in C . One can construct a purely exponential extension F^* of $C(x)$, containing F , which has the following property:*

(5.2) *If u and v satisfy $v' = u'v$ and u and v are algebraic over F with v^2 in F , then u and v are in F^* .*

Proof. Let $F = C(x, \theta_1, \dots, \theta_n)$ where $\theta'_i = u'_i \theta_i$ with u_i in $C(x, \theta_1, \dots, \theta_{i-1})$ and let $F^* = C(x, \theta_1^{1/2}, \dots, \theta_n^{1/2})$. One can easily show that F^* is a purely exponential extension of $C(x)$ containing F . Since u and v are algebraic over F , we have by [ROCA79, Thm. 3.1], that $v = d \prod_{i=1}^n \theta_i^{r_i}$ where d is in C and the r_i are in \mathbf{Q} . Since $v^2 = d^2 \prod_{i=1}^n \theta_i^{2r_i}$ is in F and F is a purely transcendental extension of C , we have that $2r_i$ is an integer, for each i . Therefore v is in F^* and so u is in F^* . \square

6. Squares in purely transcendental extensions. In our decision procedure, we will be called upon to decide when certain elements of a field are squares. We discuss this algebraic question in this section. Let K be a field and $K(x_1, \dots, x_n)$ a purely transcendental extension of K . Let P be an element of $K(x_1, \dots, x_n)$ with P not in K and let \tilde{K} be the algebraic closure of K . We wish to show that the set of α in \tilde{K} such that $P + \alpha = Q^2$ for some Q in $\tilde{K}(x_1, \dots, x_n)$ is finite (or empty) and computable if K is a computable field. We first prove the following ancillary lemma.

LEMMA 6.1. *Let f and g be elements of $K[x_1, \dots, x_n]$ with no common factors and assume that either f or g is not in K . Then the set of α in \tilde{K} such that $f^2 + \alpha g^2 = h^2$ for some h in $\tilde{K}[x_1, \dots, x_n]$ is a finite set and can be constructed if K is a computable field.*

Proof. Let f and g be of degree $\leq k$, let N be the dimension of the vector space of all such polynomials and let \mathbf{P}^N be the projective space of dimension N over \tilde{K} . Let S be the subset of $\tilde{K} \times \mathbf{P}^N$ consisting of all $(\alpha, (c_1, \dots, c_N, d))$ such that $d^2(f^2 + \alpha g^2) = h^2$ where h is a polynomial with coefficients c_1, \dots, c_N . S is a Zariski-closed subset of $\tilde{K} \times \mathbf{P}^N$ and if we let $p: \tilde{K} \times \mathbf{P}^N \rightarrow \tilde{K}$ be the projection map, then $p(S)$ is the set mentioned in the lemma. By classical elimination theory ([MUM76, p. 33] or [VDW50, p. 6]), we know that $p(S)$ is a Zariski closed subset of \tilde{K} and so is either finite or all of \tilde{K} . Furthermore, we know that if K is constructible, we can find the defining equations of $p(S)$. We need only check that $p(S) \neq \tilde{K}$.

Assume that $p(S) = \tilde{K}$ and that $\partial f / \partial x_1 \neq 0$ (since either f or g is not in K we may assume one of them, say f , depends on some x_i , say x_1). Let u be any element in K^n such that $g(u) \neq 0$ and let α_u be a nonzero element in K such that $f(u)^2 + \alpha_u g(u)^2 = 0$. Since we are assuming that $p(S) = \tilde{K}$, there is some polynomial h_u such that $f^2 + \alpha_u g^2 =$

h_u^2 . Note that $h_u(u) = 0$. Differentiating h_u^2 , we get

$$\frac{\partial h_u^2}{\partial x_1^2} = 2h_u \frac{\partial h_u}{\partial x_1} = 2f \frac{\partial f}{\partial x_1} + 2\alpha_u g \frac{\partial g}{\partial x_1}.$$

We therefore have the following identities

$$\begin{aligned} f(u) \cdot f(u) + \alpha_u g(u) \cdot g(u) &= 0, \\ f(u) \cdot \frac{\partial f}{\partial x_1}(u) + \alpha_u g(u) \cdot \frac{\partial g}{\partial x_1}(u) &= 0. \end{aligned}$$

From this we can conclude that

$$f(u) \frac{\partial g}{\partial x_1}(u) - g(u) \frac{\partial f}{\partial x_1}(u) = 0.$$

Since this holds for all u in the open set where $g(u) \neq 0$, we have that

$$f \frac{\partial g}{\partial x_1} - g \frac{\partial f}{\partial x_1} = 0.$$

Since f and g have no common factors, we have that f divides $\partial f / \partial x_1$, a contradiction. Therefore $p(S) \neq \tilde{K}$ and so must be finite. \square

PROPOSITION 6.2. *Let $P \in K(x_1, \dots, x_n)$ with $P \notin K$. Then there exist only a finite number of values α in \tilde{K} such that $P + \alpha = Q^2$ for some $Q \in K(x_1, \dots, x_n)$. Furthermore, if K is computable, we can find these α .*

Proof. As before, we can show that the set of such α is a Zariski closed subset of \tilde{K} , whose defining equations can be calculated if K is computable. To show that this set is finite, it is enough to show that it is not all of \tilde{K} . Assuming that it is, we would have 0 being an element of this set and so P would be a square. Write $P = f^2/g^2$ where f and g have no common factors. For each α , we could find relatively prime f_α, g_α such that

$$\frac{f^2}{g^2} + \alpha = \frac{f_\alpha^2}{g_\alpha^2} \quad \text{or} \quad f^2 + \alpha g^2 = \frac{f_\alpha^2 g^2}{g_\alpha^2}.$$

From this we see that g_α is a constant multiple of g so $f^2 + \alpha g^2 = c f_\alpha^2$ for some constant c . Now apply the preceding lemma to get a contradiction. \square

7. The decision procedure. In this section we shall prove Theorem 4.2. Let C be a computable field, F a purely exponential extension of $C(x)$ and $\gamma \in F$. Extend F to F^* as in Proposition 5.4 and use Proposition 5.3 to extend to a field E which is of minimal rank over $C(x)$. We may assume that C is algebraically closed since the algebraic closure of a computable field is still computable. Using Theorem 4.1 and Proposition 5.4, we see that we want to decide if there exist c_i and d_i in C and w_i, u_i, v_i in E such that

$$(7.1) \quad \gamma = w'_0 + \sum c_i \frac{w'_i}{w_i} + \sum_{i \in I} d_i u'_i v_i \quad \text{where } v'_i = (-u_i^2)' v_i.$$

We can assume that if $\sum_{i \in J} d_i u'_i v_i$ has an elementary antiderivative for some subset $J \subset I$ and constants d_i , then $d_i = 0$ for all i in J . This just means that all of the elementary part of the antiderivative of γ is contained in $w'_0 + \sum c_i (w'_i/w_i)$. The idea behind the procedure is to first determine the possible expressions of the form $u'_i v_i$, with $u_i, v_i \in E$ and $v'_i = (-u_i^2)' v_i$, which could appear in (7.1). This is done as follows.

Let $C(x) = E_0 \subset \dots \subset E_r = E$ where $E_i = E_{i-1}(\theta_{i1}, \dots, \theta_{im_i})$ and $u'_{ij} = u'_{ij}\theta_{ij}$ with $u_{ij} \in E_{i-1}$ and $u_{ij} \notin E_{i-2}$. Let v and u be elements of E such that $v' = (-u^2)'v$ and let v have rank s (i.e. $v \in E_s$ but $v \notin E_{s-1}$). The Structure Theorem of [ROCA79] permits us to write

$$(7.2) \quad v = d \prod_{\substack{1 \leq j \leq m_i \\ 1 \leq i \leq s}} \theta_{ij}^{-n_{ij}}$$

with $n_{ij} \in \mathbf{Z}$, $d \in C$. For notational convenience, our n_{ij} are the negatives of those in [ROCA79]. Note that i ranges from 1 to s but no further since E is of minimal rank and that $n_{ij} \in \mathbf{Z}$ and not just in \mathbf{Q} since E is a purely transcendental extension of C . We also have

$$(7.3) \quad u = \left(\sum_{\substack{1 \leq j \leq m_i \\ 1 \leq i \leq s}} n_{ij}u_{ij} + c \right)^{1/2}$$

where $c \in C$. We need one more piece of notation. Given any θ_{ij} , we can write γ in its partial fraction decomposition with respect to θ_{ij} over the field $C(x)$ ($\theta_{11}, \dots, \hat{\theta}_{ij}, \dots, \theta_{rm_r}$). (Where $\hat{}$ over an element means this element is omitted). Let

$$\gamma = A_{-m}\theta_{ij}^{-m} + \dots + A_0 + \dots + A_l\theta_{ij}^l + \sum_{a,b} \frac{P_{ab}(\theta_{ij})}{Q_b(\theta_{ij})^{n_{ab}}}$$

where Q_b is an irreducible polynomial in θ_{ij} , not equal to θ_{ij} . We define $o_{ij}(\gamma) = \max(m, l)$. We claim that given v of rank s appearing in (7.1), we have $|n_{sj}| \leq o_{sj}(\gamma)$ for $1 \leq j \leq m_s$. We are saying that if v is of rank s , those θ_{ij} 's which are also of rank s appear to a power of absolute value less than $o_{sj}(v)$ in (7.2). This claim will be proven below, so let us assume it for a moment. We still must bound the other exponents appearing in (7.2). It would be natural to conjecture that $|n_{ij}| \leq o_{ij}(\gamma)$, but this is not true, as the following example illustrates.

Example 7.1. Let $E = C(x, \exp x, \exp(-\exp(2x) + x))$. This is of minimal rank over $C(x)$. Letting $\gamma = \exp(-\exp(2x) + x)$, $\theta_1 = \exp x$ and $\theta_2 = \exp(-\exp(2x) + x)$ we have $\gamma = u'v$ where $u = \exp x = \theta_1$, $v = \exp(-\exp(2x)) = \theta_2\theta_1^{-1}$ and $v' = (-u^2)'v$. Note that both v and θ_2 are of rank 2 and that the exponent of θ_2 in v is bounded by (in fact equal to) $o_2(\gamma)$. Here θ_1 does not appear in γ , yet it does appear in v , u.i.e. $n_1 = -1$ while $o_1(\gamma) = 0$.

We will bound the n_{ij} for $i < s$ using the results of § 6. Rewrite (7.3) as

$$u = \left(n_{s1}u_{s1} + \dots + n_{sm_s}u_{sm_s} + \sum_{\substack{1 \leq j \leq m_i \\ 1 \leq i < s}} n_{ij}u_{ij} + c \right)^{1/2}$$

Let

$$P = n_{s1}u_{s1} + \dots + n_{sm_s}u_{sm_s} \quad \text{and} \quad \alpha = \sum_{\substack{1 \leq j \leq m_i \\ 1 \leq i < s}} n_{ij}u_{ij} + c.$$

Note that since E is of minimal rank over $C(x)$, we have that $P \in E_{s-1}$, $P \notin E_{s-2}$, and $\alpha \in E_{s-2}$. This is precisely where the notion of minimal rank is crucial. If we let $K = E_{s-2}$, we can apply Proposition 6.2 and find, for each choice of $(n_{s1}, \dots, n_{sm_s})$ satisfying $|n_{sj}| \leq o_{sj}(\gamma)$, all $\alpha \in E_{s-2}$ such that $P + \alpha = Q^2$ for some Q in E_{s-1} . Each such α can be written in at most one way as $\sum n_{ij}u_{ij} + c$ since if $\sum n_{ij}u_{ij} + c = \sum \bar{n}_{ij}u_{ij} + \bar{c}$ we would have $\sum (n_{ij} - \bar{n}_{ij})u_{ij} \in C$, so the θ_{ij} would be algebraically dependent over C unless $n_{ij} = \bar{n}_{ij}$ and $c = \bar{c}$. The α 's which can be written as such a sum will give us the exponents for the θ_{ij} 's of lower rank. Therefore, using our claim and Proposition 6.2, we can

determine for each $s, 1 \leq s \leq r$, all u and v in E which may appear in (7.1) such that $v' = (-u^2)'v$ and v is of rank s . We now wish to decide if there exist a_i, b_i in C and w_i in E such that

$$\gamma - \left(\sum a_i u_i' v_i \right) = w_0' + \sum b_i \frac{w_i'}{w_i}.$$

A procedure to decide this is presented in [MACK76]. Since this paper has never been published we have included in the Appendix a proof of the relevant theorem.

All that remains to be done is to prove that for v of rank s appearing in (7.1), $|n_{s_j}| \leq o_{s_j}(\gamma)$. The proof of this claim follows closely Risch's proof of [RISC69, Main Theorem] and will yield a proof of Theorem 4.2 independent of [MACK76].

We will proceed by induction on the number of generators of E over $C(x)$. If we write $E = E_r = E_{r-1}(\theta_{r1}, \dots, \theta_{rm_r})$ as before, let $\theta = \exp(u)$ denote one of the θ_{ri} and write $E = K(\theta)$, where $K = E_{r-1}(\theta_{r1}, \dots, \hat{\theta}_{rib}, \dots, \theta_{rm_r})$. Expanding γ in partial fractions with respect to θ , and assuming that γ satisfies (7.1) we have

$$\begin{aligned} \gamma &= A_k \theta^k + \dots + A_i \theta + A_0 + A_{-1} \theta^{-1} + \dots + A_{-m} \theta^{-m} \\ &\quad + \frac{A_{1k_1}}{p_1^{k_1}} + \dots + \frac{A_{11}}{p_1} + \dots + \frac{A_{sk_s}}{p_s^{k_s}} + \dots + \frac{A_{s1}}{p_s} \end{aligned}$$

$$(7.4) \quad \left. \begin{aligned} &B_i \theta^i + \dots + B_1 \theta + B_0 + B_{-1} \theta^{-1} + \dots + B_{-l} \theta^{-l} \\ &+ \int \sum_{i \in \mathcal{S}} d_i u_i' v_i + \int \sum_{i \in \mathcal{T}} d_i u_i' v_i + \sum c_j \log D_j \\ &+ \frac{B_{1k_1}}{p_1^{k_1-1}} + \dots + \frac{B_{11}}{p_1} + \int \frac{B_{10}}{p_1} \\ &+ \dots + \\ &+ \frac{B_{sk_s}}{p_s^{k_s-1}} + \dots + \frac{B_{s1}}{p_s} + \int \frac{B_{s0}}{p_s} \end{aligned} \right\}$$

where the A 's, B 's and D 's are in K , the p_i 's are irreducible polynomials in $K[\theta]$, $B_{i0}/p_i = \sum c_{ij} (q_{ij}/q_{ij})$ where $p_i = \prod q_{ij}$ in $\tilde{K}[\theta]$. $v_i' = (-u_i^2)'v_i$ and \mathcal{S} is the set of i such that $v_i \notin k$ while \mathcal{T} is the set of i such that $v_i \in K$. Note that [ROS76, Thm. 2] implies that in either case u_i in K . Some justification is required for our implicit assumption that p_i^{-1} appears to a power of at most $k_i - 1$ in the second expression. This follows from the fact that for $i \in \mathcal{S}$, $v_i = f_i \theta^{n_i}$ for some $f_i \in K$ (again by [ROS 76, Theorem 2]) and so when differentiating the second expression we get no cancellation in this expression. Note that for $i \in \mathcal{S}$ we can write $v_i = c_i \theta^{n_i} \prod \theta_{ij}^{n_{ij}}$ where $n_i \neq 0$. We shall first prove our claim for v_i with $i \in \mathcal{S}$, that is, that $|n_i| \leq \max(k, m)$. Assume not and let $n = \max_{i \in \mathcal{S}} (n_i)$. We then have

$$(B_n \theta^n)' + \sum_{\mathcal{S}'} d_i u_i' v_i = 0$$

where \mathcal{S}' is the set of i such that $n_i = n$. This implies that $\sum_{\mathcal{S}'} d_i u_i' v_i$ has an elementary antiderivative, contrary to our assumptions. This proves our claim for the v_i with $i \in \mathcal{S}$. Furthermore, we see by comparing powers of θ in our two expressions that $t = k$ and $l = m$.

We now proceed to determine those v_i with $i \in \mathcal{S}$ which may appear in (7.4). We do this using the results of § 6 as above. Let

$$\sum_{i \in \mathcal{S}} d_i u'_i v_i = \sum_{\substack{j \neq 0 \\ -m \leq j \leq k}} C_j \theta^j$$

where C_j is of the form $\sum d_i C_{ij}$ with C_{ij} known elements of K and d_i constants to be determined. Equating powers of θ in (7.4) we get

$$\begin{aligned} A_k &= B'_k + ku'B_k + \sum d_i C_{ik}, \\ &\vdots \\ A_1 &= B'_1 + u'B_1 + \sum d_i C_{i1}, \\ A_{-1} &= B'_{-1} - u'B_{-1} + \sum d_i C_{i,-1}, \\ &\vdots \\ A_{-m} &= B'_{-m} - mu'B_{-m} + \sum d_i C_{i,-m}. \end{aligned}$$

For each j we must determine if there exist constants d_i and elements B_j such that

$$B'_j + ju'B_j = A_j - \sum d_i C_{ij}.$$

This can be done using [RISC69, Main Theorem, part (b)]. Note that a solution is uniquely determined if it exists. In this way determine the B_j 's and d_i 's (for $i \in \mathcal{S}$). Proceeding as in [RISC69, p. 183], we can determine the $B_{i,k-1}, \dots, B_{i,1}$ until we get down to an equation of the form

$$\sum_{i=1}^s \frac{A_{i1}}{p_i} + A_0 = \sum_{i=1}^s \frac{B_{i0}}{p_i} + B'_0 + \sum_{i \in \mathcal{T}} d_i u'_i v_i + (\sum c_j \log D_j)'.$$

Again we proceed as in [RISC69] and reduce the problem to deciding if

$$A_0 - u' \sum m_i c_{ij} = \left(B_0 + \sum c_j \log D_j + \int \sum_{i \in \mathcal{T}} d_i u'_i v_i \right)'.$$

This is equivalent to deciding if

$$A_0 = \left(\bar{B}_0 + \sum c_j \log D_j + \int \sum_{i \in \mathcal{T}} d_i u'_i v_i \right)'$$

for some \bar{B}_0, D_j, u_i, v_i in K . Note that $o_{ij}(A_0) \leq o_{ij}(\gamma)$ so by the induction hypothesis we have

$$o_{ij}(v_i) \leq o_{ij}(A_0) \leq o_{ij}(\gamma)$$

for all v_i appearing in (7.1). \square

Appendix. In this section, we present a proof of the result of Carola Mack alluded to in § 7. We must first recall some definitions from [ROCA79]. Let kCK be differential fields. For $t \in K$ with $t' \in k$, we say that t is *simple logarithmic* over k if there exist u_1, \dots, u_m in k ($m \geq 1$) such that for some constant c , $t + c \in k(\log u_1, \dots, \log u_m)$. We say it is *nonsimple* over k if it is not simple logarithmic over k . K is a *regular log-explicit extension* of k if K and k have the same subfield of constants and there exists a tower $k = K_0 C \cdots CK_n = K$ such that $K_i = K_{i-1}(\theta_i)$ where θ_i is transcendental over K_{i-1} and either

- (i) $\theta'_i \in K_{i-1}$ and θ_i is nonsimple over K_{i-1} , or
- (ii) $\theta'_i = u'_i/u_i$ for some $u_i \in K_{i-1}$, or
- (iii) $\theta'_i = u'_i \theta_i$ for some $u_i \in K_{i-1}$.

We shall use the following fact several times in the proof of Theorem A1 below. Given a system L_1 of linear equations over a field K in $n+m$ variables $(x_1, \dots, x_m, y_1, \dots, y_m)$, there exists a system L_2 of linear equations over K in n variables (x_1, \dots, x_n) such that $(a_1, \dots, a_n) \in K^n$ satisfies L_2 if and only if there exists $(b_1, \dots, b_m) \in K^m$ such that $(a_1, \dots, a_n, b_1, \dots, b_m)$ satisfies L_1 . This follows from the fact that the projection to K^n of an affine subspace in K^{n+m} is still an affine subspace. We will refer to L_2 as the projection of L_1 onto the first n variables.

THEOREM A1. *Let K be a finitely generated extension of Q and let $F = K(z, \theta_1, \dots, \theta_n)$ be a regular log-explicit extension of $K(z)$, where $z' = 1$ and $c' = 0$ for all c in K .*

(a) *Let f_0, f_1, \dots, f_N be elements of F . Then one can determine in a finite number of steps a system L of linear equations in N variables with coefficients in K so that $f_0 + d_1 f_1 + \dots + d_N f_N$ has an integral in an elementary extension of K for d_1, \dots, d_N in \bar{K} (the algebraic closure of K) if and only if (d_1, \dots, d_N) satisfies L . For each (d_1, \dots, d_N) in \bar{K}^N satisfying L , we can find $v_0 \in F, v_i \in \bar{K}F$ for $i = 1, \dots, m$ and c_1, \dots, c_m in \bar{K} such that*

$$f_0 + d_1 f_1 + \dots + d_N f_N = v_0 + \sum_{i=1}^m c_i^{v_i/v_i}$$

(b) *Let $f, g_i, i = 1, \dots, m$ be elements of F . Then one can find, in a finite number of steps h_1, \dots, h_r in F and a set L of linear equations in $m+r$ variables with coefficients in K , such that $y' + fy = \sum_{i=1}^m c_i g_i$ holds for $y \in F$ and c_i in K if and only if $y = \sum_{i=1}^r y_i h_i$ where y_i are elements of K and $c_1, \dots, c_m, y_1, \dots, y_r$ satisfy L .*

Proof. We shall mimic the proof of [RISC69, Main Theorem] (and assume that the reader is familiar with that paper) and so proceed by induction on n . If $n = 0$, then $F = K(z)$ so we may take $L = \{0, 0\}$, since any element in $K(z)$ has an elementary integral. The proof of part (b) is the same as in [RISC69]. To proceed with the induction step, let $D = K(z, \theta_1, \dots, \theta_{n-1})$ and $F = D(\theta)$ where $\theta = \theta_n$.

(a) *Case 1.* $\theta' \in D$. Let $f = f_0 + d_1 f_1 + \dots + d_N f_N$. We can write

$$f = \begin{pmatrix} A_k \theta^k + \dots + A_0 \\ + \frac{A_1 k_1}{p_1^{k_1}} + \dots + \frac{A_1 1}{p_1} \\ \vdots \\ + \frac{A_s k_s}{p_s^{k_s}} + \dots + \frac{A_s 1}{p_s} \end{pmatrix} = \begin{pmatrix} B_{k+1} \theta^{k+1} + \dots + B_0 + \sum e_j \log D_j \\ + \frac{B_1 k_1 - 1}{p_1^{k_1 - 1}} + \dots + \frac{B_1 1}{p_1} + \int \frac{B_1 0}{p_1} \\ \vdots \\ + \frac{B_s k_s - 1}{p_s^{k_s - 1}} + \dots + \frac{B_s 1}{p_s} + \int \frac{B_s 0}{p_s} \end{pmatrix}'$$

Here the A 's are linear polynomials in d_1, \dots, d_N with coefficients in D and the B 's are to be determined. Equating powers of θ , we have

$$0 = B'_{k+1}$$

so $B_{k+1} \in K$, and

$$A_k = (k+1)B_{k+1}\theta' + B'_k$$

We can write $A_k = a_{k,0} + d_1 a_{k,1} + \dots + d_N a_{k,N}$ with $a_{k,i}$ in D for $i = 0, \dots, N$, so this last equation can be written as

$$(A.1) \quad B'_k = a_{k,0} + d_1 a_{k,1} + \dots + d_N a_{k,N} - (k+1)B_{k+1}\theta'$$

Using the induction hypothesis for (b), we conclude that there exist $h_{1,k}, \dots, h_{r,k}$ in D and L_k , a system of linear equations in $N+r_k+1$ variables with coefficients in K

such that (A.1) has a solution B_k in D if and only if $B_k = \sum_{i=1}^{r_k} y_{ik} h_{ik}$ where $y_{ik} \in K$ and $(d_1, \dots, d_N, B_{k+1}, y_{1k}, \dots, y_{r_k k})$ satisfies L_k . Notice that for each choice of d_1, \dots, d_N in K there is at most one choice of B_{k+1} in K for which there exist $y_{1k}, \dots, y_{r_k k}$ in K such that $(d_1, \dots, d_N, B_{k+1}, y_{1k}, \dots, y_{r_k k})$ satisfies L_k . Now let $B_k = \sum_{i=1}^{r_k} y_{ik} h_{ik}$ where the h_{ik} are indeterminants. We then have

$$A_{k-1} = kB_k\theta' + B'_{k-1} = k\left(\sum_{i=1}^{r_k} y_{ik} h_{ik}\right)\theta' + B'_{k-1}.$$

We can write $A_{k-1} = a_{k-1,0} + d_1 a_{k-1,1} + \dots + d_N a_{k-1,N}$, so

$$(A.2) \quad B'_{k-1} = a_{k-1,0} + d_1 a_{k-1,1} + \dots + d_N a_{k-1,N} + \sum_{i=1}^{r_k} y_{ik} (kh_{ik}\theta').$$

Using the induction hypothesis for (b) allows us to conclude that there exist $h_{1,k-1}, \dots, h_{r_{k-1},k-1}$ in D and L_{k-1} , a system of linear equations in $N + r_k + r_{k-1}$ variables with coefficients in K such that (A.2) has a solution B_{k-1} in D if and only if $B_{k-1} = \sum_{i=1}^{r_{k-1}} y_{i,k-1} h_{i,k-1}$ where $y_{i,k-1} \in K$ and $(d_1, \dots, d_N, y_{1,k-1}, \dots, y_{r_{k-1},k-1}, y_{1,k-1}, \dots, y_{r_{k-1},k-1}, k-1)$ satisfies L_{k-1} . Again, for each choice of d_1, \dots, d_N , there is at most one choice of $(y_{1k}, \dots, y_{r_k k})$ for which there exists $(y_{1,k-1}, \dots, y_{r_{k-1},k-1})$ satisfying L_{k-1} . We continue in this way, getting linear systems L_{k-2}, \dots, L_2 whose solutions guarantee the existence of B_{k-2}, \dots, B_2 . Finally, we have $A_0 = B_1\theta' + B'_0 + \sum e_j (\log D_j)'$. If we set $A_0 = a_{00} + d_1 a_{1,0} + \dots + d_N a_{N,0}$ and $B_1 = \sum_{i=1}^{r_1} y_{i1} h_{i1}$ we get

$$(A.3) \quad a_{00} + d_1 a_{1,0} + \dots + d_N a_{N,0} - \sum_{i=1}^{r_1} h_{i1} (y_{i1}\theta') = B'_0 + \sum e_j (\log D_j)'.$$

Using the induction hypothesis for part (a), we see that there exists a linear system L^* in $N + r_1$ variables such that an equation of this form holds for some $(d_1, \dots, d_N, h_{11}, \dots, h_{r_1 1})$ satisfying L^* .

Now consider

$$(A.4) \quad \sum_{j=1}^{k_1} \frac{A_{ij}}{p_i^j} = \left[\sum_{j=1}^{k_1-1} \frac{B_{ij}}{p_i^j} + \int \frac{B_{i0}}{p_i} \right]'.$$

We can find unique R, S , linear in d_1, \dots, d_N in $D[\theta, d_1, \dots, d_N]$, with $\deg_\theta R < \deg_\theta p'_1$ and $\deg_\theta S < \deg_\theta p_1$ such that

$$Rp_1 + Sp'_1 = A_{1,k_1}.$$

Let $-(k-1)B_{1,k-1} = S$, substitute into (A.4), and obtain a new relation

$$\sum_{j=1}^{k_1-1} \frac{A_{ij}^{(*)}}{p_1^j} = \left[\sum_{j=1}^{k_1-2} \frac{B_{1j}}{p_1^j} + \int \frac{B_{10}}{p_1} \right]'.$$

Continuing in this manner, we determine $B_{1,k-2}, \dots, B_{1,1}$, all linear in d_1, \dots, d_N . We are left with an equation of the form

$$\frac{A_{11}^{(**\dots)}}{p_1} = \left[\int \frac{B_{10}}{p_1} \right]'.$$

Here $B_{10}/p_1 = \sum_{j=1}^{s_1} c_{1j} q_{1j}/q'_1$ where $p_1 = \prod_{j=1}^{t_1} q_{1j}$ is a factorization of p_1 into monic irreducible factors over $\bar{K}D$. We must determine if c_{ij} exist in \bar{K} such that the equation holds. Let

$$\frac{A_{11}^{(**\dots)}}{p_1} = \sum_{j=1}^{t_1} \frac{Q_{ij}}{q_{ij}}$$

where Q_{ij} is linear in d_1, \dots, d_N and let L_1^{**} be the system of linear equations in $d_1, \dots, d_N, c_1, \dots, c_{s_1}$ gotten by equating terms in the partial fraction decomposition of

$$\sum \frac{Q_{ij}}{q_{ij}} = \sum c_{ij} \frac{q'_{ij}}{q_{ij}}.$$

Similarly we get $L_2^{**}, \dots, L_s^{**}$ for p_2, \dots, p_s . We now get L by projecting $L_k \cup \dots \cup L_2 \cup L^* \cup L_1^{**} \cup \dots \cup L_s^{**}$ onto the first N variables (see the remark preceding the statement of the Theorem 5).

Case 2. $\theta = \exp \zeta$. Let

$$f = \left\{ \begin{array}{l} A_k \theta^k + \dots + A_0 \theta \\ + A_{-m} \theta^{-m} + \dots + A_{-1} \theta^{-1} + A_0 \\ + \frac{A_{1k_1} + \dots}{p_1^{k_1}} + \dots + \frac{A_{11}}{p_1} \\ \vdots \\ + \frac{A_{sk_s} + \dots}{p_s^{k_s}} + \dots + \frac{A_{s1}}{p_s} \end{array} \right\} = \left\{ \begin{array}{l} B_k \theta^k + \dots + B_1 \theta \\ + B_{-m} \theta^{-m} + \dots + B_{-1} \theta^{-1} + B_0 + \sum e_j \log D_j \\ + \frac{B_{1k_1-1} + \dots}{p_1^{k_1-1}} + \dots + \frac{B_{11}}{p_1} + \int \frac{B_{10}}{p_1} \\ \vdots \\ + \frac{B_{sk_s-1} + \dots}{p_s^{k_s-1}} + \dots + \frac{B_{s1}}{p_s} + \int \frac{B_{s0}}{p_s} \end{array} \right\}$$

where the A 's are linear polynomials in d_1, \dots, d_N . We have

$$A_i = B'_i + i \zeta' B_k$$

for all $i, -m \leq i \leq k, i \neq 0$. Setting $A_i = a_{i0} + a_{i1}d_1 + \dots + a_{iN}d_N$ we get for each $i, -m \leq i \leq k, i \neq 0$,

$$(A.5) \quad B'_i + i \zeta' B_i = a_{i1} + a_{i1}d_1 + \dots + a_{iN}d_N.$$

Using the induction hypothesis for (b), there exists T_{ij} in D and linear systems L_i such that $B_i = \sum y_{ij} T_{ij}$ is a solution of (A.5) for y_{ij} in U if and only if the d_1, \dots, d_N and y_{ij} satisfy L_i .

Determine $B_{1k_1-1}, \dots, B_{11}; B_{2k_2-1}, \dots, B_{2,1}; \dots; B_{sk_s-1}, \dots, B_{s1}$ as before until we obtain

$$\sum_{i=1}^s \frac{A_{i1}^{(**\dots)}}{p_i} + A_0 = \sum_{i=1}^s \frac{B_{i0}}{p_i} + B'_0 + (\sum e_j \log D_j)'$$

The $A_{i1}^{(**\dots)}$ and A_0 are linear polynomials in d_1, \dots, d_N . Let $p_i = \prod q_{ij}$ be the factorization of p_i into monic irreducible factors over $\bar{K}D$ and let degree $q_{ij} = n_{ij}$. We then have for each i

$$\frac{A_{i1}^{(**\dots)} + n_{ij} \zeta' (\sum c_{ij}) p_i}{p_i} = \frac{B_{i0}}{p_i} = \sum c_{ij} \frac{q'_{ij}}{q_{ij}}.$$

For each i , we get a linear system L_i^* in the c_{ij} and d_1, \dots, d_N by equating terms in the partial fraction decomposition. We finally must check to see that

$$A_0 - \zeta' \sum n_{ij} c_{ij} = (B_0 + \sum e_j \log D_j)'$$

Using the induction hypothesis for part (a), this gives a linear system L^{**} in d_1, \dots, d_N and the c_{ij} . We now get L by projecting $L_{-m} \cup \dots \cup L_k \cup L_1^* \cup \dots \cup L_s^* \cup L^{**}$ onto the first N variables d_1, \dots, d_N .

(b) Case 1. $\theta \in D$. For $y = A/p_1^{\alpha_1} \dots p_k^{\alpha_k}$ we proceed as in [RISC69, p. 184] to determine bounds for the α_i . Using these bounds we can set $y = Y/p_1^{\alpha_1} \dots p_k^{\alpha_k}$, substitute

in $y' + fy = \sum c_i q_i$, clear denominators and get

$$(A.6) \quad RY' + SY = \sum c_i T_i.$$

We set

$$\begin{aligned} Y &= y_\alpha \theta^\alpha + y_{\alpha-1} \theta^{\alpha-1} + \dots + y_0, \\ R &= r_\beta \theta^\beta + \dots + r_0, \\ S &= s_\gamma \theta^\gamma + \dots + s_0, \\ \sum c_i T_i &= t_\delta \theta^\delta + \dots + t_0, \end{aligned}$$

with y_j, r_j, s_j in D and t_j linear in the c_i with coefficients in D . Substituting these expressions in (A.6) and comparing powers of θ , we get: (1) when $y'_\alpha \neq 0$, either (a) $\alpha + \beta \leq \delta + 1$ or (b) $\alpha + \gamma \leq \delta + 1$ or (c) $\alpha + \beta = \alpha + \gamma > \delta + 1$; (2) when $y'_\alpha = 0$, either (a) $\alpha + \beta - 1 \leq \delta$ or (b) $\alpha + \gamma \leq \delta$ or (c) $\alpha + \beta - 1 = \alpha + \gamma > \delta$. Case (1a), (1b), (2a), and (2b) yield bounds for α .

Case (1c) occurs when $r_\beta y'_\alpha + s_\gamma y_\alpha = 0$ and $r_\beta y'_{\alpha-1} + s_\gamma y_{\alpha-1} + r_{\beta-1} y'_\alpha + (\alpha \theta' r_\beta + s_{\gamma-1}) y_\alpha = 0$. Letting $y_{\alpha-1} = v y_\alpha$ with $v \in D$ we have

$$\begin{aligned} r_\beta y_\alpha v' + (r_\beta y'_\alpha + s_\gamma y_\alpha) v + r_{\beta-1} y'_\alpha + (\alpha \theta' r_\beta + s_{\gamma-1}) y_\alpha &= 0, \\ v' - r_{\beta-1} s_\alpha / r_\beta^2 + s_{\gamma-1} / r_\beta + \alpha \theta' &= 0, \\ \left(\int \frac{r_{\beta-1} s_\gamma - r_\beta s_{\gamma-1}}{r_\beta^2} \right) - \alpha \theta &= v. \end{aligned}$$

We now deal with the cases when θ is nonsimple over D and when $\theta = \log \eta$ for some η in D (this is the only place where the hypothesis of a log-explicit extension comes into play). If θ is nonsimple over D , then using the induction hypothesis we find a linear system L in one indeterminate α such that α satisfies L if and only if

$$\left(\int \frac{r_{\beta-1} s_\gamma - r_\beta s_{\gamma-1}}{r_\beta^2} \right) - \alpha \theta$$

is elementary over D . Furthermore, there is at most one α in K satisfying L , since the existence of two such values would imply θ is simple. Therefore we can bound α in this case. If $\theta = \log \eta$ for some η in D , we use the original Risch Algorithm to determine α such that

$$\int \frac{r_{\beta-1} s_\gamma - r_\beta s_{\gamma-1}}{r_\beta^2} = v + \alpha \log \eta$$

for some v in D . If such an α exists it must be unique, otherwise $\log \eta$ would be in D . This allows us to again bound α .

To bound α in Case (2c), note that this case occurs when $r_\beta (y'_{\alpha-1} + \alpha \theta' y_\alpha) + s_\gamma y_\alpha = 0$, or

$$\left(\int \frac{s_\gamma}{r_\beta} \right) + \alpha \theta = \frac{-y_{\alpha-1}}{y_\alpha}.$$

Treating the nonsimple and logarithmic cases separately as in Case (1c) above yields the bound for α . The rest of the proof is the same as [RISC69, pp. 185–186]. \square

We can deduce the following corollary from Theorem A.1. By a regular Liouvillian extension we mean a Liouvillian extension (see the definition in § 4) where each θ used in building up the tower is transcendental over the preceding field.

COROLLARY A.2. *Let K be a finitely generated extension of Q and let $F = U(z, \theta_1, \dots, \theta_n)$ be a regular Liouvillian extension of $K(z)$, where $z' = 1$ and $c' = 0$ for all c in U . Let f_0, f_1, \dots, f_N be elements of F . Then one can determine in a finite number of steps a system of linear equations in N variables with coefficients in K so that $f_0 + d_1 f_1 + \dots + d_N f_N$ has an elementary integral for d_1, \dots, d_N in \bar{K} if and only if (d_1, \dots, d_N) satisfies L . For each (d_1, \dots, d_N) in \bar{K}^N satisfying L , we can find $v_0 \in F$, $v_i \in \bar{K}F$ for $i = 1, \dots, m$ and c_1, \dots, c_m in \bar{K} such that*

$$f_0 + d_1 f_1 + \dots + d_N f_N = v_0' + \sum_{i=1}^m c_i \frac{v_i'}{v_i}.$$

Proof. This follows from Theorem A.1 and the fact, shown in [ROCA79], that one can effectively embed a regular Liouvillian extension of $K(z)$ into a regular log-explicit extension of $K(z)$. \square

Since any purely exponential extension of $K(z)$ is a regular log-explicit extension of $K(z)$, Theorem A.1 gives the result needed in § 7. A result similar to Theorem A.1, for regular elementary extensions of $K(z)$ was stated and proven in [MACK76].

REFERENCES

- [KOL73] E. KOLCHIN, *Differential Algebra and Algebraic Groups*, Academic Press, New York, 1973.
- [LANG65] S. LANG, *Algebra*, Addison-Wesley, Reading, MA, 1965.
- [MACK76] C. MACK, *Integration of affine forms over elementary functions*, Computer Science Dept., Univ. of Utah, Technical Report, VCP-39, 1976.
- [MUM76] D. MUMFORD, *Algebraic Geometry I, Complex Projective Varieties*, Springer-Verlag, New York, 1976.
- [MOSE69] J. MOSES, *The integration of a class of special functions with the Risch algorithm*, SIGSAM Bull. 13 (1969), pp. 14–27.
- [MOZI79] J. MOSES AND R. ZIPPEL, *An extension of Liouville's Theorem* in Symbolic and Algebraic Computation, E. W. Ng, ed, Springer-Verlag, New York, 1979, pp. 426–430.
- [RISC69] R. H. RISCH, *The problem of integration in finite terms*, Trans. Amer. Math. Soc., 139 (1969), pp. 167–189.
- [RITT48] J. F. RITT, *Integration in Finite Terms*, Columbia Univ. Press, New York, 1948.
- [ROCA79] M. ROTHSTEIN AND B. F. CAVINESS, *A Structure theorem for exponential and primitive functions*, this Journal, 8 (1979), pp. 357–367.
- [ROS75] M. ROSENBLICHT, *Differential extension fields of exponential type*, Pacific J. Math., 57 (1975), pp. 289–300.
- [ROS76] ———, *On Liouville's theory of elementary functions*, Pacific J. Math., 65 (1976), pp. 485–492.
- [ROSI77] M. ROSENBLICHT AND M. F. SINGER, *On elementary, generalized elementary and Liouvillian extension fields*, in Contributions to Algebra, Bass, Cassidy and Kovacic, ed, Academic Press, New York, pp. 329–342.
- [SING77] M. F. SINGER, *Functions satisfying elementary relations*, Trans. Amer. Math. Soc., 227 (1977), pp. 185–206.
- [SSC81] M. F. SINGER, B. D. SAUNDERS AND B. F. CAVINESS, *An extension of Liouville's theorem on integration in finite terms*, Proc. 1981 ACM Symposium on Symbolic and Algebraic Computation, P. S. Wang, ed., Association for Computing Machinery, New York, 1981.
- [VDW50] B. L. VAN DER WAERDEN, *Modern Algebra, Vol. II*, Second Edition, Frederick Ungar, New York, 1950.

CONCURRENT PROBABILISTIC PROGRAMS, OR: HOW TO SCHEDULE IF YOU MUST*

SERGIU HART† AND MICHA SHARIR‡

Abstract. Consider a finite set of processes, such that each one may use randomizations in its course of execution; these processes are running concurrently, under a fair interleaving schedule. We analyze the worst-case probability of termination, i.e., program convergence to a specified set of goal states. Several methods for computing this probability are presented, and characterizations of the special case where it is identically 1 are derived. Specializations of these characterizations to the case of deterministic and nondeterministic programs, and to the case of programs with finite state spaces, are also discussed.

Key words. concurrent probabilistic program, scheduler, fairness, program termination, Markov chains

1. Introduction. This paper continues the study, begun in [HSP], of termination of concurrent probabilistic programs. The model that we assume is that of a finite set K of concurrent processes, each of which is allowed to use randomization, i.e., draw randomly according to probability distributions. These processes execute asynchronously, and we can thus consider each process $k \in K$ as a discrete Markov chain (with stationary transition probability matrix P^k) on the set I of common execution states. The overall execution behavior of these processes is described in terms of the interleaving pattern in which they are scheduled by some imaginary scheduler σ . Each process k scheduled at a state i can reach more than one subsequent state, so that to specify σ we need to consider all these transitions simultaneously. We may therefore represent σ as a tree (referred to as the *execution-tree* or the *transition-tree* induced by σ) each of whose nodes is labeled by a pair (i, k) , where $i \in I$ is the state reached at that node, and where $k \in K$ is the process to be scheduled there next. A node (i_1, k_1) will be a son of (i, k) in the tree if there exists a positive transition probability of reaching i_1 from i under a single execution step of process k , and if process k_1 will be next scheduled at i_1 , provided that this transition has indeed taken place.

Given such a σ , it induces in a standard manner a probability measure μ_σ on the space of all infinite sequences of states.

We consider here general schedules σ , with the sole restriction that they be *fair*, meaning that no process stops being scheduled; i.e., that the μ_σ -measure of the set of all tree paths on which each process $k \in K$ is scheduled infinitely often is 1.

This model is discussed and justified more fully in [HSP]. We note that it coincides with the model assumed by Lehmann and Rabin in [LR], and also with that used by Dubins and Savage [DS] in their study of optimal gambling strategies (with the essential exception that they do not require fairness). It does differ, though, from various other models used in the literature (cf. [Ra1], [Ra2], [RS1], [RS2]). The crucial distinction lies in the degree to which the imaginary scheduler can base its scheduling decisions on the outcome of random draws made by the processes, or, more generally, on their internal states. These more restrictive scheduling models usually correspond to situations in which the execution time of a single step of a process is independent of its current state and of the outcome of the random draws it has made. Our model is more general, and allows for such dependence, thereby being a more realistic model for

* Received by the editors September 15, 1982, and in revised form August 1, 1984.

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

‡ The work of this author was supported in part by the Bat-Sheva Fund at Tel Aviv University, and by the Office of Naval Research under grant N00014-75-C-0571 at the Courant Institute.

general concurrent or distributed probabilistic execution. Moreover, properties established for concurrent probabilistic programs under our model will continue to hold under the more restrictive models mentioned above, but not necessarily vice-versa (for example, Rabin's synchronization algorithm described in [Ra1] is shown in [HSP] to fail in our model).

In the preceding paper [HSP], we have analyzed *termination* of concurrent probabilistic programs having a *finite* state space. We have obtained there necessary and sufficient conditions for such a program to reach (with probability 1) a given set X of *goal states* from some initial state, under any fair schedule. These conditions can be checked mechanically, and are independent of the particular values of nonzero transition probabilities of the processes involved.

In this paper we generalize and extend these results to programs with *infinite* state spaces. As in the case of a single Markov chain, the analysis of program termination becomes much more complicated in the general case, and becomes dependent upon the actual values of the nonzero transition probabilities involved. The basic problem that we treat in this paper is the computation of the function φ on the set of states I , where, for each $i \in I$, $\varphi(i)$ is the minimum probability of program termination starting at state i , under any fair schedule. We establish various properties and characterizations of φ , and derive from them several techniques for the calculation of this function. This theory enables us to gain a better understanding of the structure of the (worst-case) convergence of the program towards termination. For example, one can interpret this convergence process as a game between the program and the scheduler, in which each move of the program requires the scheduler to schedule one of the processes and the scheduler responds by scheduling this process eventually, but only after scheduling some other processes prior to it, in a way which would hurt as much as possible the program's probability to terminate. We show that the optimal payoff for the program in this game is the function φ , provided that the game is long enough, where the length of such a game is measured by some (infinite) ordinal.

The various characterizations of φ are next used to obtain necessary and sufficient conditions for the special case $\varphi \equiv 1$ (i.e. for worst-case almost-sure termination from any initial state) to hold. Some of these conditions generalize similar conditions given in the preceding paper [HSP] for programs with finite state spaces. These characterizations of program termination are next specialized to the case in which the processes are deterministic or nondeterministic.¹ Some of these characterizations are shown to reduce to the conditions given by Lehmann, Pnueli and Stavi [LSP] for the termination of nondeterministic programs, while others are new. Finally, the special case of probabilistic programs with finite state spaces is reconsidered from the viewpoint of the general theory developed in this paper, enabling us to obtain the decomposition of the state space described in [HSP] in a different manner. The results of this paper are exemplified on several running example programs. The techniques developed in this paper can be immediately interpreted as (sound and complete) proof methods for almost sure program termination.

This paper is organized as follows: Section 2 presents the notations and terminology used in the paper, and begins the analysis of φ by establishing some more elementary properties of this function. Section 3 develops the main technical tools for the analysis

¹ A nondeterministic program is one where each execution step of any of its processes may lead from a state $i \in I$ to several succeeding states, but where there is no probability distribution associated with these states; instead, each of these succeeding states must be considered as being potentially the sole successor of i . Such a program is said to terminate if *every* execution sequence terminates.

and characterizations of φ , and obtains φ as the limit of a certain transfinite sequence of functions. Section 4 gives further characterizations of φ . Section 5 treats the special case $\varphi \equiv 1$ (i.e. of almost-sure worst-case termination), and derives various characterizations of this property. Section 6 specializes the preceding results to the case of deterministic and nondeterministic programs. The new characterization of termination of such programs is also given a direct proof. Section 7 treats the special case of probabilistic programs with finite state spaces. Some concluding remarks are presented in § 8.

2. Preliminaries. In this section we present our model of probabilistic concurrent programs in more precise terms, introduce some notations, and establish several preliminary properties of the worst-case termination probability of the program.

A *concurrent probabilistic program* consists of a finite set K of processes acting on a *state space* I ; each $i \in I$ is a common execution state of the processes, and is specified by the program location at each process, by the values of all variables—shared and private—etc. Each $k \in K$ can be regarded as a stationary discrete Markov chain on I . (This extra restriction of discreteness, which is quite adequate for actual programs, simplifies the analysis considerably, by avoiding the technical difficulties of treating non σ -additive measures, which would be otherwise necessary as in Dubins and Savage [DS].) Under this assumption, each process $k \in K$ is specified in terms of its transition probability matrix P^k , that is, for each $i, j \in I$, $P^k_{i,j}$ is the probability of reaching state j from state i in a single (indivisible) execution step of process k . The nonnegative matrix P^k is stochastic: for each i , $P^k_{i,j} > 0$ for at most countably many j , and $\sum_{j \in I} P^k_{i,j} = 1$.

As already stated, program execution is assumed to consist of *interleaving* execution steps of the processes, each executing in its turn one indivisible step. Let $i \in I$ be an initial execution state. Let $H(i)$ denote the set of all *finite execution histories* starting at i ; formally,

$$H(i) = \{i\} \times \left(\bigcup_{n=0}^{\infty} I^n \right).$$

An (infinite) *schedule* σ starting at i is simply defined as a function $\sigma: H(i) \rightarrow K$, that is, for each finite history $h \in H(i)$, $\sigma(h)$ is the next process to perform an execution step, given that execution has proceeded so far through the states in h . The set of all schedules starting at i will be denoted by $\Sigma(i)$. To each such schedule σ there corresponds an *execution tree*, defined inductively as follows. Each node of this tree is labelled by a pair (j, k) where j is the current execution state, and k the next process to be scheduled in this node. The root of the tree is labelled by $(i, \sigma(i))$. For each node ν in the tree, let $h \in H(i)$ be the sequence of states along the path from the root to ν , let j be the last state in h , and let $k = \sigma(h)$; then ν is labelled by (j, k) , and its sons are nodes labelled by $(j', \sigma(h, j'))$, (where (h, j') is the concatenation of j' to h) for $j' \in I$ such that $P^k_{j,j'} > 0$.

Let $H^*(i)$ denote the set of all *infinite execution histories* starting at i , that is,

$$H^*(i) = \{i\} \times I^\infty \quad \left(\text{where } I^\infty = \prod_{n=1}^{\infty} I \right).$$

Each schedule $\sigma \in \Sigma(i)$ induces a probability measure μ_σ on the cylindrical σ -field on $H^*(i)$, such that for each cylinder $(i, i_1, i_2, \dots, i_n)$, consisting of all histories whose initial $n+1$ states are i, i_1, \dots, i_n ,

$$\mu_\sigma\{(i, i_1, \dots, i_n)\} = \prod_{s=0}^{n-1} P^k_{i_s, i_{s+1}},$$

where $i_0 = i, k_s = \sigma(i_0, i_1, \dots, i_s)$. Expectation with respect to μ_σ will be denoted by E_σ .

Let $H^* = \bigcup_{i \in I} H^*(i)$. Throughout the paper we will use the following notational convention: Elements of H^* —which we call paths or histories—will be denoted by π ; for each such π and each $n \geq 0$, the $(n + 1)$ th state along π will be denoted by i_n , and the subpath consisting of the first $n + 1$ states in π will be denoted by $\pi_n \equiv (i_0, i_1, \dots, i_n)$. A path π is a *fair path* with respect to a given schedule σ if each $k \in K$ appears infinitely often in the sequence $\{\sigma(\pi_n)\}_{n=0}^\infty$; the schedule σ is a *fair schedule* if $\mu_\sigma\{\pi: \pi \text{ is fair}\} = 1$. For each $i \in I$ we denote

$$\Sigma F(i) = \{\sigma \in \Sigma(i): \sigma \text{ is fair}\}.$$

Let $X \subset I$ be a given set of *goal states*, fixed henceforth. Our aim is to study the convergence of program execution to states in X ; we will therefore assume in the sequel, without loss of generality, that all states in X are *absorbing* for each $k \in K$; i.e., that $P_{i,i}^k = 1$ for each $i \in X$ and each $k \in K$.

The basic problem studied in this paper is that of analyzing and computing the worst-case probability of the program to reach X (i.e., to *terminate*) when executed from a given initial state under a fair schedule. To formalize this notion, let χ_X be the characteristic function of X (defined on I); we extend this function to H^* by putting $\chi_X(\pi) = \lim_{n \rightarrow \infty} \chi_X(i_n)$ (recall that $\pi = (i_n)_{n \geq 0}$). Since X is absorbing, $\chi_X(\pi) = 1$ if X is ever reached along π , and 0 otherwise. The probability of reaching X under σ is then simply $E_\sigma(\chi_X)$. The following standard observation, which also establishes the measurability of the extended χ_X , will be quite useful in the sequel: For each $n \geq 0$ define a “truncated” extension $\chi_X^{(n)}$ of χ_X by putting $\chi_X^{(n)}(\pi) = 1$, if X is reached during the first n steps of π , and 0 otherwise. Then $E_\sigma(\chi_X^{(n)})$ is the probability of reaching X during the first n steps of σ , and we have

$$\lim_{n \rightarrow \infty} E_\sigma(\chi_X^{(n)}) = \sup_n E_\sigma(\chi_X^{(n)}) = E_\sigma(\chi_X).$$

The worst-case termination probability that we seek is defined, for each initial state $i \in I$, as

$$\varphi(i) = \inf_{\sigma \in \Sigma F(i)} E_\sigma(\chi_X).$$

We will shortly establish several preliminary properties of the function φ , but first we introduce additional notations concerning finite portions of program execution. Let \mathbb{N} denote the set of nonnegative integers, and out $\bar{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$. A *stopping time* N is a mapping from H^* into $\bar{\mathbb{N}}$ such that if $N(\pi) = m$ then $N(\pi') = m$ for each path π' which coincides with π at all steps up to, and including m . In other words, $N(\pi)$ may depend only on i_0, i_1, \dots, i_N —i.e., on states visited before this step, but not on future steps (i.e., on i_{N+1}, \dots). A *finite subschedule at* $i \in I$ is a pair $\tau = (\sigma, N)$ where $\sigma \in \Sigma(i)$, and where N is a stopping time on $H^*(i)$ such that $\mu_\sigma(N < \infty) = 1$ (this corresponds to the notion of “policy” of Dubins and Savage [DS]). The intuitive meaning of such a pair is the initial portion of σ up to, and including N ; in particular, the actual value of σ is relevant only up to the stopping time N . The set of all finite subschedules at i will be denoted by $T(i)$ (note that the empty subschedule—i.e., when $N \equiv 0$ —is included in $T(i)$).

In the sequel we will occasionally use the following standard decomposition of an infinite schedule $\sigma \in \Sigma(i)$: Let N be a stopping time, $\mu_\sigma(N < \infty) = 1$; then σ is equivalent to its initial portion $\tau = (\sigma, N)$, followed by the collection of continuation schedules; that is, for each $\pi \in H^*(i)$ (with $N < \infty$), the continuation $\sigma_{\pi_N} \in \Sigma(i_N)$ of σ after the *end state* i_N of τ . Note in particular that σ is fair if and only if each of the continuations σ_{π_N} is fair.

Let α be a real function on I . Then for each finite subschedule $\tau = (\sigma, N) \in T(i)$, the expectation of α with respect to τ is defined as

$$E_\tau(\alpha) = E_\sigma(\alpha(i_N)).$$

For example, let $\sigma \in \Sigma(i)$, and define, for each $n \geq 0$, $\tau_n = (\sigma, n) \in T(i)$ (i.e., with the stopping time $N \equiv n$). Then, as already noted, $E_\sigma(\chi_X) = \lim_{n \rightarrow \infty} E_{\tau_n}(\chi_X)$. Note also that $E_{\tau_0}(\alpha) = \alpha(i)$, and that $E_{\tau_k}(\alpha) = (P^k \alpha)(i)$, where $k = \sigma(i)$.

Having introduced all the required terminology, we begin by establishing a few elementary properties of the function φ .

PROPOSITION 2.1. (a) $\varphi \geq 0$; $\varphi|_X \equiv 1$.

(b) $\varphi(i) = \min_{k \in K} (P^k \varphi)(i)$ for each $i \in I$.

Proof. (a) is trivial, since X is absorbing; note also that $\varphi \leq 1$.

(b) To show that $\varphi(i) \leq (P^k \varphi)(i)$ for $k \in K$, $i \in I$, use a schedule $\sigma \in \Sigma F(i)$ which starts by scheduling k at i , and then continues so as to approximate φ at each of the resulting states. For the converse inequality, take a sequence of schedules σ_n in $\Sigma F(i)$ such that $E_{\sigma_n}(\chi_X)$ converges to $\varphi(i)$ and such that they all start by scheduling the same process $k \in K$ (since K is finite this is always possible); then it is easily seen that $(P^k \varphi)(i) \leq \varphi(i)$. More details can be found in [HS]. Q.E.D.

Extending standard notations in Markov chain theory, we say that a real function α on I is *subharmonic* if $\alpha \leq P^k \alpha$ for each $k \in K$. Similarly α will be called *min-harmonic* if $\alpha = \min_{k \in K} P^k \alpha$ (note that each min-harmonic function is subharmonic).

In the special case where K contains a single process k , the function φ is *harmonic* (i.e., $\varphi = P^k \varphi$). Moreover, it is well-known (cf. [SPH] for example) that φ is the smallest nonnegative harmonic function which is 1 on X . This might lead us to conjecture that for a general (finite) K , φ is also the smallest nonnegative min-harmonic function which is 1 on X . This, however, is not true in general, as can be seen from the following simple example: Let $I = \{0, 1\}$, $X = \{0\}$, and $K = \{1, 2\}$, with the nonzero transition probabilities $P_{1,0}^1 = P_{1,1}^2 = 1$. Obviously, any fair execution of this program brings it into X with certainty, so that $\varphi \equiv 1$, yet the function $\psi(0) = 1$, $\psi(1) = 0$ is a smaller nonnegative min-harmonic function which is 1 on X . The reason for this phenomenon is that fairness is not directly connected to the min-harmonicity of φ . Indeed, let us define a function ψ on I by

$$\psi(i) = \inf_{\sigma \in \Sigma(i)} E_\sigma(\chi_X), \quad i \in I.$$

(i.e., infimum over *all* schedules, not necessarily fair). Then it can be shown that

PROPOSITION 2.2. ψ is the smallest nonnegative min-harmonic function which is 1 on X .

Our next result is a strong form of a “zero-one law” for φ , which generalizes the zero-one law established in [HSP] for finite state spaces.

THEOREM 2.3 (zero-one law). $\inf_{i \in I} \varphi(i)$ is either 0 or 1. Moreover, for each $i \in I$ and $\sigma \in \Sigma F(i)$ define a sequence $\{f_n\}_{n \geq 0}$ of functions on $H^*(i)$ by putting $f_n(\pi) = \varphi(i_n)$, $\pi \in H^*(i)$, $n \geq 0$. Then $\{f_n\}$ converges μ_σ -a.s. to χ_X (extended to $H^*(i)$).

Proof. Let $i \in I$ and $\sigma \in \Sigma F(i)$ be given. The subharmonicity of φ implies that the sequence $\{f_n\}$ is a submartingale² on $H^*(i)$. Since $0 \leq f_n \leq 1$ for each $n \geq 0$, it follows from the (sub)martingale convergence theorem that $\{f_n\}$ converges μ_σ -a.s. to a limit f_∞ . Put $\tau_n = (\sigma, n)$, $n \geq 0$. Then

$$E_\sigma(\chi_X) = E_{\tau_n}(E_{\sigma_{\tau_n}}(\chi_X)) \geq E_{\tau_n}(\varphi) = E_\sigma(f_n)$$

² i.e., for all $n \geq 0$, $E_\sigma(f_{n+1} | \pi_n) \geq f_n$, where π_n is any history of length n with $\mu_\sigma(\pi_n) > 0$.

(since each σ_{π_n} is fair). Letting $n \rightarrow \infty$, we obtain

$$E_\sigma(\chi_X) \geq E_\sigma(f_\infty) \geq 1 \cdot \mu_\sigma\{\pi: f_\infty(\pi) = 1\}.$$

But for each $\pi \in H(i)$, if X is ever reached along π then $f_n(\pi) = \varphi(i_n) = 1$ for all sufficiently large n , so that $f_\infty(\pi) = 1$. Thus

$$\mu_\sigma\{\pi: f_\infty(\pi) = 1\} \geq \mu_\sigma\{\pi: X \text{ reached along } \pi\} = E_\sigma(\chi_X).$$

Therefore we must have equalities throughout; that is

$$E_\sigma(\chi_X) = E_\sigma(f_\infty) = \mu_\sigma\{\pi: f_\infty(\pi) = 1\}.$$

This, however, implies that f_∞ is almost everywhere either 0 or 1, and that $f_\infty(\pi) = 1$ if and only if X is ever reached along π . The zero-one law is now immediate, because if φ is not identically 1, take $i \in I$, $\sigma \in \Sigma F(i)$ such that $E_\sigma(\chi_X) = c < 1$. Then $\varphi(i_n) \rightarrow 0$ on a set of paths whose μ_σ -measure is $1 - c > 0$, thus there exists states with arbitrarily small φ , or $\inf_{i \in I} \varphi(i) = 0$. Q.E.D.

As a final preliminary note, we would like to point out that, unlike the case of a finite state space, the actual values of nonzero transition probabilities of the processes involved can have significant influence on the termination probabilities φ . This is indeed well known even for a single Markov chain. (Consider e.g. the case of a random walk on the nonnegative integers, where the “leftward” transition probability is p . Then the probability of converging towards 0 is identically 1 if $p \geq \frac{1}{2}$, and is exponentially decreasing otherwise; cf. [Ch] for details). Thus, for infinite state spaces there is no hope to obtain purely combinatorial analysis techniques (as have been developed in [HSP] for finite state spaces), and more complex techniques are needed. Development of such techniques is indeed the main purpose of the present paper.

3. φ -iterates. Direct calculation of the function φ from its definition is rather complicated. The purpose of this section is to develop machinery needed for a simpler calculation and characterization of φ . Specifically, we will show that φ is the limit of a transfinite sequence of iterates of a certain operator. We will call these φ -iterates.

DEFINITION. We define an operator Q , and an auxiliary set of operators $\{Q^k\}_{k \in K}$, on the space of all bounded real functions on I , as follows: For each bounded real function α on I , each $i \in I$, and each $k \in K$, put

$$(Q^k \alpha)(i) = \inf_{\tau \in T(i,k)} E_\tau(\alpha),$$

where $T(i, k)$ is defined as

$$\{(\sigma, N + 1): \sigma \in \Sigma(i), N \text{ an a.s. finite stopping time with } \sigma(\pi_N) \equiv k\} \subset T(i);$$

i.e., $T(i, k)$ is the set of all subschedules which start at i , schedule k eventually almost surely, and stop right after scheduling k . Q is then defined as

$$(Q\alpha)(i) = \max_{k \in K} (Q^k \alpha)(i).$$

Let R be any of the operators Q^k or Q ; then plainly R is *monotone* (i.e., $\alpha_1 \leq \alpha_2$ implies $R\alpha_1 \leq R\alpha_2$), $R0 = 0$, and $R1 = 1$. The following lemma gives two characterizations of the operators Q^k , one of which is constructive while the other is not.

LEMMA 3.1. *Let α be a bounded real function on I . For each $k \in K$, $Q^k \alpha$ is the largest subharmonic function which does not exceed $P^k \alpha$ (i.e., (1) $Q^k \alpha \leq P^k \alpha$, (2) $Q^k \alpha$ is subharmonic, and (3) if $\beta \leq P^k \alpha$ is subharmonic, then $\beta \leq Q^k \alpha$). Furthermore, $Q^k \alpha$*

is the limit (or infimum) of the following nonincreasing sequence of functions:

$$\beta_1(i) = (P^k\alpha)(i), \quad i \in I,$$

$$\beta_{n+1}(i) = \min \left\{ \beta_n(i), \min_{l \in K} (P^l\beta_n)(i) \right\}, \quad i \in I, \quad n \geq 1.$$

Proof. Let β be the limit of the nonincreasing sequence $\{\beta_n\}$; then β is the largest subharmonic function $\leq P^k\alpha$. The rest follows by noting that, for each $n \geq 1$

$$\beta_n(i) = \inf_{\tau \in T_n(i,k)} E_\tau(\alpha),$$

where $T_n(i, k)$ consists of those subschedules in $T(i, k)$ which stop after at most n steps. Q.E.D.

LEMMA 3.2. For each subharmonic function α and each $k \in K$ we have

$$\alpha \leq Q^k\alpha \leq P^k\alpha.$$

Proof. By Lemma 3.1, $Q^k\alpha$ is the largest subharmonic function which is $\leq P^k\alpha$. Since α itself is subharmonic we have $\alpha \leq P^k\alpha$, so that $\alpha \leq Q^k\alpha$. Q.E.D.

DEFINITION. For each ordinal a we define on I real functions γ_a and γ_a^k , $k \in K$, by the following transfinite inductive process:

$$\gamma_0 = \gamma_0^k = \chi_X, \quad k \in K,$$

$$\gamma_a^k = \sup_{b < a} Q^k\gamma_b \quad \text{for each ordinal } a > 0 \text{ and } k \in K,$$

$$\gamma_a = \max_{k \in K} \gamma_a^k \quad \text{for each ordinal } a.$$

The functions γ_a^k and γ_a are called the φ -iterates of order a of the program (the reason for this terminology will be apparent at the end of this section).

Since X is absorbing, $Q^k\chi_X \cong \chi_X$, thus $\gamma_1^k \cong \gamma_0^k$ for each $k \in K$, hence $\gamma_1 \cong \gamma_0$. Also, by definition, $\gamma_a^k \cong \gamma_b^k$ for each pair of ordinals $a > b > 0$. Thus, for each $k \in K$ the transfinite sequence $\{\gamma_a^k\}_{a \geq 0}$ is nondecreasing, and so is the sequence $\{\gamma_a\}_{a \geq 0}$. From this it follows that $\gamma_{a+1}^k = Q^k\gamma_a^k$ for each ordinal a , and that $\gamma_a = \sup_{b < a} \gamma_b$ for limit ordinals a .

Since the transfinite sequence $\{\gamma_a\}_{a \geq 0}$ is nondecreasing, and each of its elements is obviously bounded between 0 and 1, this sequence must converge to a limit function γ , and there must exist an ordinal c such that $\gamma_c = \gamma$. (Indeed, for each $i \in I$ the transfinite sequence $\{\gamma_a(i)\}$ is a nondecreasing and bounded sequence of real numbers, and so must attain its supremum at some ordinal c_i ; the required ordinal c is simply $\sup_{i \in I} c_i$.) Obviously $Q\gamma = Q\gamma_c = \gamma_{c+1} = \gamma_c = \gamma$. Moreover, using standard fixpoint arguments, it is easily seen by transfinite induction that γ is the smallest fixpoint of Q which is $\cong \chi_X$.

Remarks. (1) To motivate these definitions, it is helpful to consider the following interpretation of the functions γ_a and γ_a^k : $\gamma_0(i)$ is just an indication whether $i \in X$. $\gamma_1^k(i)$ is the smallest probability of reaching X by any subschedule which starts execution at i , and is forced to schedule k eventually (a.s.). Thus $\gamma_1(i)$ is the smallest probability of reaching X that must be yielded by any subschedule starting at i which is forced to schedule any one of the processes at least once. Arguing inductively, $\gamma_n(i)$ is the smallest probability of reaching X that must be yielded by any subschedule starting at i which has to schedule any sequence of n processes one after the other. (Note that this sequence need not be specified in advance; rather the first process k_1

to be scheduled is specified, then the second process to be scheduled is specified, but it may depend on the state reached after scheduling k_1 , and so on.)

(2) $\gamma_n(i)$ can be viewed as the minmax value of a two-person zero-sum game $\Gamma_n(i)$. In this game, the aim of the first player, called “player X,” is to reach X during program execution with the highest possible probability, whereas the aim of the second player, called “the scheduler,” is to prevent the program from reaching X as much as possible. The game $\Gamma_n(i)$ consists of n stages. Each stage starts at some state $j \in I$ (stage 1 starts at i). Player X chooses some $k \in K$, and then the scheduler chooses some $\tau \in T(j, k)$. The program is then run according to τ ; when it stops, the next stage is played. After n such stages, player X receives a payoff of one unit from the scheduler if a state in X has been reached, and zero otherwise.

This interpretation can be extended to higher-order ordinals. Specifically, for each ordinal a we define a collection of games $\Gamma_a(i)$, for each $i \in I$, in the following transfinite inductive manner:

(i) $\Gamma_0(i)$ is the “empty” game; player X receives a payoff of 1 from the scheduler if $i \in X$, and zero otherwise.

(ii) If a is not a limit ordinal, say $a = b + 1$, player X first chooses a process k and then the scheduler chooses a subschedule $\tau \in T(i, k)$, and the program is run according to τ ; for each end state j of τ , the game continues as $\Gamma_b(j)$.

(iii) If a is a limit ordinal, player X first chooses an ordinal $b < a$, and then the game continues as $\Gamma_b(i)$.

The definitions (ii) and (iii) imply that after each stage, games with smaller ordinals are played; since every strictly decreasing sequence of ordinals is finite, it follows that every play of any of these games is finite, so that Γ_0 is reached eventually, and the payoff is therefore well defined. Moreover, by the definition of the sequence $\{\gamma_a\}_{a \geq 0}$, one easily obtains by transfinite induction, that $\gamma_a(i)$ is precisely the value of $\Gamma_a(i)$. Indeed, an ε -optimal strategy for player X is constructed as follows (for each $\varepsilon > 0$): If $a = b + 1$, player X first chooses $k \in K$ for which $\gamma_a(i) = \gamma_a^k(i)$, and from each end state j of the subschedule $\tau \in T(i, k)$ subsequently chosen by the scheduler, he continues with an ε -optimal strategy of $\Gamma_j(b)$. If a is a limit ordinal, player X first chooses an ordinal $b < a$ such that $\gamma_a(i) - \gamma_b(i) < \varepsilon/2$, and then continues with an $\varepsilon/2$ -optimal strategy of $\Gamma_b(i)$. As for the scheduler, at the first ordinal $b + 1 \leq a$ where he is called upon to move, he chooses $\tau \in T(i, k)$ such that $E_\tau(\gamma_b) - \varepsilon/2 < (Q^k \gamma_b)(i) = \gamma_{b+1}^k(i)$, and then he continues with an $\varepsilon/2$ -optimal strategy in the corresponding $\Gamma_b(j)$.

Furthermore, the ordinal c (at which $\gamma_{c+1} = \gamma_c$ is first obtained) is such that the expected payoff that player X can guarantee in the game $\Gamma_c(i)$ is the largest possible among all games $\{\Gamma_a(i)\}_{a \geq 0}$ —uniformly in the initial state i . As we shall see later in this section, this maximum payoff is exactly $\varphi(i)$.

(3) Note that if Q were σ -order continuous, i.e., if for any nondecreasing sequence $\{h_n\}$ of uniformly bounded functions we had

$$Q\left(\sup_n h_n\right) = \sup_n Qh_n,$$

then convergence of the γ_a 's would be attained at $c = \omega$ (the first infinite ordinal) or earlier. This is indeed so when I is a finite set, since then each such sequence $\{h_n\}$ converges uniformly to its supremum, in which case Q is clearly continuous. However, this does not hold in general, and so higher ordinals may be needed. (A similar phenomenon is noted by Lehmann, Pnueli and Stavi [LPS] concerning nondeterministic concurrent programs; see § 6 for a detailed comparison between their technique and ours.)

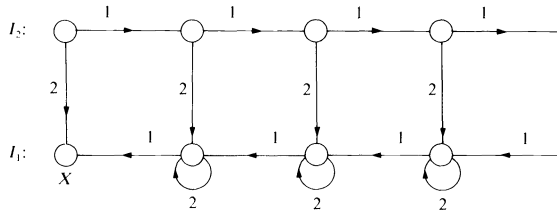
To illustrate the possible discontinuity of Q (and hence the need for higher ordinals), consider the following example (in which both processes involved are actually deterministic).

Example 1. Let $K = \{1, 2\}$, and let $I = I_1 \cup I_2$, where $I_1 = \mathbb{N} \times \{1\}$, $I_2 = \mathbb{N} \times \{2\}$, and $X = \{(0, 1)\}$. The nonzero transition probabilities are

$$P^1_{(n,1),(n-1,1)} = P^2_{(n,1),(n,1)} = 1, \quad n > 0,$$

$$P^1_{(n,2),(n+1,2)} = P^2_{(n,2),(n,1)} = 1, \quad n \geq 0.$$

These transitions are displayed in the following diagram:



It is easily seen that

$$\gamma_n(i, 1) = \begin{cases} 0, & i \geq n, \\ 1, & i < n, \end{cases} \quad i \in \mathbb{N}.$$

By definition of γ_ω we thus have

$$\gamma_\omega(i, 1) = 1, \quad i \in \mathbb{N}.$$

On the other hand, $\gamma_n(i, 2) = 0$ for each $i, n \in \mathbb{N}$ (to obtain $(Q^2 \gamma_n)(i, 2)$, schedule process 1 sufficiently many times so as to reach a state $(j, 2)$ with $j \geq n$, and then schedule process 2). Thus

$$\gamma_\omega(i, 2) = 0, \quad i \in \mathbb{N}.$$

But $\gamma_{\omega+1} = Q\gamma_\omega > \gamma_\omega$. Indeed, for each $(i, 2) \in I_2$ we have

$$\gamma_{\omega+1}(i, 2) = (Q\gamma_\omega)(i, 2) = (Q^2 \gamma_\omega)(i, 2) = \gamma_\omega(j, 1) = 1 \quad (\text{where } j \geq i).$$

Thus $\gamma_{\omega+1} \equiv 1$, and convergence of the φ -iterates is attained at the ordinal $\omega + 1$.

Remarks. (1) In the game-theoretic terminology established earlier, player X cannot achieve a nonzero payoff in any of the games $\Gamma_n(i, 2)$, $n \in \mathbb{N}$, or even $\Gamma_\omega(i, 2)$, because if the number of rounds n is fixed in advance, the scheduler will initially schedule process 1, $n + 1$ times, and this will prevent player X from reaching X in n moves. On the other hand, a payoff of 1 is guaranteed in $\Gamma_{\omega+1}(i, 2)$ as follows: Player X first chooses process 2; no matter what subschedule in $T((i, 2), 2)$ will be chosen by the scheduler, it will end at some state $(j, 1)$ in I_1 , and the game continues from there as $\Gamma_\omega(j, 1)$. Now player X chooses the ordinal $j < \omega$, and this guarantees its entry into X after j additional moves, by requiring to schedule process 1 in each of these moves.

(2) One can easily obtain along similar lines examples where higher and higher ordinals are needed to attain convergence.

(3) If we take in Example 1, $P^1_{(1,1),(0,1)} = \frac{1}{2}$ (instead of 1) and $P^1_{(1,1)(0,2)} = \frac{1}{2}$ (instead of 0), it can be verified that the first ordinal c where $\gamma_c = \varphi \equiv 1$ is $c = \omega^2$.

The main purpose of this section is to prove that $\gamma = \varphi$. The proof of this assertion is quite involved and will be split into proving both inequalities $\gamma \leq \varphi$ and $\gamma \geq \varphi$. It consists of the following sequence of lemmata.

LEMMA 3.3. *Each of the φ -iterates γ_a^k and γ_a is subharmonic.*

Proof. Lemma 3.1 and the fact that the supremum of subharmonic functions is subharmonic. Q.E.D.

LEMMA 3.4. *For each $k \in K$, $Q^k \varphi \leq \varphi$.*

Proof. For each $\sigma \in \Sigma F(i)$ let N be the first time k is scheduled, and let $\tau = (\sigma, N+1) \in T(i, k)$. Then, if $\sigma_{\pi_{N+1}}$ denotes the continuation of σ after the end of τ , we have

$$\begin{aligned} E_\sigma(\chi_X) &= E_\tau[E_{\sigma_{\pi_{N+1}}}(\chi_X)] \quad (\text{because } N+1 \text{ is a stopping time}) \\ &\geq E_\tau[\varphi(i_{N+1})] \quad (\text{because } \sigma_{\pi_{N+1}} \in \Sigma F(i_{N+1})) \\ &\geq (Q^k \varphi)(i) \quad (\text{by definition of } Q^k). \end{aligned}$$

Since this holds for each $\sigma \in \Sigma F(i)$, we have $\varphi(i) \geq (Q^k \varphi)(i)$. Q.E.D.

PROPOSITION 3.5. $\varphi = Q\varphi = Q^k \varphi$, for each $k \in K$.

Proof. By the preceding lemma, $Q\varphi = \max_{k \in K} Q^k \varphi \leq \varphi$. On the other hand, for each $k \in K$, $Q^k \varphi \geq \varphi$ by Lemma 3.2, since φ is subharmonic by Proposition 2.1(b). Q.E.D.

LEMMA 3.6. $\gamma \leq \varphi$.

Proof. We will show, using transfinite induction, that $\gamma_a \leq \varphi$ for each ordinal a . For $a = 0$, $\varphi \geq \chi_X = \gamma_0$ (see Proposition 2.1(a)). Assume $\gamma_b \leq \varphi$ for each $b < a$; then $Q^k \gamma_b \leq Q^k \varphi = \varphi$ by the preceding proposition, thus $\gamma_a^k \leq \varphi$ for each $k \in K$, so that $\gamma_a \leq \varphi$. Thus $\gamma = \gamma_c \leq \varphi$. Q.E.D.

LEMMA 3.7. $\varphi \leq \gamma$.

Proof. Note that, since $\gamma = \max_{k \in K} Q^k \gamma$, we have

$$(*) \quad \gamma(i) \geq \inf_{\tau \in T(i,k)} E_\tau(\gamma), \quad i \in I, \quad k \in K,$$

(actually, with equality holding for at least one k , although we will not make use of this fact). Let $i \in I$ be given. Choose $\varepsilon > 0$ and a sequence $\varepsilon_n \downarrow 0$ such that $\sum_n \varepsilon_n = \varepsilon$. Let $\{k_n\}_{n \geq 1}$ be a fixed sequence of processes in which each $k \in K$ appears infinitely many times. We will use (*) to construct a fair schedule σ starting at i by building it layer-by-layer from subschedules, as follows: Suppose that the first n layers of σ have already been constructed, the union of which being some subschedule τ_n starting at i (initially, τ_0 is "empty"). The $(n+1)$ th layer of σ is defined by appending to τ_n at each of its end nodes j a subschedule $\rho_j \in T(j, k_{n+1})$ such that

$$\gamma(j) \geq E_{\rho_j}(\gamma) - \varepsilon_n$$

(such a subschedule exists by (*)). Repeating this process inductively, we obtain the required (infinite) schedule σ , which is fair by our choice of the sequence $\{k_n\}_{n \geq 1}$.

Let $\{N_n\}_{n \geq 0}$ be the increasing sequence of stopping times defined by our construction; namely—the n th layer (i.e., τ_n) ends at N_n (in particular $N_0 \equiv 0$). For each $n \geq 0$ define the function

$$g_n(\pi) = \gamma(\pi_{N_n}), \quad \pi \in H^*(i);$$

in particular, $g_0 \equiv \gamma(i)$. By the choice of the subschedules ρ_j we have

$$(**) \quad g_n \equiv E_\sigma(g_{n+1} | \pi_{N_n}) - \varepsilon_{n+1}, \quad n \geq 0.$$

Hence, the sequence of functions $\{g'_n\}_{n \geq 0}$ given by

$$g'_n \equiv g_n - \sum_{m=1}^n \varepsilon_m, \quad n \geq 0$$

forms a supermartingale, which is bounded between 1 and $-\varepsilon$. Hence it converges almost surely to a limit g'_∞ , so that $\{g_n\}$ converges almost surely to the function

$$g_\infty \equiv g'_\infty + \sum_{m=1}^\infty \varepsilon_m = g'_\infty + \varepsilon.$$

Note that $\gamma|_X \equiv 1$; thus, if X is reached along π , then $g_\infty(\pi) = 1$, because for all sufficiently large n we will have $g_n(\pi) = 1$. Hence, by (**),

$$\begin{aligned} \gamma(i) &= g'_0 \equiv E_\sigma(g'_\infty) = E_\sigma(g_\infty) - \varepsilon \\ &\equiv \mu_\sigma(g_\infty = 1) - \varepsilon \\ &\equiv \mu_\sigma(X \text{ is reached}) - \varepsilon \\ &= E_\sigma(\chi_X) - \varepsilon \equiv \varphi(i) - \varepsilon. \end{aligned}$$

Since ε was arbitrary, the proof is complete. Q.E.D.

Thus we have shown

THEOREM 3.8. $\varphi = \gamma$.

Next, we give an example of explicit calculation of φ as the limit of the φ -iterates.

Example 2. Let $I = \mathbb{N}$, $X = \{0\}$, $K = \{1, 2\}$ such that each process is a random walk on I (with X absorbing). It turns out that a fair interaction of two random walks, under the worst kind of schedules, yields essentially the same absorption probabilities as those yielded by the "worse" of the two walks alone. We exhibit here one simple case:

$$\begin{aligned} P_{i,i-1}^1 &= \frac{1}{3}, & P_{i,i+1}^1 &= \frac{2}{3}, & i &\geq 1, \\ P_{i,i-1}^2 &= 1, & i &\geq 1. \end{aligned}$$

It can be inductively shown that the φ -iterates for this program are

$$\gamma_{n-1}(i) = \begin{cases} \xi_{n-i} / \xi_n, & 0 \leq i \leq n, \\ 0, & i > n, \end{cases} \quad n \geq 1,$$

where $\xi_i = 2^i - 1$, $i \geq 0$, and

$$\gamma_\omega(i) = \gamma_{\omega+1}(i) = \varphi(i) = \frac{1}{2^i}, \quad i \geq 0.$$

Comparison with the iterates $\hat{\gamma}_n$ and their limit $\hat{\gamma}_\omega$ for the case in which only process 1 is activated shows that $\gamma_\omega = \hat{\gamma}_\omega$ but $\gamma_n > \hat{\gamma}_n$ for each finite n . Thus the fair interleaving of process 2 with process 1 increases the probability of convergence under any finite number of fairness constraints, but does not affect the overall (worst-case) convergence probability.

4. Characterizations of φ . This section contains the main results of the paper. Using the machinery developed in §§ 2 and 3, we will derive several characterizations of φ , which provide a variety of rather simple techniques for its calculation, or for deriving various properties of this function. Obviously, the most important such

property is whether $\varphi \equiv 1$ (i.e., whether the program terminates almost surely from any initial state). Relaxation of the characterizations of φ given here will enable us to derive necessary and sufficient conditions for program termination, and these conditions are presented in § 5.

THEOREM 4.1. (a) φ is the smallest fixpoint of the equation

$$\varphi = Q\varphi$$

which is $\cong \chi_X$.

(b) φ is the smallest simultaneous solution of the equations

$$\varphi = Q^k\varphi \text{ for each } k \in K,$$

which is $\cong \chi_X$.

Proof. By Propositions 3.5 and 2.1(a), $\varphi = Q\varphi = Q^k\varphi$ for each $k \in K$, and $\varphi \cong \chi_X$. To prove (a) we repeat the argument used in the proof of Lemma 3.6. That is, let $\psi \cong \chi_X$ be such that $\psi = Q\psi$. Then $\psi \cong \gamma_0$, thus $\psi = Q\psi \cong Q\gamma_0 = \gamma_1$, and by transfinite induction $\psi \cong \gamma_a$ for each ordinal a , thus $\psi \cong \gamma = \varphi$. As for (b), note that $\psi = Q^k\psi$ for all $k \in K$ implies $\psi = Q\psi$, and then use (a). Q.E.D.

Next we restate the second assertion of Theorem 4.1 in a manner which makes it more convenient for actual calculation of φ .

DEFINITION. Let α be a real function on I . We say that α has property (A) if the following are satisfied:

(A.1) $\alpha|_X \equiv 1$;

(A.2) α is subharmonic;

(A.3) for each $k \in K$ the only subharmonic function lying between α and $P^k\alpha$ is α itself.

(Note that the constant function 1 has property (A).)

THEOREM 4.2. φ is the smallest nonnegative function on I having property (A) (i.e., if $\alpha \cong 0$ satisfies (A), then $\alpha(i) \cong \varphi(i)$ for each $i \in I$).

Proof. By Lemmata 3.1 and 3.2, (A.2) and (A.3) imply $\alpha = Q^k\alpha$ for all $k \in K$, or $\alpha = Q\alpha$. We now use Theorem 4.1. Q.E.D.

Theorem 4.2 suggests the following procedure for computing φ : Take any nonnegative subharmonic function $\alpha \cong \chi_X$. For each $k \in K$ compute the largest subharmonic function which is $\cong P^k\alpha$, and require that it coincide with α . Find the general solution of these constraints, and obtain φ as the smallest such solution. Later on in this section we will use this procedure to compute φ for several exemplary programs, and show that this technique is quite feasible in practice.

Put

(A.2') α is min-harmonic,

and let property (A') be defined as the conjunction of (A.1), (A.2') and (A.3). Then we also have

COROLLARY 4.3. φ is the smallest nonnegative function having property (A').

Proof. Immediate, since φ itself is min-harmonic, by Proposition 2.1(b), and every min-harmonic function is also subharmonic. Q.E.D.

Remark. In carrying out the calculations of the procedure just outlined, it may sometimes be more convenient to employ the "1-complement" version of Theorem 4.2; that is, instead of computing φ we compute the function $\psi \equiv 1 - \varphi$, which is then the largest function ≤ 1 which is a fixpoint of the equation

$$\psi(i) = \min_{k \in K} \sup_{\tau \in T(i,k)} E_\tau(\psi)$$

or, alternatively, is the largest function $\beta \leq 1$ having *property (B)*, defined as

(B.1) $\beta|_X = 0$;

(B.2) β is *superharmonic*, i.e., $\beta \geq P^k \beta$ for each $k \in K$;

(B.3) for each $k \in K$, the only superharmonic function between $P^k \beta$ and β is β itself.

(Again, we can replace (B.2) by (B.2'), namely require that β be *max-harmonic*, that is $\beta = \max_{k \in K} P^k \beta$.)

The usefulness of this complementation lies in the fact that property (B) is positively homogeneous (i.e., β satisfies (B) implies $\lambda \beta$ satisfies (B) for every $\lambda > 0$, where $(\lambda \beta)(i) \equiv \lambda \cdot \beta(i)$); note that (A) was not such (due to (A.1)). For example, we obtain

COROLLARY 4.4. $\varphi \equiv 1$ if and only if no bounded function having some positive entries has property (B).

Proof. Assume β satisfies (B) so that $\lambda \equiv \sup_{i \in I} \beta(i) < \infty$ and is positive. Then $(1/\lambda)\beta$ also satisfies (B) and is ≤ 1 . Q.E.D.

We can also give now a second short proof of the Zero-One Law for φ ; namely, that $\inf_{i \in I} \varphi(i)$ is either 0 or 1 (see Theorem 2.3; however, the original proof is more elementary).

Second proof of the zero-one law (Theorem 2.3). Let $\psi = 1 - \varphi$ and put $\lambda = \sup_{i \in I} \psi(i)$. If $0 < \lambda < 1$, then the function $\psi' = (1/\lambda)\psi$ is larger than ψ , satisfies (B), and is ≤ 1 —contradicting the fact that ψ is the largest such function.

Examples. We will now apply the techniques presented in this section to several programs, to compute the function φ for each of these programs. These examples include two programs with finite state spaces (which had already been analyzed in a preceding paper [HSP] by different special techniques developed there for finite-state programs), and another program having an infinite state space.

Example 3. Let $K = \{1, 2\}$. The following program arises in an analysis of freedom from lockout in a simple synchronization protocol (cf. [HSP, Example 1] for details). Using a notation slightly different from that of [HSP], we have $I = X \cup \{i_1, i_2, i_3, i_4\}$, with nonzero transition probabilities

$$\begin{aligned} P^1_{i_1, X} &= P^2_{i_1, i_1} = 1, \\ P^1_{i_2, i_1} &= P^1_{i_2, i_4} = P^2_{i_2, i_1} = P^2_{i_2, i_4} = \frac{1}{2}, \\ P^1_{i_3, i_3} &= P^2_{i_3, i_2} = 1, \\ P^1_{i_4, i_4} &= P^2_{i_4, i_3} = 1. \end{aligned}$$

To compute φ , we first write down the form of the general subharmonic function which is 1 on X . Such a function $\alpha = (\alpha_1, \alpha_2, \alpha_3, \alpha_4)$ (where α_t is a shorthand for $\alpha(i_t)$, $1 \leq t \leq 4$) must satisfy

$$\begin{aligned} \alpha_1 &\leq 1, \\ \alpha_2 &\leq \frac{1}{2}\alpha_1 + \frac{1}{2}\alpha_4, \\ \alpha_3 &\leq \alpha_2, \\ \alpha_4 &\leq \alpha_3. \end{aligned}$$

Next, we spell out condition (A.3) for such an α : First consider $k = 1$. It is easily checked that the function

$$P^1 \alpha = (1, \frac{1}{2}\alpha_1 + \frac{1}{2}\alpha_4, \alpha_3, \alpha_4)$$

is also subharmonic. Hence we must have $\alpha = P^1\alpha$, i.e.,

$$\alpha_1 = 1, \quad \alpha_2 = \frac{1}{2}\alpha_1 + \frac{1}{2}\alpha_4 = \frac{1}{2} + \frac{1}{2}\alpha_4.$$

Similarly, for $k = 2$ we have

$$P^2\alpha = (\alpha_1, \frac{1}{2}\alpha_1 + \frac{1}{2}\alpha_4, \alpha_2, \alpha_3),$$

which is also seen to be subharmonic. Hence $\alpha = P^2\alpha$, i.e.,

$$\alpha_2 = \alpha_3, \quad \alpha_3 = \alpha_4.$$

Thus we have $\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = 1$. That is, the only—and thus, the smallest—function satisfying (A) is $\varphi \equiv 1$.

Example 4. This example is also taken from [HSP], and arises in the analysis of another synchronization protocol. Here $K = \{1, 2\}$, $I = X \cup \{i_1, \dots, i_6\}$, and the transition probability matrices are

| | P^1 | | | | | | | P^2 | | | | | | |
|-------|---------------|---------------|-------|---------------|---------------|---------------|---------------|---------------|-------|---------------|---------------|---------------|-------|-----|
| | i_1 | i_2 | i_3 | i_4 | i_5 | i_6 | X | i_1 | i_2 | i_3 | i_4 | i_5 | i_6 | X |
| i_1 | $\frac{1}{2}$ | | | | | | $\frac{1}{2}$ | $\frac{1}{2}$ | | | | $\frac{1}{2}$ | | |
| i_2 | | $\frac{1}{2}$ | | $\frac{1}{2}$ | | | | $\frac{1}{2}$ | | | | $\frac{1}{2}$ | | |
| i_3 | $\frac{1}{2}$ | | | | | | $\frac{1}{2}$ | | | $\frac{1}{2}$ | $\frac{1}{2}$ | | | |
| i_4 | | $\frac{1}{2}$ | | $\frac{1}{2}$ | | | | | | $\frac{1}{2}$ | $\frac{1}{2}$ | | | |
| i_5 | | | | | $\frac{1}{2}$ | $\frac{1}{2}$ | | 1 | | | | | | |
| i_6 | | | | | $\frac{1}{2}$ | $\frac{1}{2}$ | | | | | | | 1 | |

It is straightforward to check that a general subharmonic function $\alpha = (\alpha_1, \dots, \alpha_6)$ which is 1 on X must satisfy

$$\alpha_1 = \alpha_2 = \dots = \alpha_6 \leq 1.$$

It now follows that (A.3) holds for each such function α , because any function constant on $I - X$ and lying between α and $P^1\alpha$ (resp. $P^2\alpha$) must coincide with α (since $P^1\alpha$ (resp. $P^2\alpha$) coincides with α at some of these states). Thus φ , which is the smallest nonnegative such function, is χ_X .

Example 5 (“The Two Combs”). Let $K = \{1, 2\}$, $I = X \cup \mathbb{Z}$ (where \mathbb{Z} denotes the set of signed integers); the nonzero transition probabilities are

$$\begin{aligned} P^1_{n,n+1} &= p_n, & P^1_{n,X} &= p'_n = 1 - p_n, \\ P^2_{n,n-1} &= q_n, & P^2_{n,X} &= q'_n = 1 - q_n, \end{aligned} \quad n \in \mathbb{Z}.$$

To avoid degeneracy, we assume that $0 < p_n q_{n+1} < 1$ for each $n \in \mathbb{Z}$. Denote, for $n \in \mathbb{Z}$,

$$P_n = \prod_{m=n}^{\infty} p_m, \quad Q_n = \prod_{m=-\infty}^n q_m.$$

Denote by (C^+) the condition

$$(C^+) \quad \prod_{n>0} p_n > 0 \quad \text{and} \quad \limsup_{n \rightarrow \infty} q_n = 1,$$

and by (C^-) the condition

$$(C^-) \quad \prod_{n<0} q_n > 0 \quad \text{and} \quad \limsup_{n \rightarrow -\infty} p_n = 1.$$

PROPOSITION 4.5. (a) *If neither (C^+) nor (C^-) hold, then $\varphi \equiv 1$.*

(b) *If (C^+) holds but (C^-) does not hold, then $\varphi_n = 1 - P_n$, $n \in \mathbb{Z}$.*

(c) *If (C^-) holds but (C^+) does not hold, then $\varphi_n = 1 - Q_n$, $n \in \mathbb{Z}$.*

(d) *If both (C^+) and (C^-) hold, then $\varphi_n = 1 - \max\{P_n, Q_n\}$, $n \in \mathbb{Z}$.*

Proof. It will be more convenient to work in “1-complement” mode, calculating $\psi \equiv 1 - \varphi$, and using property (B). The calculation of ψ proceeds through the following steps (for details, see [HS]).

(1) If $\psi_n = 0$ for some $n \in \mathbb{Z}$, then $\psi \equiv 0$.

(2) Put $\psi^1 = P^1\psi$, $\psi^2 = P^2\psi$; if $\psi \equiv 0$ then it is impossible to have for some $n \in \mathbb{Z}$, $\psi_n^1 = \psi_n$ and $\psi_{n+1}^2 = \psi_{n+1}$.

(3) $\psi_n > \psi_n^1 \Rightarrow \psi_m > \psi_m^1$ for each $m \leq n$, and $\psi_n > \psi_n^2 \Rightarrow \psi_m > \psi_m^2$ for each $m \geq n$.

(4) Thus only the following four cases are possible:

(a) $\psi = \psi^1 = \psi^2 \equiv 0$;

(b) $\psi_n = \psi_n^1 > \psi_n^2$ for each $n \in \mathbb{Z}$;

(c) $\psi_n = \psi_n^2 > \psi_n^1$ for each $n \in \mathbb{Z}$;

(d) there exists $n_0 \in \mathbb{Z}$ such that $\psi_n = \psi_n^1 > \psi_n^2$ for each $n > n_0$, and $\psi_n = \psi_n^2 < \psi_n^1$ for each $n < n_0$.

(5) Suppose $\psi > 0$. If, for some $n_0 \in \mathbb{Z}$, $\psi_n = \psi_n^1$ for each $n > n_0$, then $\prod_{n > n_0} p_n > 0$. Similarly, if $\psi_n = \psi_n^2$ for each $n < n_0$, then $\prod_{n < n_0} q_n > 0$.

(6) In particular, if $\prod_{n > 0} p_n = \prod_{n < 0} q_n = 0$, then $\psi \equiv 0$.

(7) Suppose $\psi > 0$. If, for some $n_0 \in \mathbb{Z}$, $\psi_n = \psi_n^1$ for each $n > n_0$, then $\limsup_{n \rightarrow \infty} p_n q_{n+1} = 1$. Similarly, if $\psi_n = \psi_n^2$ for each $n < n_0$, then $\limsup_{n \rightarrow -\infty} p_n q_{n+1} = 1$.

(8) The following is a partial converse to (7): Let $\psi > 0$ be any max-harmonic function satisfying $\psi_n = \psi_n^1$ for each $n > n_0$, and suppose $\limsup_{n \rightarrow \infty} p_n q_{n+1} = 1$. Then the unique superharmonic function lying between ψ and ψ^2 is ψ itself. A similar statement holds if $\psi_n = \psi_n^2$ for each $n < n_0$ and $\limsup_{n \rightarrow -\infty} p_n q_{n+1} = 1$.

(9) In case (b) condition (C^+) holds; in case (c) condition (C^-) holds; and in case (d) both conditions (C^+) and (C^-) hold.

(10) Conversely, if (C^+) holds but (C^-) does not, then case (b) must occur. Similarly, if (C^-) holds but not (C^+) , then case (c) must occur.

(11) Finally, if both (C^+) and (C^-) hold, then case (d) must occur. Q.E.D.

5. Verification of program termination. The results developed in the two preceding sections provide us with methods for calculating the function φ for any concurrent probabilistic program. However, in many applications the only question of interest concerning φ is whether $\varphi \equiv 1$, i.e., whether the program terminates almost surely from any initial state under any fair schedule. In this section we will present several characterizations of program termination, the first two of which are straightforward specializations of the general results of the preceding sections, while the third involves a somewhat different approach, generalizing that used in [HSP] for finite state spaces.

PROPOSITION 5.1. *$\varphi \equiv 1$ if and only if no min-harmonic function smaller than 1 has property (A).*

Proof. See Corollary 4.3.

PROPOSITION 5.2. *$\varphi \equiv 1$ if and only if there exist an ordinal c and transfinite sequences of functions $\{\delta_a^k\}_{a \leq c}$, $k \in K$, and $\{\delta_a\}_{a \leq c}$ having the following properties:*

(1) $\delta_0 = \delta_0^k = \chi_x$, $k \in K$;

(2) δ_a^k is subharmonic for each $a \leq c$ and each $k \in K$;

(3) $\delta_a \leq \max_{k \in K} \delta_a^k$, $a \leq c$;

(4) $\delta_{a+1}^k \leq P^k \delta_a$, $k \in K$, $a < c$;

(5) $\delta_a^k \leq \sup_{b < a} \delta_b^k$, for limit ordinals a , and $k \in K$;

(6) $\inf_{i \in I} \delta_c(i) > 0$.

Proof. If $\varphi \equiv 1$ then the φ -iterates can be taken as the δ 's. Conversely, if such sequences of functions are given, then by transfinite induction $\delta_a \leq \varphi$ for each ordinal a . In particular $\delta_c \leq \varphi$, so that $\inf_{i \in I} \varphi(i) > 0$, and by the zero-one law (Theorem 2.3) we must have $\varphi \equiv 1$. Q.E.D.

Our next characterization of program termination generalizes one of the characterizations given in [HSP] for finite-state programs. Intuitively speaking, if the program does not always terminate, then there must exist some "ergodic structure" of nonterminating states, through which an "adversary" fair scheduler can iterate forever without reaching X . Unlike the case of a finite state space, where such a structure was a single "K-ergodic" set, ergodicity in general state spaces is a much more complex notion, and is defined as follows.

DEFINITION. A *K-ergodic chain* is a nonincreasing sequence $\{E_n\}_{n \geq 1}$ of nonempty subsets of $X^c \equiv I - X$ such that

$$\limsup_{\substack{n \rightarrow \infty \\ m \geq 1}} (Q\chi_{E_m^c})(i) = 0.$$

In other words, let $n \geq 1, i \in E_n, m \geq 1$ and $k \in K$ be given. Then there exists a subschedule in $T(i, k)$ which reaches E_m with probability tending to 1 uniformly as $n \rightarrow \infty$. That is, without losing too much probability, we can reach any of the sets E_m from any state in E_n after scheduling any required process.

THEOREM 5.3. $\varphi \equiv 1$ if and only if $I - X$ does not contain any *K-ergodic chain*.

Before proving this theorem, we need two lemmata.

LEMMA 5.4. Let $\delta > 0$, and define $D = \{i \in I : \varphi(i) \geq \delta\}$. Then $\varphi \geq Q\chi_D$.

Proof. Let $i \in I, k \in K$, and $\sigma \in \Sigma F(i)$. For each $n \geq 1$ define a stopping time N_n on $H^*(i)$ so that $N_n(\pi)$ is the n th time k has been scheduled along π ; note that $\{N_n\}_{n \geq 1}$ is an increasing sequence of μ_0 -a.s. finite stopping times, whose limit is $+\infty$. For each $n \geq 1$ the subschedule $\tau_n = (\sigma, N_n) \in T(i, k)$, so that

$$(Q^k\chi_D)(i) \leq E_{\tau_n}(\chi_D) = \mu_\sigma\{\varphi(i_{N_n}) \geq \delta\}.$$

Consider the sequence of functions $\{f_m\}_{m \geq 0}$, defined by $f_m(\pi) = \varphi(i_m), m \geq 0, \pi \in H^*(i)$. By Theorem 2.3 $\{f_m\}$ converges a.s. to a limit f_∞ , such that $f_\infty(\pi)$ is 1 if X is reached, and is otherwise 0. Therefore we also have $\varphi(i_{N_n}) \rightarrow f_\infty$ a.s. as $n \rightarrow \infty$, so that

$$\mu_\sigma\{f_\infty \geq \delta\} \geq \overline{\lim}_{n \rightarrow \infty} \mu_\sigma\{\varphi(i_{N_n}) \geq \delta\} \geq (Q^k\chi_D)(i).$$

Since $\delta > 0$, we have $f_\infty(\pi) \geq \delta$ if and only if $f_\infty(\pi) = 1$, or, alternatively, if and only if X is reached along π . Thus

$$E_\sigma(\chi_X) = \mu_\sigma\{f_\infty \geq \delta\} \geq (Q^k\chi_D)(i),$$

from which our assertion follows. Q.E.D.

LEMMA 5.5. Let $\{G_n\}_{n \geq 1}$ be a nondecreasing sequence of subsets of I , all of which contain X , and let $\{\varepsilon_n\}_{n \geq 1}$ be a sequence of positive numbers converging to 0. Suppose that

$$Q\chi_{G_m}(i) \leq \varepsilon_n$$

for each $m, n \geq 1$ and each $i \in G_n^c$. Then

$$\varphi \leq \sup_m Q\chi_{G_m}.$$

Proof. Put $\beta \equiv \sup_m Q\chi_{G_m}$. The above assumption concerning $\{G_n\}$ can be restated as

$$Q\chi_{G_m} \leq \varepsilon_n \cdot \chi_{G_n^c} + 1 \cdot \chi_{G_n} = \varepsilon_n + (1 - \varepsilon_n)\chi_{G_n},$$

for each $m, n \geq 1$. This implies

$$\beta \leq \varepsilon_n + (1 - \varepsilon_n)\chi_{G_n}, \quad n \geq 1,$$

and thus

$$Q\beta \leq Q(\varepsilon_n + (1 - \varepsilon_n)\chi_{G_n}), \quad n \geq 1.$$

However, it is easily checked that for any scalars $a, b > 0$ and any nonnegative function α we have

$$Q(a + b\alpha) = a + bQ\alpha.$$

Hence,

$$Q\beta \leq \varepsilon_n + (1 - \varepsilon_n)Q\chi_{G_n} \leq \varepsilon_n + (1 - \varepsilon_n)\beta,$$

for each $n \geq 1$. Letting $n \rightarrow \infty$, we obtain $Q\beta \leq \beta$. But β is subharmonic (as a supremum of subharmonic functions), thus $\beta = Q\beta$ (see Lemmata 3.1 and 3.2), implying $\varphi \leq \beta$ by Theorem 4.1. Q.E.D.

Proof of Theorem 5.3. The theorem is now an easy consequence of the last two lemmata. For example, if φ is not identically 1, then, by Lemma 5.4, the collection $\{E_n\}_{n \geq 1}$ is a K -ergodic chain, where

$$E_n = \left\{ i \in I : \varphi(i) < \frac{1}{n} \right\}.$$

The converse statement follows similarly from Lemma 5.5 (for more details, see [HS]). Q.E.D.

Example 5 revisited. Consider the three cases in the example of “the two combs” in which $\varphi < 1$. It is easily verified that in case (b) the chain $E_n^+ = \{i : i \geq n\}$, $n \geq 1$, is K -ergodic; similarly, in case (c) the chain $E_n^- = \{i : i \leq -n\}$, $n \geq 1$, is K -ergodic; and in case (d) both these chains are K -ergodic.

Remark. In the case of a finite state space, ergodicity is manifested in a single set (see [HSP]). In an analogous manner, we could have considered here the set $E = \{i \in I : \varphi(i) = 0\}$ as a natural candidate for being K -ergodic (that is, consider the constant chain $E_n = E$, $n \geq 1$). There are two problems, however, with this approach, which make it infeasible for general state spaces. One problem is that E may be empty (as is indeed the case in Example 5 just considered). Moreover, even if E is not empty, it may happen that, starting from some $i \in E$, we never reach E again, but instead reach states j at which $\varphi(j)$ is arbitrarily small, but positive. Thus, for general state spaces ergodicity must be defined in terms of an infinite chain of sets rather than in terms of a single set. (In the finite case, though, any K -ergodic chain must reduce to a constant set from a certain index on.) Note that this phenomenon occurs in Markov chains as well.

We conclude this section with a further property of φ .

PROPOSITION 5.6. *There exists a nondecreasing sequence $\{D_n\}_{n=1}^\infty$ of subsets of I such that*

- (1) $\varphi|_{D^c} \equiv 0$, where $D = \bigcup_{n=1}^\infty D_n$,
- (2) $\varphi = \lim_{n \rightarrow \infty} Q\chi_{D_n}$.

Proof. Put $D_n = \{i : \varphi(i) \geq 1/n\}$. Q.E.D.

Note that in the case $\varphi \equiv 1$ we can take $D_n \equiv I$ for all n . Moreover,

COROLLARY 5.7. *If I is a finite set, then there exists $D \subset I$ such that*

- (1) $\varphi|_{D^c} \equiv 0$,
- (2) $\varphi = Q\chi_D$.

6. Comparison with the deterministic and the nondeterministic cases. In this section we consider the special case in which each process is deterministic; programs with nondeterministic processes are also included, since any such program can be simulated by a deterministic one involving additional processes. (For example, suppose that one of the processes k_1 makes a nondeterministic choice from some set A of alternatives; the same behavior can be achieved by introducing a new shared variable ν which k_1 sets to some value in A prior to making the choice, and by introducing another process k_2 whose only action is to iterate ν over the set A . k_1 then makes its choice deterministically, depending on the current value of ν . Thus the nondeterminism is now transferred to the scheduler—the final choice depends on how many times k_2 has been scheduled in between.) Thus, by specializing the various equivalent criteria for program termination developed so far in this paper to the deterministic case, we can obtain similar criteria for the termination of deterministic (or nondeterministic) concurrent programs. As it turns out, the criterion obtained in this way from the characterization of φ as the limit of the φ -iterates (Theorem 3.8, Proposition 5.2) essentially coincides with the known criterion of Lehmann, Pnueli and Stavi [LPS]. On the other hand, specialization of Theorem 4.2 leads to a new characterization for deterministic and nondeterministic termination. So as not to make this characterization appear too deep, we provide a direct nonprobabilistic proof of its validity.

We begin by observing that in the deterministic case all transitions have probability 0 or 1, so that each of the operators Q^k , $k \in K$, and Q , when applied to a function which takes only the values 0, 1 (i.e., a characteristic function of some subset of I) yields a similar function. Hence each of the φ -iterates γ_a^k , $k \in K$, (resp. γ_a) is a characteristic function of the form $\chi_{G_a^k}$ (resp. χ_{G_a}). Note also that a characteristic function χ_A is subharmonic if and only if for each $k \in K$ and each $i \in A$ the (unique) k -transition from i is to a state in A , i.e., there are no transitions from states in A to states outside A . Hence, spelling out the conditions in Proposition 5.2 in terms of the subsets of I corresponding to the functions appearing there, we obtain the following.

COROLLARY 6.1. *A deterministic program terminates if and only if there exist transfinite (increasing) sequences $\{G_a^k\}_{a \geq 0}$, $k \in K$, and $\{G_a\}_{a \geq 0}$ of subsets of I having the following properties:*

- (1) $G_0 = G_0^k = X$, $k \in K$;
- (2) *there are no transitions from states in G_a^k to states outside G_a^k , for each ordinal a and each $k \in K$;*
- (3) $G_a = \bigcup_{k \in K} G_a^k$, for each ordinal a ;
- (4) *for each $k \in K$ and each ordinal a , all k -transitions from states in G_{a+1}^k are to states in G_a ;*
- (5) $G_a^k = \bigcup_{b < a} G_b^k$, for each limit ordinal and each $k \in K$;
- (6) *there exists an ordinal c such that $G_c = I$.*

These conditions, however, are merely a rephrasing of the characterization for termination of “just” programs given by Lehmann, Pnueli and Stavi in [LPS]. To see this, define a function ρ from I to the ordinals by

$$\rho(i) = \min \{a : i \in G_a\}, \quad i \in I,$$

and a function $h : I \rightarrow K$ which maps each $i \in G_{\rho(i)}$ to some $k \in K$ such that $i \in G_{\rho(i)}^k$. Then it is easily checked that these functions satisfy the conditions in [LPS] for just termination, i.e.: the “ranking” map ρ never increases during execution; activating process $h(i)$ at state i always strictly decreases the value of ρ ; and h remains unchanged

as long as ρ does not decrease. By the remark made in the beginning of this section, it is easily seen that Corollary 6.1 also applies to nondeterministic (nonprobabilistic) programs. Thus our Proposition 5.2 generalizes the results of [LPS] to the probabilistic case.

Next we specialize Theorem 4.2 to the deterministic case. As noted above, we can identify subharmonic functions with subsets A of I such that there are no transitions from A to A^c . This notion is formalized in the following.

DEFINITION. (a) A *cut* (I_0, I_1) is a partition of I into two disjoint subsets I_0, I_1 with $I_0 \cup I_1 = I$, such that $X \subset I_1$ and such that there are no transitions from I_1 to I_0 .

(b) For each cut (I_0, I_1) and each $k \in K$ put

$$I_0^k = \{i \in I_0: \text{the } k\text{-transition from } i \text{ is into } I_0\}.$$

(Note that χ_{I_1} is subharmonic and $\cong \chi_X$, and that $P^k \chi_{I_0} = \chi_{I_0^k}$.) Using these notations, Theorem 4.2 translates into the following theorem (which merely states that (A.3) does not hold for any subharmonic function < 1).

THEOREM 6.2. *A deterministic program terminates if and only if for each nontrivial cut (I_0, I_1) (i.e., $I_0 \neq \emptyset$) there exists $k \in K$ and another cut (J_0, J_1) such that*

$$I_0^k \subseteq J_0 \subsetneq I_0.$$

Proof. As this result (and its appropriate generalization to the nondeterministic case) is new, and may be of interest in its own right, we provide here a direct proof of this characterization, which does not use the probabilistic techniques developed in this paper.

Assume first that the condition of the theorem does not hold, i.e., that there exists a nontrivial cut (I_0, I_1) such that for each $k \in K$ and each set $I_0^k \subseteq J \subsetneq I_0$, the pair (J, J^c) is not a cut, that is, there exist transitions from J^c to J (these transitions can only be from states in $I_0 - J$). Let $i \in I_0$, and let $F(i)$ denote the set of all states in I_0 (including i) reachable from i by some finite sequence of process activations. We claim that for each $k \in K$, $F(i)$ intersects I_0^k . For otherwise, put $J = I_0 - F(i)$, so that $I_0^k \subset J \subsetneq I_0$. By our assumption there exist transitions from $I_0 - J = F(i)$ into J , which contradicts the definition of $F(i)$. This implies that I_0 is ergodic, i.e., that there exists a fair schedule σ which can keep the program in I_0 forever. To prove this it suffices to show that for each $i \in I_0$ and each $k \in K$ there exists a finite scheduling sequence starting at i and ending by scheduling k and reaching a state in I_0 . Since $F(i) \cap I_0^k \neq \emptyset$, take a finite sequence of process activations which takes the program from i into some state in $F(i) \cap I_0^k$, and then schedule k , thereby reaching a state in I_0 .

Next suppose that the condition of the theorem does hold. We will construct a "ranking" function ρ from I to the ordinals and an "assistance" function $h: I \rightarrow K$ which will satisfy the conditions of [LPS] for program termination. These functions will be constructed in the following transfinite inductive manner. Put initially $\rho|_X = 0$, and define $h|_X$ in an arbitrary manner. Suppose inductively that ρ and h have already been defined on some subset M of I such that (L, M) is a cut (where $L = M^c$). By the above condition, there exists $k \in K$ and another cut (J, J^c) such that $L^k \subseteq J \subsetneq L$. Put $H = L - J \neq \emptyset$, and define $\rho|_H = 1 + \sup \rho|_M$, and $h|_H = k$. Note that since (J, J^c) is a cut, there are no transitions from H into J , and since H is disjoint from L^k , each k -transition from H is into M . Repeating this construction transfinitely, we obtain everywhere-defined functions ρ and h whose properties imply by [LPS] that the program terminates. Q.E.D.

Remark. To obtain the nondeterministic version of Theorem 7.2, define the sets I_0^k by

$$I_0^k = \{i \in I_0: \text{there exists a } k\text{-transition from } i \text{ into } I_0\},$$

and leave all other definitions and assertions unchanged.

Example 1 revisited. Let us apply Theorem 6.2 to the deterministic program given in Example 1, § 3. As is easily checked, a cut (I_0, I_1) of I must be one of the following three types:

- (a) $I_0 = [0, n] \times \{2\}$, for some $n \in \mathbb{N}$;
- (b) $I_0 = \mathbb{N} \times \{2\}$;
- (c) $I_0 = (\mathbb{N} \times \{2\}) \cup ([n, \infty] \times \{1\})$ for some $n \geq 1$.

In cases (a) and (b), $I_0^2 = \emptyset$, so that $(I_0^2, (I_0^2)^c) = (\emptyset, I)$ is a cut satisfying the condition of Theorem 6.2. In case (c), $I_0^1 = (\mathbb{N} \times \{2\}) \cup ([n+1, \infty] \times \{1\})$, so that $(I_0^1, (I_0^1)^c)$ is the required cut. Thus program termination is ensured by Theorem 6.2.

7. Programs with finite state space. In a preceding paper [HSP], the special case of concurrent probabilistic programs with finite state spaces has been analyzed, and a characterization of almost sure program termination in terms of the existence of a certain decomposition of the state space has been obtained. In this section we show how to obtain this characterization from the general theory developed so far in this paper.

Let us now assume that I is finite, and that $\varphi \equiv 1$. We will obtain a decomposition of I into (finitely many) disjoint sets $\{I_m\}_{m \geq 0}$ such that the following properties hold (here we use the notation $P_{i,E}^k \equiv \sum_{j \in E} P_{ij}^k$):

- (a) $I_0 = X$;
- (b) for each $m \geq 1$, each $i \in I_m$ and each $k \in K$, if $P_{i,J_m}^k = 0$ then $P_{i,I_m}^k = 1$ (where $J_m = \bigcup_{s < m} I_s$);
- (c) for each $m \geq 1$ there exists $k = k(m) \in K$ such that for each $i \in I_m$, $P_{i,J_m}^k > 0$. (This is the decomposition obtained in [HSP].)

To obtain it, we proceed inductively. Initially put $I_0 = X$. Suppose I_0, \dots, I_{m-1} have already been constructed. Let $J_m = \bigcup_{s < m} I_s$ and $H_m = J_m^c$. If $H_m = \emptyset$, the decomposition is complete. Otherwise, $\chi_{J_m} \neq 1 = \varphi$, thus we cannot have $\varphi \leq \chi_{J_m}$, therefore $Q\chi_{J_m} \not\leq \chi_{J_m}$ (indeed, for any $\alpha \geq \chi_X$, if $Q\alpha \leq \alpha$ then $\varphi \leq Q\alpha \leq \alpha$). Thus there exists $k \in K$ such that $\delta \equiv Q_{J_m}^k \not\leq \chi_{J_m}$. Define

$$c = \max_{j \in H_m} \delta(j) \quad \text{and} \quad I_m = \{i \in H_m: \delta(i) = c\}.$$

Note that $c > 0$, for otherwise $\delta|_{H_m} \equiv 0$, so that $\delta \leq \chi_{J_m}$, which is impossible.

It can be now seen that conditions (b) and (c) hold for I_m .

Remarks. (1) The converse statement, namely that the existence of such a decomposition implies that $\varphi \equiv 1$, is also easy to establish, e.g., by proving that $\min_{i \in I} \varphi(i) > 0$, and using the zero-one law (cf. [HSP] for a detailed proof).

(2) Once the existence of such a decomposition has been established, it can also be obtained in the following different manner (for details, see [HS]).

Define an equivalence relation on I so that $i, j \in I$ are equivalent if and only if $\alpha(i) = \alpha(j)$ for every subharmonic function α . The sets I_m , $m \geq 0$, are then simply the equivalence classes of this relation. Furthermore, to assign indices to these sets, look for a subharmonic function α which assumes distinct values on each of these sets, and order the equivalence classes in decreasing order of the values of α on them. Such a separating subharmonic function always exists; moreover the decomposition will satisfy condition (c) if and only if $\varphi \equiv 1$. (Thus, in the finite case there always exists a

decomposition of I into an ordered sequence of sets $I_m, m \geq 0$, which satisfy conditions (a) and (b); each of these sets either satisfies conditions (c) or else is K -ergodic.)

An open problem is whether the existence of a similar decomposition is equivalent to $\varphi \equiv 1$ in the general (discrete) case as well.

8. Conclusions. In this paper we have analyzed termination of concurrent probabilistic programs having discrete infinite state spaces. Our aim has been to calculate the worst-case probability of such a program to reach a given set of terminating states under an arbitrary but fair scheduling of its processes. We have obtained several characterizations of the required probability function φ , which yielded useful techniques for the calculation of this function. Specializing to the case of deterministic (or nondeterministic) programs, our techniques have been shown to generalize known techniques for proving termination of such programs, and also to yield new such techniques. From the point of view of the theory of probability, our results extend the classical theory of optimal gambling strategies by Dubins and Savage [DS] to the case where such strategies must be "fair."

The model that we have introduced in this paper and in the preceding one [HSP] for the (fair) execution of concurrent probabilistic programs is very general, natural, and easy to work with, and we believe that it should serve as a standard model for execution of such programs. A more detailed discussion concerning this model can be found in [HSP].

The techniques developed in the present paper can be immediately interpreted as *sound* and *complete* proof methods for termination of concurrent probabilistic programs. It would be interesting to generalize these techniques to proof methods for additional properties of such programs, or, alternatively, to develop temporal probabilistic logics, based upon our techniques, for reasoning about such programs (see, e.g., [HS2]). One such generalization can be achieved as follows: Let α be a subharmonic function defined on I . For each schedule σ define $E_\sigma(\alpha)$ as $\lim_{n \rightarrow \infty} E_{(\sigma,n)}(\alpha)$ (which always exists, by the subharmonicity of α). Then we want to compute

$$\varphi_\alpha(i) = \inf_{\alpha \in \Sigma_{F(i)}} E_\sigma(\alpha),$$

which generalizes the function $\varphi \equiv \varphi_{X_X}$ studied in this paper. Intuitively, $\varphi_\alpha(i)$ is the smallest "long-term" expected value of α under a fair schedule starting at i . Most of the theory developed in §§3 and 4 can be generalized to the case of a general subharmonic α .

(An interesting choice for α is where $\alpha_{|I-X} \equiv 0, \alpha_{|X} \geq 0$; then φ_α gives the smallest expected value of α upon termination under fair execution. Thus, appropriate adaptation of the techniques developed in this paper will enable us to derive lower bounds for the expected value of such functions upon termination; compare with [SPH]).

A final corollary of the results developed in this paper concerns *bounded waiting time* (cf. [Ra1] for example). Let us define a *round* of execution as a portion of the execution during which each process has been scheduled at least once. We then say that the program has the (*local*) *bounded waiting time property* at some $i \in I$ if for each $\epsilon > 0$ there exists an integer $N = N(i, \epsilon)$ such that the probability of reaching X from i after N rounds under any fair schedule is at least $1 - \epsilon$. The program has the *global bounded waiting time property* if the above holds for all $i \in I$ and if N is independent of i . A simple application of Theorem 3.8 implies the next corollary.

COROLLARY 8.1. *The program has the local bounded waiting time property at $i \in I$ if and only if $\gamma_\omega(i) = 1$. It has the global property if and only if $\gamma_\omega \equiv 1$ (i.e., if the convergence ordinal of the program is $\leq \omega$) and $\gamma_n \uparrow 1$ uniformly on I as $n \uparrow \omega$.*

REFERENCES

- [Ch] K. L. CHUNG, *Markov Chains with Stationary Transition Probabilities*, Springer-Verlag, New York, 1967.
- [DS] L. E. DUBINS AND L. J. SAVAGE, *Inequalities for Stochastic Processes; How to Gamble if You Must*, Dover, New York, 1976.
- [HSP] S. HART, M. SHARIR AND A. PNUELI, *Termination of concurrent probabilistic programs*, ACM Trans. Prog. Languages and Systems, 5 (1983), pp. 356–380.
- [HS] S. HART AND M. SHARIR, *Concurrent probabilistic programs, or how to schedule if you must*, Tech. Rept., School of Mathematical Sciences, Tel Aviv University, May 1982.
- [HS2] ———, *Probabilistic propositional temporal logics*, Proc. 16th Symposium on Theory of Computing, April 1984, pp. 1–13.
- [LPS] D. LEHMANN, A. PNUELI AND J. STAVI, *Impartiality, justice, fairness: the ethics of concurrent termination*, Proc. 8th ICALP Conference, 1981, pp. 264–277.
- [LR] D. LEHMANN AND M. O. RABIN, *On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers' problem*, Proc. 8th Symposium on Principles of Programming Languages, 1981, pp. 133–138.
- [Ra1] M. O. RABIN, *N process synchronization by a $4 \log_2 N$ -valued shared variable*, J. Comp. Syst. Sciences, 25 (1982), pp. 66–75.
- [Ra2] ———, *The choice coordination problem*, Acta Inform., 17 (1982), pp. 121–134.
- [RS1] J. REIF AND P. SPIRAKIS, *Distributed algorithms for synchronizing interprocess communication within real time*, Proc. 13th Symposium on Theory of Computing, 1981, pp. 133–145.
- [RS2] ———, *Unbounded speed variability in distributed communication systems*, Proc. 9th Symposium on Principles of Programming Languages, 1982, pp. 46–56.
- [SPH] M. SHARIR, A. PNUELI AND S. HART, *Verification of probabilistic programs*, this Journal, 13 (1984), pp. 292–314.

SEARCHING UNINDEXED AND NONUNIFORMLY GENERATED FILES IN $\log \log N$ TIME*

DAN E. WILLARD†

Abstract. The first algorithm that searches unindexed and nonuniformly distributed ordered files in $\log \log N$ expected time is presented in this paper. Our analysis rests on a synthesis of concepts from the literature on interpolation search and on the method of regula falsi in numerical analysis.

Key words. interpolation search, binary search, method of regula falsi, method of false position, secant method, padded list, dense sequential file, fast trie, priority queue, stratified tree, Van Emde Boas tree

1. Goals and main results. Searching an ordered file is a very common operation in data processing and so should be accomplished as rapidly as possible. Given N records, stored (ideally) in successive memory-locations in the order of their numeric keys $Y_1 < Y_2 < \dots < Y_N$, one often wishes to find a particular record whose key equals y . Several recent papers [GRG-80], [PR-77], [PIA-78], [YY-76] have shown how an algorithm called *interpolation search* and its close cousins perform this operation in expected runtime $\log \log N$ when the keys in the ordered file are generated by the uniform probability distribution.¹ In this paper, we study how retrieval can be performed efficiently for nonuniform probability distributions.

Two new results are presented in this paper. The first is that the $\log \log N$ asymptotic retrieval time of interpolation search does not remain in force for most nonuniform probability distributions. Our second result is more surprising, and it is that this $\log \log N$ expected runtime can be re-established for nearly all nonuniform probability distributions by using a modified version of interpolation search.

More specifically, let μ denote a probability density function over the real line. We will say that an ordered file of cardinality N has been *generated by μ* iff this file was constructed by taking N records whose keys are independently determined by μ and storing them in ascending order. The probability density μ will be said to be *regular* iff there exists some vector (b_1, b_2, b_3, b_4) such that μ and its first derivative satisfy the following conditions:

$$(1.1) \quad \mu(y) \geq b_1 > 0 \quad \text{whenever } b_3 < y < b_4,$$

$$(1.2) \quad |\mu'(y)| \leq b_2 \quad \text{whenever } b_3 < y < b_4,$$

$$(1.3) \quad \mu(y) = 0 \quad \text{whenever } y \leq b_3 \text{ or } y \geq b_4.$$

The symbol B will denote this vector, and we will say that μ is bounded by B when the equations above hold. Their meaning is that μ is nonzero on only a bounded interval and that it has a lower bound and bounded derivative on this interval. The main algorithm of this paper will attain $\log \log N$ expected runtime on any regular probability density in the sense that there will exist one constant K (whose value is independent of B) and a second constant K_B (whose value depends on this bounding vector) such that the equation below bounds the expected time to search a file whose

* Received by the editors June 3, 1981 and in revised form May 5, 1984. This work was partially supported by National Science Foundation grant DCR-8412447.

† State University of New York, Albany, New York 12222 and Consultant, Bell Communications Research.

¹ Throughout this paper "log" designates base 2 logarithm.

density is bounded by B :

$$(1.4) \quad K \log \log N + K_B.$$

Equation (1.4) is the strongest type of asymptote one could hope for, since it states that the bounding vector of μ only contributes an additive constant to the total expected time. The main unsolved open question is to make precise estimates for the values of the constants K and K_B . The lower bound of Yao and Yao [YY-76] implies that no search algorithm can have a constant $K < 1$.

Our main theorem should not be confused with remarks made in earlier papers [GRG-80], [PIA-78] to the effect that results on uniform distributions extend readily to nonuniform distributions if the distribution function $D(y) = \int_{-\infty}^y \mu(x) dx$ is employed to map an initial nonuniform distribution onto a uniform distribution. The disadvantage of this method is that it relies on detailed information about μ (or D) that is typically unavailable or expensive. This paper considers the more difficult problem where such information is inaccessible and answers an open question from [YY-76] by showing that asymptotic time $\log \log N$ is possible without it.

The discussion in this paper will be divided into four parts. Sections 2 and 3 will define our new algorithm and review the previous literature. The main theorem proofs appear in § 4, and § 5 discusses extensions and open questions.

Two earlier versions of our result appeared in the conference papers [Wi-81], [Wi-83b], and these papers played a role in stimulating some subsequent interesting research by Mehlhorn and Tsakalidis [MT-84].

2. Description of algorithms. Given an ordered file F of keys $Y_1 < Y_2 < \dots < Y_N$, the algorithms considered in this paper will be iterative procedures whose i th iteration searches the segment $F_i = (Y_{L_i} < \dots < Y_{R_i})$ by generating a cut index C_i and comparing the cut value Y_{C_i} with the search key y . If $Y_{C_i} = y$ then the search terminates successfully; otherwise, the next iteration will examine either the subfile $(Y_{L_i} < \dots < Y_{C_i})$ or $(Y_{C_i} < \dots < Y_{R_i})$ according to whether or not $Y_{C_i} > y$.

The term *iterative reduction* will refer to such algorithms. The only difference between the various algorithms within this class of procedures is their specific rule for computing the cut index.

The simplest iterative reduction algorithm is binary search, whose cut index is the middle position in the subfile $F_i = (Y_{L_i} < \dots < Y_{R_i})$. This index can therefore be defined as:

$$(2.1) \quad C_i^{\text{BIN}} = \left\lceil \frac{(L_i + R_i)}{2} \right\rceil.$$

Under interpolation search [GRG-80], [PIA-78], [YY-76], the cut index approximates the expected position of y when the $R_i - L_i - 1$ untested interior keys of the subfile $F_i = (Y_{L_i} < \dots < Y_{R_i})$ are generated by the uniform distribution. In order to define this concept formally, let N_i , l_i , and C_i^{INT} denote the following quantities:

$$(2.2) \quad N_i = R_i - L_i,$$

$$(2.3) \quad l_i = Y_{R_i} - Y_{L_i},$$

$$(2.4) \quad C_i^{\text{INT}} = L_i + \frac{y - Y_{L_i}}{l_i} N_i.$$

In this notation, the cut index of interpolation search is defined to be C_i^{INT} , rounded to the nearest integer.

The new algorithm proposed in this section will be called RETRIEVE (α, θ, ϕ) . Throughout our discussion of this algorithm, the unsubscripted small letter y will designate the target key that RETRIEVE is trying to find. RETRIEVE is defined as follows: Let Δ_i , C_i^+ , and C_i^- denote the following three quantities, where the parameters α , θ , and ϕ , satisfying $0 < \alpha \leq 1$ and $\theta, \phi > 0$, are parameters used to fine-tune the runtime coefficient:

$$(2.5) \quad \Delta_i = \theta N_i l_i^\alpha + \phi \sqrt{N_i},$$

$$(2.6) \quad C_i^+ = \lceil C_i^{\text{INT}} + \Delta_i \rceil,$$

$$(2.7) \quad C_i^- = \lfloor C_i^{\text{INT}} - \Delta_i \rfloor.$$

During its i th iteration, RETRIEVE will set the cut index C_i equal to:

- I) the smaller of C_i^+ and $R_i - 1$ when $i = 1 \pmod{3}$,
- II) the larger of C_{i-1}^- and $L_i + 1$ when $i = 2 \pmod{3}$,
- III) C_i^{BIN} when $i = 3 \pmod{3}$.

It will then use this cut index to reduce the search space in a manner similar to that of all other iterative reduction algorithms.

A more formal algorithmic description of the procedure RETRIEVE can be found in Appendix A. Our formalism can prove RETRIEVE has an expected time $\log \log N$ (in the sense of (1.4)) whenever $0 < \alpha < 1$, $\theta > 0$ and $\phi \geq 0$, but it is preferable to focus on the case where $\alpha = \frac{1}{2}$ and $\theta = \phi = 1$ to simplify the presentation. The term SIMPLE.RETRIEVE will refer to the variation of RETRIEVE that uses these three values for α , θ and ϕ . The resulting values for Δ_i , C_i^+ and C_i^- in this algorithm are

$$(2.8) \quad \Delta_i = N_i \sqrt{l_i} + \sqrt{N_i},$$

$$(2.9) \quad C_i^+ = \lceil C_i^{\text{INT}} + \Delta_i \rceil = \lceil C_i^{\text{INT}} + N_i \sqrt{l_i} + \sqrt{N_i} \rceil,$$

$$(2.10) \quad C_i^- = \lfloor C_i^{\text{INT}} - \Delta_i \rfloor = \lfloor C_i^{\text{INT}} - N_i \sqrt{l_i} - \sqrt{N_i} \rfloor.$$

The nice aspect of SIMPLE.RETRIEVE is that it has a short proof that its search time is characterized by (1.4). However, the algorithm does not produce the optimal constants K and K_B in this equation, and the best algorithm for controlling the coefficients remains an open question. This paper seeks simplicity rather than to attain the best coefficients because simplicity seems more appropriate for an article introducing these results.

In § 5, we will outline several techniques that can improve the coefficients but are not discussed in detail because they complicate the presentation. For instance, the subscript $i-1$ in cutting rule II will simplify our main proof, but it is less efficient than a subscript of i .

3. Background literature and intuition. This section will begin by surveying the literature on searching, and it will then explain the intuition behind our algorithm by showing how it can be seen as a natural synthesis of the computer science literature on interpolation search and the numerical literature on regula falsi. Searching an ordered file is of course one of the oldest subjects in computer science. B -trees are a very practical data structure in applications seeking to minimize disc accesses. They provide a number of page accesses less than $\log_2 N$, but they are still asymptotically logarithmic. Since the middle 1970's, a growing number of mathematical computer scientists have started to investigate whether better asymptotes were possible with more refined techniques.

The literature can be divided into two branches studying quite separate mathematical questions. [AFK-83], [FKS-82], [Jo-81], [TY-79], [Va-75], [Va-77], [VKZ-77], [Wi-83a], [Wi-84a], [Ya-82] consider how to obtain better than logarithmic time when searching a set of N positive integers bounded by M . These articles illustrate a diversity of different upper and lower bounds on worst-case time depending on the amount of memory space consumed, the specific types of queries allowed, the question of whether N and M are allowed to obtain any possible configuration of values, and the question of whether or not the data structure supports insertions and deletions. All the data structures cited in this paragraph use indices, and they do not address the specific mathematical question about the types of search times which would be possible if no index were present and the keys were arbitrary real numbers, chosen from a universe of infinite size, simply stored in sequential order.

The answer to the latter mathematical question appears in the literature about interpolation search and its variants. [GRG-80], [PIA-78], [YY-76] show that interpolation search has a log-logarithmic time when it searches the uniform distribution, and [PR-77] describes a variant that has an especially short proof. The strongest version of the interpolation result appears in Yao and Yao [YY-76]—where the interpolation algorithm is shown to have a complexity $\log \log N + O(1)$ —which is proven to be optimal up to an additive constant. [GRG-80] offers an alternate proof of the upper bound. [PIA-78], [PR-77] establish short proofs of weaker results where the multiplicative constant in front of the $\log \log N$ is $\neq 1$, whose advantage is that they are very useful for classroom presentation.

All the articles introducing interpolation search [GRG-80], [PIA-78], [PR-77], [YY-76] have confined their discussion to a static environment, and we make a similar simplifying assumption in this paper. However, both our results and those of [GRG-80], [PIA-78], [PR-77], [YY-76] do generalize to a dynamic environment. The most recent article on this subject is [MT-84]. It draws upon the earlier conference versions of Willard's papers on nonuniform densities [Wi-81], [Wi-83b] and generalizes these results to a dynamic environment. Other articles on dynamic manipulations of sequential files include [Fr-79], [HKW-85], [IKR-80], [MG-80], [Wi-82]. These articles have less robust models for expected time than [MT-84] and focus on quite different issues related to numbers of page shift operations in mostly worst-case and amortized cost models.

Two other articles that may interest the reader are [RW-84], [De-79]. Reif and Willard [RW-84] examine parallel implementations of interpolation search and prove surprisingly that $p \geq 2$ parallel processors have essentially no value except in the very last iteration of this algorithm. Devroye [De-79] examines sorting problems on nonuniformly generated files. His work should interest some readers because the sorting and searching are so different in this context.

The thesis of our research project in this paper is that numerical analysis is very relevant to interpolation search because it will enable the results of [GRG-80], [PIA-77], [PR-77], [YY-76] to generalize to any nonuniform regular probability distribution. The remainder of this section will prove that interpolation search must be inefficient on nonuniform distributions and explain the intuition behind the added efficiency of our new algorithm by citing the background literature from numerical analysis.

The analogue of interpolation search in numerical analysis is called the method of regula falsi. Given a function g and a value y , this method consists of an iterative procedure that conducts repeated interpolations to find an approximate solution for $g(x) = y$. The general procedure of this algorithm is illustrated in Fig. 1. It begins with two points, x_0 and x_1 , which bracket the solution point x , and applies the interpolation

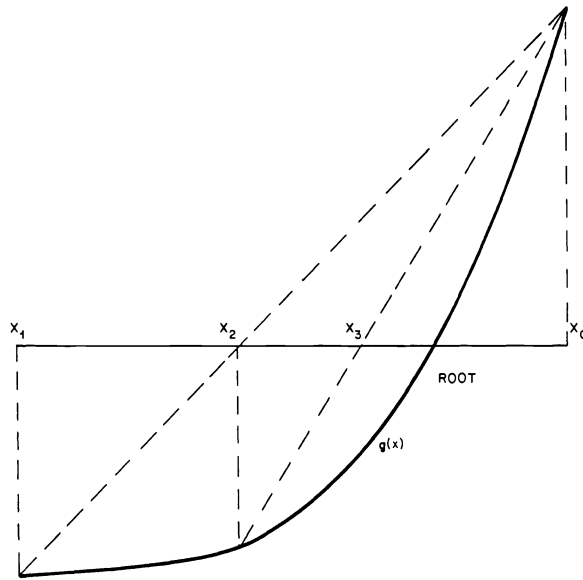


FIG. 1. The curve in the above graph represents the set of points where $g(x) = y$. Note that the points x_1 , x_2 and x_3 , converge upon the root from the left. This type of asymmetry always arises when the second derivative is positive.

formula of (3.1) to derive the first approximation point, x_2 :

$$(3.1) \quad x_2 = x_0 + (x_1 - x_0) \cdot \frac{y - g(x_0)}{g(x_1) - g(x_0)}.$$

Next x_0 and x_2 are combined in like manner to produce the further improved estimate of x_3 . Successive approximations follow until the desired level of accuracy is attained. Further details about the method of regula falsi are offered in [RR-78]. The following theorem and proof illustrate the relationship between this method and interpolation search.

THEOREM 1. *Let μ denote a regular distribution and F an ordered file generated by this density. Suppose that the derivative of μ is continuous and not equal to zero at the point y . Then the expected time for interpolation search to find the record in file F whose key value equals y is $\Omega(\log N)$ with a coefficient inside the Ω -notation that depends on μ .*

The formal proof of Theorem 1 rests on applying concepts from probability theory and numerical analysis in a very tedious but straightforward manner. Since the only purpose of Theorem 1 is to explain our motivation for developing Theorems 2, 3 and 4, its formal proof is omitted, and we instead provide the following informal proof sketch.

Intuitive justification of Theorem 1. Our intuitive proof will require establishing three facts. The first fact is that each key Y_i in a μ -generated file of cardinality N is a random variable with standard deviation bounded by $O(1/\sqrt{N})$. To understand this point, let μ' denote a uniform density on an interval of length $1/b_1$, and Y'_i be the i th smallest order statistic of this distribution. Then Y'_i will have standard deviation $O(1/(b_1\sqrt{N}))$, by the well-known properties of order statistics on uniform densities [KS-77]. The central point is that if μ is a nonuniform density bounded by (b_1, b_2, b_3, b_4)

then Y_i cannot have a greater standard deviation under μ than it had under μ' , since μ has nonzero values on an interval no longer than that for μ' and since $\mu(y)$ is bounded below by b_1 on this interval. Hence, each Y_i must have standard deviation bounded by $O(1/(b_1\sqrt{N}))$. We regard the latter quantity as asymptotically $O(1/\sqrt{N})$, since the notation of Theorem 1 allows the search time coefficient to depend on μ .

Some notation is necessary to introduce the second fact that will be employed in our proof of Theorem 1. Let $f(x)$ denote the integral

$$(3.2) \quad f(x) = \int_{b_3}^x \mu(y) dy,$$

g the inverse of the function f , and $I(\varepsilon)$ the number of iterations that the method of regula falsi, starting with the initial bracketing points b_3 and b_4 , needs to solve $g(x) = y$ with an accuracy ε .

[RR-78] notes that the method of regula falsi requires $\Omega[\log(1/\varepsilon)]$ runtime to make approximations with an accuracy of ε , for any function whose second derivative is positive (intuitively because of an asymmetry where regula falsi converges exclusively from the left, as shown in Fig. 1, when the second derivative is positive). Since g has a positive second derivative whenever μ has a negative first derivative, $I(\varepsilon)$ must clearly have a lower bound $\Omega[\log(1/\varepsilon)]$ when μ has a consistently negative derivative between b_3 and b_4 . By the same reasoning, this lower bound also holds when the derivative of μ is consistently positive.

[RR-78] also observes that $I(\varepsilon)$ is $\Omega(\log(1/\varepsilon))$ for nearly all other types of executions of regula falsi (intuitively because its final iterations typically converge upon an interval that has either locally positive or locally negative derivative for μ). Rather than use this fact to prove Theorem 1 in its full generality, our proof sketch will make the simplifying assumption that μ is a density with either uniformly positive or uniformly negative derivative. In this case, the previous paragraph allows one to automatically presume that $I(\varepsilon)$ is $\Omega(\log(1/\varepsilon))$.

Let $x_1, x_2, \dots, x_{I(N^{-1/4})}$ denote the sequence of tested values for x that regula falsi generates to solve the equation $g(x) = y$ with an accuracy $\varepsilon = N^{-1/4}$, and j_1, j_2, \dots, j_t denote the set of tests that interpolation search makes to find the key Y_j whose stored value is closest to the target y that the algorithm seeks. As the sequence x_1, x_2, \dots was shown to have length $I(N^{-1/4}) \cong \Omega(\log N)$ by the last two paragraphs, we need only prove that the j_1, j_2, \dots, j_t sequence has expected length at least comparable to the length of the $x_1, x_2, \dots, x_{I(N^{-1/4})}$ sequence to establish Theorem 1.

This last step of our intuitive justification of Theorem 1 will be especially informal. (Theorem 1 is proven informally because the remaining details are tedious, and the only purpose of the theorem is to explain our motivation for studying modifications of interpolation search.)

Consider the standard deviation for the random variable Y_{j_s} . Since all elements in the sequence j_1, j_2, \dots, j_t other than j_1 are themselves nonconstant random variables, we cannot apply the first paragraph of our proof-sketch to blithely assume that the Y_{j_s} are random variables with a standard deviation $O(1/\sqrt{N})$. However for any $\frac{1}{4} < \alpha < \frac{1}{2}$, a formal proof can show that the j -values will drift away from their mean at a sufficiently slow rate to assure that Y_{j_s} will have a standard deviation no greater than $O(N^{-\alpha})$ and that the mean value of Y_{j_s} will differ from $g(x_{j_s})$ by an amount also bounded by $O(N^{-\alpha})$. Since the x_1, x_2, \dots sequence requires $\Omega(\log N)$ iterations to produce an x -value satisfying $|g(x) - y| < N^{-1/4}$, the previous sentence implies that the Y_{j_1}, Y_{j_2}, \dots

sequence must have expected length $\Omega(\log N)$ to produce an index j satisfying $|Y_j - y| < N^{-1/4}$.

The main point is that the definition of regular distributions implies that the expected difference between the target y sought by interpolation search and the closest Y_i -value actually stored in the ordered file (Y_1, Y_2, \dots, Y_N) will be $O(1/N)$ (with a coefficient included in the O -notation that is proportional to $1/b_1$). This result implies interpolation search requires time $\Omega(\log N)$ because the previous paragraph showed this many iterations were on the average necessary for the sequence j_1, j_2, \dots to produce an index j satisfying even the weaker constraint $|Y_j - y| < N^{-1/4}$! Q.E.D.

Since binary search is known to require logarithmic retrieval time and since Theorem 1 shows the same is true for applications of interpolation search to nonuniform densities, it is natural to conjecture that any straightforward hybrid of these procedures would also have logarithmic search time on nonuniform densities. The latter statement is surprisingly not true! [Wi-83b] shows that a procedure which simply alternates between the methods of binary and interpolation search actually attains a retrieval time $O(\sqrt{\log N})$.

An even more ideal hybrid is the procedure RETRIEVE defined in § 2. Section 4 will show that RETRIEVE has a log log N search time on nonuniform densities.

Part of the intuition behind this time improvement is that since μ is continuous, it is very similar to the uniform distribution on very short intervals (where it is essentially constant). This implies that the efficiency of interpolation search increases as the probability distribution becomes more nearly uniform. The algorithm RETRIEVE, defined in § 2, takes advantage of these properties by having the cuts produced by its rules I and II resemble those of interpolation search increasingly as l_i gets smaller. The principal theme of this paper is that such a method of gradual transformation into interpolation search leads to the first algorithm with log log N search times on unindexed and nonuniformly generated files.

Although it was not initially conceived in this manner, some partial analogues of RETRIEVE can be found in the numerical literature on the "modified" method of regula falsi [RR-78]. These algorithms are based on the observation that the unmodified version of regula falsi produces a relatively inefficient calculation of the root of a function—with an unfortunate bias toward searching mostly on one side of the designated root. Numerical analysts [RR-78] have found that altering their search points so that there is a more symmetric two-sided convergence upon the root will dramatically improve the efficiency of regula falsi. The positive and negative increments Δ_i of the indices C_i^+ and C_i^- of RETRIEVE lead to a related type of gain in efficiency. The mathematical machinery of this paper can thus be viewed as the synthesis of the probability techniques which have been applied to interpolation search with a relaxation method whose partial analogue can be found in the literature on regula falsi.

4. Runtime analysis of SIMPLE.RETRIEVE. This section will prove that the time complexity of SIMPLE.RETRIEVE satisfies (1.4). We begin by introducing some notation.

For $j = 1, 2$ or 3 , a *level- j* number is an integer which equals $j \pmod 3$. The term *3-cycle* will refer to a sequence of three consecutive iterations of RETRIEVE beginning with some level-1 iteration. This terminology is convenient because RETRIEVE has an inherently periodic nature, executing each of the cutting rules I through III once during every 3-cycle.

In our discussion, it will also be useful to speak of the i th 3-cycle. This term will only be used when i is a level-1 integer, and it refers to the 3-cycle that begins with the iteration i .

We will often characterize the subfile $F_i = (Y_{L_i} < \dots < Y_{R_i})$ that is searched by the i th iteration of SIMPLE.RETRIEVE with the vector $(L_i, R_i, Y_{L_i}, Y_{R_i})$. S_i will denote this vector, and it will be called the *state of the i th iteration*. Intuitively, S_i encompasses all the information known about F_i at the beginning of the i th iteration. Each of the quantities of $l_i, N_i, \Delta_i, C_i^{\text{BIN}}, C_i^{\text{INT}}, C_i^+$ and C_i^- are functions of S_i (since they are computable from the four quantities of L_i, R_i, Y_{L_i} , and Y_{R_i}). Other functions of S_i that we will soon need are:

$$(4.1) \quad \delta_i = \frac{l_i}{\sqrt{N_i}} + (l_i)^{3/2},$$

$$(4.2) \quad \bar{y}_i = Y_{L_i} + \frac{l_i(C_i - L_i)}{N_i}.$$

\bar{y}_i intuitively represents the expected value of the key Y_{C_i} when the records belonging to the subfile $F_i = (Y_{L_i} < \dots < Y_{R_i})$ are generated by the uniform probability distribution. We will also use the quantity \bar{y}_i^* defined in (4.3); it differs from \bar{y}_i by representing the expected value of Y_{C_i} when the keys belonging to the file $F_{i-1} = (Y_{L_{i-1}} < \dots < Y_{R_{i-1}})$ are generated by the uniform distribution.

$$(4.3) \quad \bar{y}_i^* = Y_{L_{i-1}} + \frac{l_{i-1}(C_i - L_{i-1})}{N_{i-1}}.$$

If i is a level-1 iteration, we will say its cut value Y_{C_i} is *good* iff it is sufficiently close to the expected value \bar{y}_i to satisfy:

$$(4.4) \quad |Y_{C_i} - \bar{y}_i| < \delta_i.$$

Similarly if j is level-2, we will say it is *good* if Y_{C_j} satisfies:

$$(4.5) \quad |Y_{C_j} - \bar{y}_j^*| < \delta_{j-1}.$$

The intuitive reason that slightly different rules are used for defining level-1 and level-2 goodness is that § 2 assigned a subscript $i-1$ to cutting rule II but not I. (This subtle distinction will simplify our proof because if i is level-1 then $j = i+1$ is level-2 and the right sides of (4.4) and (4.5) both equal δ_i .) A 3-cycle will be said to produce a *good divide* if both its level-1 and level-2 iterations are good.

The probability of a 3-cycle producing a good divide depends of course on the value of l_i at the beginning of the 3-cycle and on the particular density μ being searched. Appendix B proves $\forall \mu \exists T_\mu > 0$ such that if $l_i < T_\mu$ then the i th 3-cycle will have a probability greater than $\frac{1}{2}$ of producing a good divide. This fact is significant because the next five lemmas will establish the existence of a constant K such that SIMPLE.RETRIEVE can produce no more than $K \log \log N$ good divides while $l_i < T_\mu$. The combination of these two facts will provide the main machinery for proving the log-logarithmic expected time of SIMPLE.RETRIEVE.

LEMMA 1. *Each iteration of a reduction algorithm searching for the key y will satisfy:*

$$(4.6) \quad \frac{(C_i - C_i^{\text{INT}})l_i}{N_i} = \bar{y}_i - y,$$

$$(4.7) \quad \frac{(C_i - C_{i-1}^{\text{INT}})l_{i-1}}{N_{i-1}} = \bar{y}_i^* - y.$$

Proof. Since (4.6) and (4.7) differ only in their subscripts, both have similar proofs, and we will prove only the former.

Equation (4.2) clearly implies

$$(4.8) \quad C_i = L_i + \frac{(\bar{y}_i - Y_{L_i})N_i}{l_i}.$$

The right side of (4.6) then follows by substituting (2.4) and (4.8) into the left side of this equation and simplifying the result. Q.E.D.

Throughout the rest of this paper, we use the special notation convention that if the i th iteration either finds the key y or reduces the search space to the empty set (indicating y is not stored in the file F) then $N_{i+1} = l_{i+1} = 0$. The iteration $i + 1$ will be called *terminating* when these conditions hold.

LEMMA 2. *Suppose i is a level-1 iteration. Then*

- A) *Equation (4.9) will hold if the i th iteration produces a good cut but not a terminating condition.*
- B) *Both (4.9) and (4.10) will hold if the i th 3-cycle produces a good divide but not a terminating condition.*

$$(4.9) \quad y < Y_{C_i} < y + 3\delta_i,$$

$$(4.10) \quad y > Y_{C_{i+1}} > y - 3\delta_i.$$

Proof of assertion A. The proof is divided into two parts, confirming, respectively, $Y_{C_i} < y + 3\delta_i$ and $Y_{C_i} > y$.

Proof that good level-1 iterations satisfy $Y_{C_i} < y + 3\delta_i$. Recall that SIMPLE RETRIEVE differs from RETRIEVE by having Δ_i , C_i^+ and C_i^- defined by (2.8)–(2.10) rather than by (2.5)–(2.7). From the combination of (2.8) and (4.1), we may infer that

$$(4.11) \quad \delta_i = \frac{l_i \Delta_i}{N_i}.$$

Section 2 defined the cut indices of level-1 iterations to be $C_i = \text{MIN}(C_i^+, R_i - 1)$. From (2.8) and (2.9), and the fact² $\Delta_i \geq 1$, it is easy to see then that

$$(4.12) \quad C_i - C_i^{\text{INT}} \leq 2\Delta_i.$$

Multiplying the left and right sides of (4.12) by l_i/N_i and using (4.6) and (4.11) to simplify the result we obtain

$$(4.13) \quad \bar{y}_i - y \leq 2\delta_i.$$

Substituting this inequality into (4.4), we conclude that a good level-1 cut must satisfy $Y_{C_i} < y + 3\delta_i$. Q.E.D.

Proof that good nonterminating level-1 iterations satisfy $Y_{C_i} > y$. The proof is divided into the two cases where $C_i = R_i - 1$ and $C_i = C_i^+$. The proof for the first case is trivial since if $Y_{C_i} \leq y$ then there can be no space for the key y between the addresses $R_i - 1$ and R_i , implying a terminating condition in contradiction to the hypothesis of

² Δ_i is always ≥ 1 in (2.8) because N_i is ≥ 2 for nonterminating iterations.

Lemma 1. The proof for the case $C_i = C_i^+$ uses the fact that (2.9) implies

$$(4.14) \quad C_i - C_i^{\text{INT}} > \Delta_i.$$

Multiplying both sides by l_i/N_i and again using Lemma 1 and (4.11) to simplify the result, we obtain $\bar{y}_i - y > \delta_i$. Inserting this inequality into (4.4), we conclude that good level-1 cuts must satisfy $Y_{C_i} > y$. Q.E.D.

Proof of assertion B of Lemma 2. We have already verified (4.9) and therefore only (4.10) needs examination. The inequality $Y_{C_i} > y$ (from (4.9)) implies that $L_{i+1} = L_i$. Substituting this equality into cutting rule II, we obtain that C_{i+1} equals the larger of $L_i + 1$ and $C_i^{\text{INT}} - \Delta_i$. As this cutting formula is the precise mirror image of cutting rule I, a proof analogous to the proof of (4.9) will verify (4.10). That is, the proof is the same as the confirmation of (4.9) except that all references to (2.9), (4.4), (4.6) and cutting rule I should be replaced by corresponding citations to (2.10), (4.5), (4.7) and cutting rule II. Q.E.D.

LEMMA 3. *Let i again denote a level-1 iteration of SIMPLE.RETRIEVE. If the i th 3-cycle produces a good divide then the state S_{i+3} will satisfy*

$$(4.15) \quad l_{i+3} < 6\delta_i,$$

$$(4.16) \quad N_{i+3} < 4\Delta_i.$$

Proof. If SIMPLE.RETRIEVE terminates during the i th 3-cycle then the equations above will hold because $l_{i+3} = N_{i+3} = 0$. On the other hand, if the 3-cycle is nonterminating then Lemma 2 indicates $y - 3\delta_i < Y_{C_{i+1}} < y < Y_{C_i} < y + 3\delta_i$. This chain of inequalities immediately verifies (4.15), since it implies $l_{i+3} \leq Y_{C_i} - Y_{C_{i+1}} < 6\delta_i$. It also implies $N_{i+3} \leq C_i - C_{i+1}$ (since the chain $Y_{C_{i+1}} < y < Y_{C_i}$ forces $C_{i+1} \leq L_{i+3} \leq R_{i+3} \leq C_i$). By the definition of cutting rules I and II, $C_i - C_{i+1} \leq C_i^+ - C_i^-$. The right side of the latter inequality is less than $4\Delta_i$, by the combination of (2.9), (2.10) and the fact that $\Delta_i > 1$ under SIMPLE.RETRIEVE. Equation (4.16) follows by combining the inequalities from the last three sentences. Q.E.D.

We need one more definition before we can introduce our main theorems and lemmas. Define the 3-cycle that begins with the iteration i to be:

- 1) Type 1 iff the state S_{i+3} at the end of this cycle satisfies $N_{i+3} < 8\sqrt{N_i}$;
- 2) Type 2 iff this cycle satisfies simultaneously:

$$(4.17) \quad l_{i+3} < 12l_i^{3/2},$$

$$(4.18) \quad l_i < 1/72,$$

$$(4.19) \quad l_i > 1/N_i.$$

The remainder of our analysis of SIMPLE.RETRIEVE will have three parts. Lemma 4 will show that the condition of a good divide combined with $l_i < \frac{1}{72}$ implies the presence of either a Type-1 or Type-2 event. Lemma 5 will state that no more than $O(\log \log N)$ such events may occur before the search terminates. The last part of our discussion will show that these events occur with sufficient probability to imply the $\log \log N$ runtime of SIMPLE.RETRIEVE.

LEMMA 4. *Let i again denote a level-1 iteration of SIMPLE.RETRIEVE. If $l_i < \frac{1}{72}$ and if the i th cycle produces a good divide then this 3-cycle will also produce either a Type-1 or Type-2 cycle.*

Proof. It is useful to divide our proof into two cases where we separately consider the possibilities that $l_i \leq 1/N_i$ and $l_i > 1/N_i$.

Proof that good divides satisfy Type-1 when $l_i \leq 1/N_i$. Substituting this inequality into (2.8), we obtain $\Delta_i \leq 2\sqrt{N_i}$. In this context, Lemma 3 implies a good divide satisfies $N_{i+3} < 8\sqrt{N_i}$, i.e. it is a Type-1 event. Q.E.D.

Proof that Type-2 events occur when $l_i > 1/N_i$ and the hypothesis of Lemma 4 is satisfied. The assumptions in this case already imply that l_i satisfies (4.18) and (4.19), and therefore all that remains to prove is (4.17). The proof of this inequality is also straightforward since (4.1) and the inequality $l_i > 1/N_i$ imply $\delta_i < 2l_i^{3/2}$, and we can then apply the first equation from Lemma 3 to conclude that good divides satisfy (4.17). Q.E.D.

LEMMA 5. *There exist constants K_1 and K_2 such that no more than $K_1 \log \log N_1$ Type-1 and $K_2 \log \log N_1$ Type-2 cycles can occur in the course of the search of a file whose initial cardinality is N_1 .*

We have divided the proof of Lemma 5 into two parts.

Proof for Type-1 events. The definition of an iterative reduction algorithm implies its iterations will satisfy $N_{i+1} \leq N_i - 1$. This inequality, combined with the definition of Type-1 events, implies that the number of Type-1 events needed to cause the N -value of the state S to decrease to less than or equal to 1 is $O(\log \log N_1)$. Since all iterative algorithms terminate by the time $N_i = 1$, this observation implies no more than $K_1 \log \log N_1$ events can occur, for some constant K_1 . Q.E.D.

Proof for Type-2 events. From the definition of Type-2 events, it is easy to see that each such occurrence will cause $\log \log (1/l)$ to increase by at least $\Omega(1)$ between the times of states S_i and S_{i+3} . Since $l_i < \frac{1}{72}$ at the time of the first Type-2 event, and since l_i always decreases as i increases, it follows that a sequence of $\Omega(\log \log N_1)$ Type-2 events will cause $l_i < 1/N_1$, making l_i too small for any subsequent iteration to satisfy the definition of Type-2. Therefore, we have verified the existence of some constant K_2 such that no more than $K_2 \log \log N_1$ Type-2 events can occur. Q.E.D.

Appendix B uses well-known methods from numerical analysis and statistics to prove the following result:

THEOREM 2. $\forall B \exists T > 0$ such that if μ is a regular density bounded by B and if $l_i < T$ at the beginning of the i th 3-cycle, then this 3-cycle will have a probability exceeding $\frac{1}{2}$ of producing either a good divide or a termination.

We will now use Theorem 2 to prove our main two theorems.

THEOREM 3. $\exists K \forall B \exists T_B > 0$ such that if several iterations of SIMPLE.RETRIEVE have already searched a probability density μ bounded by B and produced a state where $l_i < T_B$ then the expected number of additional 3-cycles to complete the search is $\leq K \log \log N_i$. In particular, if K_1 and K_2 are the constants satisfying Lemma 5, then one constant satisfying Theorem 3 is $K = 2(K_1 + K_2)$.

Proof. Let T denote the constant defined by Theorem 2 and $T_B = \text{MIN}(T, \frac{1}{72})$, and $I(S_i)$ denote the expected number of 3-cycles for SIMPLE.RETRIEVE to complete its search once it has reached a state S_i satisfying $l_i < T_B$. Also, let $I^*(S_i)$ denote the expected numbers of events of Types 1 and 2 during this search. Lemma 5 implies $I^*(S_i) \leq (K_1 + K_2) \log \log N_i$, and the combination of Lemma 4 and Theorem 2 imply $I^*(S_i) \geq I(S_i)/2$. Combined these inequalities imply $I(S_i) \leq 2(K_1 + K_2) \log \log N_i$.

Q.E.D.

We now state the main result of this paper where SIMPLE.RETRIEVE is shown to have an expected time $K \log \log N + K_B$ with only the second coefficient depending on the bounding vector B of μ .

THEOREM 4. $\exists K \forall B \exists K_B \forall \mu \forall N$: If μ is a probability density bounded by B and F is a file of N records generated by μ , then $K \log \log N + K_B$ upper bounds the expected number of 3-cycles needed by SIMPLE.RETRIEVE to search F .

Proof. Since the level-3 iterations of SIMPLE.RETRIEVE are defined to be binary cuts and since (1.1)–(1.3) imply that the probability density of μ is bounded below by b_1 and above³ by $b_1 + b_2(b_4 - b_3)$, it is easy to prove that $\forall B \exists \lambda < 1$: the expected value of l_{i+3} is $< \lambda l_i$ whenever SIMPLE.RETRIEVE begins in a state S_i and is searching a density bounded by B . Let T_B denote the constant defined by the $\exists \forall \exists$ statement of Theorem 3. Since l_{i+3} is always $< l_i$, the first sentence implies $\forall B \exists K_B \forall \mu \forall N: K_B$ bounds the expected number of 3-cycles for SIMPLE.RETRIEVE to reach a state where $l < T_B$ (when it is searching N records generated by a density μ bounded by B).

Theorem 3 indicates that the additional search time after reaching such a state is $\leq K \log \log N$. The total cost is therefore $\leq K \log \log N + K_B$. Q.E.D.

5. Extensions and open questions. The main open question raised by this paper is how to minimize the constants K and K_B . Our intuition is that these constants can be given practical values in many computer applications, but this paper has favored brevity of proof over a more detailed analysis.

One technique used to abbreviate the proof was to endow RETRIEVE with several features which unnecessarily increase the coefficient but simplify the presentation. For example, the subscript $i-1$ in the cutting rule II in § 2 is clearly less efficient than a subscript i , but it simplifies the proof because the consecutive level-1 and level-2 iterations then use the same values for Δ and C^{INT} in their definitions of C^+ and C^- . Another example is that Theorem 4 generalizes for any $0 < \alpha < 1$, $\theta > 0$ and $\phi \geq 0$, but we have focussed on the particular case $\alpha = \frac{1}{2}$ and $\theta = \phi = 1$ for simplicity. Our results do not generalize when $\theta = 0$, but the proof methods from [Wi-83b] imply a time $O(\sqrt{\log N})$ is possible in this case. If α were set equal to one then Theorems 3 and 4 would not hold, but a modified result would be valid which would state $\forall B \exists \theta \exists \phi \exists K$ insuring RETRIEVE $(1, \theta, \phi)$ has an expected time $K \log \log N$ on N element files whose generating density is bounded by B . This degenerate version could have practical implications since the coefficient should improve when one does not have to raise l to a fractional power. Also, note that Theorem 4 generalizes when $\phi = 0$. We assigned ϕ a nonzero value in this paper to simplify the proof.

There are several more elaborate refinements of RETRIEVE which complicate the proof but lead to better coefficients. One rule with a better coefficient would set C_i equal to:

- i) the median value of the three indices $L_i + \Delta_i$, C_i^{INT} , and $R_i - \Delta_i$ when $\Delta_i < N_i/2$,
- ii) C_i^{BIN} when $\Delta_i \geq N_i/2$.

Another example would set C_i equal to the median value of the indices C_i^{BIN} , C_i^+ , and C_i^- . A third alternative consists of setting C_i equal to:

- a) the median value of C_i^{BIN} , C_i^+ , and $R_i - 1$ when i is an even number,
- b) the median value of C_i^{BIN} , C_i^- , and $L_i + 1$ when i is odd.

All the procedures above would become even more efficient if (2.5) was replaced by

$$(5.1) \quad \Delta_i = \theta l_i^\alpha N_i + \phi \sqrt{(R_i - C_i)(L_i - C_i) / N_i^3}.$$

Numerous other algorithms can also be developed for attaining $\log \log N$ runtime on nonuniform regular probability densities, as can be surmized by surveying the large number of papers that numerical analysts have written about the method of regula falsi and its modifications (see [RR-78] for some references). Many of these numerical procedures can be modified to attain a $\log \log N$ asymptote in the context of the special

³ This upper bound is tight for some but not other vectors B .

searching problems in this article. One striking fact is that none of the commonly used versions of the method of regula falsi is directly analogous to RETRIEVE. It would be useful to determine which of the many possible alternatives has the best coefficient in typical applications.

The $\log \log N$ result of this paper can also extend beyond the set of regular probability densities to nondifferentiable functions. Specifically, $\log \log N$ time will remain valid for certain values of α , θ and ϕ when (1.2) is replaced by the weaker requirement that there exist constants b_{21} and $b_{22} > 0$ such that all y_1 and y_2 satisfy

$$(5.2) \quad |u(y_1) - u(y_2)| \leq b_{21}|y_2 - y_1|^{b_{22}}.$$

Also, if one makes the further assumption that the probability density u plays the *double* role of generating both the construction of file F and the requests for retrieving records from it, then the $\log \log N$ expected runtime of this paper can extend to densities that do not satisfy (1.1) (i.e. with zeros).

Readers who found this paper interesting may also wish to examine [Wi-84b], [Wi-84c], where we discuss two other applications of randomized search algorithms, and [MT-84]. The latter paper was stimulated by [Wi-81], [Wi-83b], and it generalizes our result by making the procedure slightly more robust than in the previous paragraph and dynamic. An important distinction between our work and [MT-84] is that we use precisely N units of space to represent N records, and [MT-84] obtains further improvements by increasing the memory space by a constant factor.

Appendix A. This appendix gives the formal algorithmic definitions for *iterative reduction* and for the algorithm RETRIEVE. In order to make our discussion precise, Y_0 and Y_{N+1} designate lower and upper bounds on the permissible range of keys. Also assume the file examined by the first iteration of the algorithm A has been padded with these pseudo-records, and it is thus equivalent to $F_1 = (Y_0 < \dots < Y_{N+1})$. (This assumption has appeared in the previous literature; it is necessary to make C_i^{INT} consistently well defined, and it is equivalent to the statements $Y_0 = b_3$ and $Y_{N+1} = b_4$ in the context of the definition of regularity (i.e. (1.1)-(1.3))).

Each iterative reduction algorithm A will possess an associated subroutine, a , whose function is to generate a cut index C_i when given arguments consisting of a subfile F_i , the record y sought by A, and an integer i indicating the number of the present iteration. The formal description of an iterative reduction procedure is given below. The specific algorithm RETRIEVE (α, θ, ϕ) is the specialized version of this procedure that results from the cuts generated by rules I-III of § 2.

```

PROCEDURE A [( $Y_L < \dots < Y_R$ );  $y$ ;  $i$ ]
  IF  $R = L + 1$  THEN RETURN ("Can't Find Key");
  ELSE DO:
    SET  $C$  = an address between  $L$  and  $R$  determined
      by the subroutine  $a$ .
    IF  $Y_C = y$  THEN RETURN ("Found Key");
    IF  $Y_C < y$  THEN CALL A [ $Y_C < \dots < Y_R$ ];  $y$ ;  $i + 1$ ];
    IF  $Y_C > y$  THEN CALL A [( $Y_L < \dots < Y_C$ );  $y$ ;  $i + 1$ ];
  END;
END;
```

Appendix B. Consider a random process that uses the uniform probability distribution to draw M independent keys between zero and one and then sets X_i^M equal to

the i th smallest of these keys. Our proof of Theorem 2 will rest on three preliminary lemmas.

LEMMA 6. $\text{Prob} \{ |X_i^M - i/(M+1)| < 1/\sqrt{M+1} \} \geq \frac{3}{4}$.

Proof. Let $P_i^M(x)$ denote the probability density indicating the likelihood that $X_i^M = x$. It is well known [KS-77] that $P_i^M(x)$ equals the beta function below:

$$(B.1) \quad P_i^M(x) = M \binom{M-1}{i-1} x^{i-1} (1-x)^{M-i}.$$

This function is known [KS-77] to have mean $= i/(M+1)$ and variance $= i(M+1-i)/((M+1)^2(M+2))$. Therefore Chebyshev's inequality implies

$$(B.2) \quad \text{Prob} \left(\left| X_i^M - \frac{i}{M+1} \right| < 2 \sqrt{\frac{i(M+1-i)}{(M+1)^2(M+2)}} \right) \geq \frac{3}{4}.$$

The inequalities $M+1 < M+2$ and $i(M+1-i)/(M+1)^2 \leq \frac{1}{4}$ certainly imply

$$(B.3) \quad 2 \sqrt{\frac{i(M+1-i)}{(M+1)^2(M+2)}} < \sqrt{\frac{1}{M+1}}.$$

Lemma 6 follows by substituting the right side of (B.3) into (B.2). Q.E.D.

We will now introduce the main notation employed to prove Theorem 2. For any $a_1 < a_2$ let $I_{\mu a_1 a_2}(y)$ denote the probability that a record randomly generated by μ is $< y$ when it is known that this record lies between a_1 and a_2 . $I_{\mu a_1 a_2}(y)$ intuitively represents the cumulative distribution function for the restriction of μ to the interval (a_1, a_2) ; it is formally defined by (B.4).

$$(B.4) \quad I_{\mu a_1 a_2}(y) = \frac{\int_{a_1}^y \mu(x) dx}{\int_{a_1}^{a_2} \mu(x) dx}.$$

Also let $I_{a_1 a_2}(y)$ denote the cumulative distribution function for the uniform distribution over the interval (a_1, a_2) , i.e.

$$(B.5) \quad I_{a_1 a_2}(y) = \frac{(y - a_1)}{(a_2 - a_1)}.$$

The symbols $I_{\mu a_1 a_2}^{-1}$ and $I_{a_1 a_2}^{-1}$ will denote the inverses of the functions defined in (B.4) and (B.5). Thus (B.5) implies

$$(B.6) \quad I_{a_1 a_2}^{-1}(x) = a_1 + x(a_2 - a_1).$$

We need two more preliminary lemmas to help prove Theorem 2.

LEMMA 7. $\forall B \exists T > 0 \forall \mu \forall x$: If μ is bounded by B and if $0 < (a_2 - a_1) < T$ then

$$(B.7) \quad |I_{\mu a_1 a_2}^{-1}(x) - I_{a_1 a_2}^{-1}(x)| < (a_2 - a_1)^{3/2}.$$

Proof. The definition of regular probability densities (i.e. (1.1)-(1.3)) implies $\forall B \exists C$ such that all μ bounded by B and all $a_1 a_2$ and y satisfy

$$(B.8) \quad |I_{\mu a_1 a_2}(y) - I_{a_1 a_2}(y)| < C(a_2 - a_1).$$

Equations (B.6) and (B.8) imply that the inverse of $I_{\mu a_1 a_2}$ will satisfy a similar condition, i.e. they imply $\forall B \exists k$ such that all μ bounded by B and all $a_1 a_2$ and x satisfy

$$(B.9) \quad |I_{\mu a_1 a_2}^{-1}(x) - I_{a_1 a_2}^{-1}(x)| < k(a_2 - a_1)^2.$$

Define T to equal $1/k^2$. Then it is apparent that $|I_{\mu a_1 a_2}^{-1}(x) - I_{a_1 a_2}^{-1}(x)| < (a_2 - a_1)^{3/2}$ whenever $(a_2 - a_1) < T$. Our construction has thus shown that the value of T depends only on the vector B , as claimed by Lemma 7. Q.E.D.

LEMMA 8. $\forall B \exists T > 0$ such that if $0 < (a_2 - a_1) < T$, if μ is bounded by B , and if M records are randomly generated by the restriction of μ onto the interval (a_1, a_2) then the j th smallest of these records, denoted Y_j^M , will satisfy

$$(B.10) \quad \text{Prob} \left\{ \left| Y_j^M - I_{a_1 a_2}^{-1} \left(\frac{j}{M+1} \right) \right| < \frac{a_2 - a_1}{\sqrt{M+1}} + (a_2 - a_1)^{3/2} \right\} > \frac{3}{4}.$$

Proof. The random variable $I_{\mu a_1 a_2}(Y_j^M)$ must have the exact same probability distribution as X_j^M , since X_j^M and Y_j^M differ only by the change of coordinates defined by $I_{\mu a_1 a_2}$. Lemma 6 therefore implies

$$(B.11) \quad \text{Prob} \left\{ \left| I_{\mu a_1 a_2}(Y_j^M) - \frac{j}{M+1} \right| < \frac{1}{\sqrt{M+1}} \right\} > \frac{3}{4}.$$

An immediate consequence of (B.11) is:

$$(B.12) \quad \text{Prob} \left\{ I_{\mu a_1 a_2}^{-1} \left[\max \left(0; \frac{j}{M+1} - \frac{1}{\sqrt{M+1}} \right) \right] < Y_j^M \right. \\ \left. < I_{\mu a_1 a_2}^{-1} \left[\min \left(1; \frac{j}{M+1} + \frac{1}{\sqrt{M+1}} \right) \right] \right\} > \frac{3}{4}.$$

The assertion of Lemma 8 now follows by substituting Lemma 7 and (B.6) into the first two inequalities in the statement (B.12). Q.E.D.

We are now ready to prove Theorem 2. The proposition claims: $\forall B \exists T > 0$ such that if SIMPLE.RETRIEVE is searching a density bounded by B and if $l_i < T$ at the beginning of its i th 3-cycle then there will be a probability exceeding $\frac{1}{2}$ that either this cycle produces a good divide or that it produces a termination.

Proof of Theorem 2. Let B denote μ 's bounding vector and T the constant satisfying Lemma 8. Assume the i th 3-cycle begins in a state S_i where $l_i < T$, and let J_+ and J_- denote the two quantities defined below.

$$(B.13) \quad J_+ = \min (R_i - 1; \lceil C_i^{\text{INT}} + \Delta_i \rceil),$$

$$(B.14) \quad J_- = \max (L_i + 1; \lfloor C_i^{\text{INT}} - \Delta_i \rfloor).$$

Then after an easy translation of notation, Lemma 8 implies that there is a probability greater than $\frac{1}{2}$ that (B.15) and (B.16) simultaneously hold:

$$(B.15) \quad \left| Y_{J_+} - Y_{L_i} - \frac{(J_+ - L_i)l_i}{N_i} \right| < \delta_i,$$

$$(B.16) \quad \left| Y_{J_-} - Y_{L_i} - \frac{(J_- - L_i)l_i}{N_i} \right| < \delta_i.$$

Since i is a level-1 iteration, it is apparent that $C_i = J_+$. Substituting this equality and (4.2) into (B.15), we obtain that the iteration i will satisfy (4.4) whenever it satisfies (B.15). This fact is significant because (4.4) is the first of the two conditions necessary to establish the occurrence of a good divide.

We claim that either a termination or a good divide must occur whenever (B.15) and (B.16) both hold. If a termination does not occur then the previous paragraph combined with Lemma 2A implies $Y_{C_i} > y$, which in turn implies that $C_{i+1} = J_-$.

Substituting this equality and (4.3) into (B.16), we obtain (4.5), by the same reasoning that was used to verify (4.4) in the last paragraph. The claim of the current paragraph follows now from the fact that (B.15), (B.16) and the absence of a termination imply the satisfaction of (4.4) and (4.5).

Since the first paragraph of this proof showed that (B.15) and (B.16) held with a joint probability greater than $\frac{1}{2}$, this probability must also characterize the likelihood that either a good divide or termination occurs. Q.E.D.

Acknowledgments. I am very grateful to S. Bing Yao for suggesting that I study applications of interpolation search to nonuniform densities. I also thank George S. Lueker and the referee for several useful comments on how to improve the presentation.

REFERENCES

- [AFK-83] M. ATJAI, M. FREDMAN AND J. KOMLOS, *Hash functions for priority queues*, Proc. 24th IEEE Symposium on Foundations of Computer Science, 1983, pp. 299–303.
- [De-79] L. DEVROYE, *Average time behavior of distributive sorting algorithms*, Tech. Rep. SOCS 79.4, 1979.
- [ES-74] P. ERDOS AND J. SPENCER, *Probabilistic Methods in Combinatorics*, Academic Press, New York, 1974.
- [Fe-68] W. FELLER, *An Introduction to Probability Theory and Its Applications*, Vol. 1, third ed., John Wiley, New York, 1968.
- [FKS-82] M. FREDMAN, J. KOMLOS AND E. SZEMEREDI, *Storing a space table with $O(1)$ worst-case access time*, Proc. 23rd IEEE Symposium on Foundations of Computer Science, 1982, pp. 165–170.
- [Fr-79] W. R. FRANKLIN, *Padded lists: set operations in expected $O(\log \log N)$ time*, Inform. Process. Lett., (1979), pp. 161–166.
- [GRG-80] G. H. GONNET, L. D. ROGERS AND J. A. GEORGE, *An algorithmic and complexity analysis of interpolation search*, Acta Inform., 13 (1980), pp. 39–52.
- [HKW-85] M. HOFRI, A. KONHEIM AND D. WILLARD, *Padded lists revisited*, to appear in this Journal.
- [IKR-80] A. ITAI, A. KONHEIM AND M. RODEH, *A sparse table implementation of priority queues*, IBM Tech. Rep. RC 8550 (#37269), Nov. 1980. Extended abstract appeared at ICALP-1981.
- [Jo-81] D. JOHNSON, *A priority queue in which initialization and queue operations take $O(\log \log D)$ time*, Rept. No. CS 81-13, Pennsylvania State Univ., University Park, PA, 1981.
- [Kn-73] D. KNUTH, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
- [KS-77] M. KENDALL AND A. STUART, *Advanced Theory of Statistics*, Vol. 1, fourth ed., Macmillan, New York, 1977.
- [MT-84] K. MEHLHORN AND A. TSAKALIDIS, *Dynamic interpolation search*, Univ. des Saarlandes, 84-05, 1984.
- [MG-80] R. MELVILLE AND D. GRIES, *Controlled density sorting*, Inform. Process. Lett., 10 (1980), pp. 169–172.
- [PIA-78] Y. PEARL, A. ITAI AND H. AVNI, *Interpolation search—a log log N search*, Comm. ACM, 21 (1978), pp. 550–554.
- [PR-77] Y. PEARL AND E. M. REINGOLD, *Understanding the complexity of interpolation search*, Inform. Process. Lett., 6 (1977), pp. 219–222.
- [RR-78] A. RALSTON AND P. RUBINOWITZ, *A First Course in Numerical Analysis*, second ed., McGraw-Hill, New York, 1978, pp. 338–344.
- [RW-84] J. REIF AND D. WILLARD, *Parallel interpolation search*, Inform. and Control, submitted for publication.
- [TY-79] R. TARJAN AND A. YAO, *Storing a sparse table*, Comm. ACM, 22 (1979), pp. 606–611.
- [Va-75] P. VAN EMDE BOAS, *Preserving order in a forest in less than logarithmic time*, Proc. 16th IEEE Symposium on the Foundations of Computer Science (1975), pp. 75–84.
- [Va-77] ———, *Preserving order in a forest in less than logarithmic time and linear space*, Inform. Process. Lett., 6 (1977), pp. 80–82.

- [VKZ-77] P. VAN EMDE BOAS, R. KAAS AND E. ZIJLSTRA, *Design and implementation of an efficient priority queue*, Math. Systems Theory 10 (1977), pp. 99–127.
- [Wi-81] D. WILLARD, *A log log N search algorithm for nonuniform distributions*, Extended Abstract in Proceedings of ORSA-TIMS Conference on the Applied Probability-Computer Science Interface, Volume II, pp. 3–14, 1981; this is the first version of our paper on interpolation search.
- [Wi-82] ———, *Maintaining dense sequential files in a dynamic environment*, extended abstract at the 14th ACM Symposium on Theory of Computing, pp. 114–122; full-length version invited paper to appear in Inform. and Control.
- [Wi-83a] ———, *Log-logarithmic Worst-case range queries are possible in space $O(N)$* , Inform. Process. Lett., 17 (1983), pp. 81–84.
- [Wi-83b] ———, *Surprisingly efficient search algorithms for nonuniformly generated files*, 21st Allerton Conference on Communication Control and Computing, 1983, pp. 656–662.
- [Wi-84a] ———, *Two very fast Trie data structures*, extended abstract at 19th Allerton Conference on Communication Control and Computing, 1981, pp. 355–363; full-length version in J. Comput. System Sci., 28 (1984), pp. 379–384.
- [Wi-84b] ———, *Log-logarithmic protocols for ethernet and semaphore-like problems*, Proc. 16th ACM Symposium on Theory of Computing, 1984, pp. 512–521.
- [Wi-84c] ———, *Sampling algorithms for differentiable batch retrieval problems*, Proc. 11th International Conference on Automata, Languages and Programming (ICALP), 1984, pp. 514–526.
- [Ya-82] A. C. YAO, *Should tables be sorted?* J. Assoc. Comput. Mach., 28 (1981), pp. 615–628.
- [YY-76] A. C. YAO AND F. F. YAO, *The complexity of searching an ordered random table*, Proc. 17th IEEE Symposium on Foundations of Computer Science, 1976, pp. 173–177.

PROBABILISTIC MODELS AND ASYMPTOTIC RESULTS FOR CONCURRENT PROCESSING WITH EXCLUSIVE AND NON-EXCLUSIVE LOCKS*

DEBASIS MITRA†

Abstract. We give a probabilistic model for conflicts among transactions in a database. We extend recent work (Mitra and Weinberger, *J. Assoc. Comput. Mach.*, 31 (1984), pp. 855–878) by allowing two kinds of locks for concurrency control, exclusive and non-exclusive. Thus read locks allow shared access and are therefore non-exclusive, while write locks are exclusive. An arriving transaction is blocked if it conflicts with any of the transactions already undergoing processing; otherwise, it is accepted for concurrent processing. Our treatment, which has probabilistic, combinatorial and analytic components, yields exact formulas for performance measures, such as mean concurrency and throughput. For large databases special formulas are derived and proven to be asymptotically exact. These formulas are simple and also fit well to the exact formulas. What are non-exclusive locks worth? A succinct answer is given for this important design question.

Key words. parallel processing, shared data, databases, concurrency control, asymptotics, stochastic networks.

1. Introduction. In this paper we probabilistically model the effect of conflicts among transactions in a database and obtain exact formulas for equilibrium performance measures. We extend the work reported in [1] by allowing the transactions to require two kinds of locks, exclusive and non-exclusive. Thus read locks allow shared access and are therefore non-exclusive, while write locks are exclusive. Formulas for large databases are derived and proven to be asymptotically exact; these formulas are simple, amenable to heuristic interpretations and found to fit well to the solutions obtained by the exact formulas.

What are non-exclusive locks worth? We obtain here a succinct answer to the above question. Obviously non-exclusive locking increases the level of concurrency in transaction processing, but it is important to know the extent of the improvement as it has to be weighed against the additional cost of administering such a facility.

There are many different concurrency control methods in databases, but all share the characteristic that when there is a conflict over the use of a resource, either a transaction is delayed, or a transaction may have to have its work undone later. Generally the conflict is over some item in the database, and without some concurrency control the database may end up in an inconsistent state. We give quantitative measures of the effects of conflicts for the model to be described below.

We use the language of databases in this paper, but the problem of quantifying the effects of conflicts is inherent to parallel processing with shared memory. Processes requiring common variables in memory interfere with each other and when this occurs concurrency control algorithms with locking as an important component are required to resolve conflicts. The results obtained here are of interest in this general context.

We model concurrency control by assuming that each transaction obtains all its locks before it begins processing. The central focus of this investigation is specifically the interference phenomenon among transactions, the cause of conflicts. In our model each transaction is specified by a list of nonrepeated items in which each item requires either an exclusive lock or a non-exclusive lock. Any two transactions interfere with each other if and only if, either an item is required to be exclusively locked by both

* Received by the editors December 15, 1983, and in revised form August 25, 1984.

† AT & T Bell Laboratories, Murray Hill, New Jersey 07974.

transactions or an item is required to be both exclusively locked and non-exclusively locked by the transactions. An arriving transaction is blocked if it interferes with any of the transactions already undergoing processing. If an arriving transaction is not blocked, then it is accepted for processing. Processing of all transactions is done concurrently. Note that at any time all transactions undergoing processing are mutually non-interfering.

We exactly analyze the above model. Our treatment has probabilistic, combinatorial and analytic components. The mathematical model is probabilistic with an underlying Markov process, in which the states are in a one-to-one correspondence with combinations of mutually non-interfering transactions. The equilibrium Markov process is tractable on account of it being time-reversible. This tractability reduces the problem to one of evaluating the equilibrium distribution's partition function. The tie with combinatorics arises thus: the partition function, while originating in the equilibrium distribution of the Markov process, is also the generating function of the sequence which enumerates the number of states having an identical number of constituent transactions. The third and final part of the paper is on the asymptotic analysis of the formulas obtained in the earlier parts of the paper. Formulas are derived which are asymptotically exact for large databases.

Recently several papers [2]-[7] have appeared which model the behavior of databases. However, we know of no previous paper¹ which reports on performance results from exact probabilistic models of the interference phenomenon in the presence of non-exclusive locking. On the other hand, we have made several simplifying assumptions on aspects of real systems. The most important of these is the assumption that the locks on all the items involved in a transaction which is accepted for processing are granted in one atomic operation. Usually [8] locks are obtained during the course of a transaction, and released when the transaction commits or aborts. If the part of the transaction during which locks are being obtained is short, then our model is a good approximation. Alternatively, blocking in our model corresponds to rejecting a transaction using optimistic concurrency control [8]. The concurrency control modelled here shares with optimistic concurrency control the following features: conflict detection is done only at one time, and the conflicting transactions are aborted so that no waiting or deadlock is involved. However, the two differ in that in the latter case the transaction commits or aborts at the conclusion of processing, while in the model treated here this is done at the beginning.

Another important simplifying feature of our model is that it is a blocking system. That is, blocked transactions are lost. In reality blocked transitions are resubmitted for processing. However, blocking systems provide fundamental insights even into lossless systems. In [1] a method has been given for using the results for the blocking system to approximate a system in which blocked transactions wait for a random period before retrying for the necessary locks. This method applies as well in the present context, i.e. in the presence of non-exclusive locking.

There are certain noteworthy and fundamental differences from the analysis in [1] beside the obvious additional complexity due to the presence of non-exclusive locks. First, in [1] it was possible to obtain a Markov process which was an aggregate of the basic, primitive Markov process which models the interactions between the transactions. The aggregated process was convenient since it was relatively simpler and also because the system performance measures could be obtained from it. Here,

¹ Note added in proof. A recent exception is by S. S. Lavenberg, *A simple analysis of exclusive and shared lock contention in a database system*, Proc. 1984 Sigmetrics Conference, Cambridge, MA, pp. 143-148.

no such relatively simple aggregated Markov process exists. The system performance measures need to be computed here from the solution of the basic Markov process. Second, a recursive formula for the partition function was derived in [1] and this too we were unable to extend. However, the important asymptotic formulas in [1] are completely subsumed by the results reported here.

We draw attention to some drawbacks of the model considered here and the potential of overcoming them in future work. First, we note that the present model does not admit skewness of the usage profile of individual items; i.e., all items are equally likely to be constituents of transactions. We have investigated a generalization of the present model in which the database is divided into subsystems, or sub-databases, and items in any subsystem are homogeneous in being equally likely to be constituents of transactions, but the transactions may have preferences concerning the subsystem from which it may pick items for their composition. As an example, the following extreme case is allowed in the generalized model: all transaction lists include a particular item; to handle this case, a subsystem is defined with the particular item as its sole member. Such a generalized model has features of a stochastic network, and we have been able to extend most of the analytical techniques developed here to the more general model. However, further work needs to be done.

The performance of realistic databases are constrained by other factors besides locking and these have been ignored in the present analysis. These include, in addition to the aforementioned skewness in usage profiles of items, available processing power and data transfer capacity for transaction processing. We should point out that in [1] we have considered this. In [1, §6] we have given the solution for a system in which, in addition to locking, the details of transaction processing are modelled by a queueing network. The reader may verify that this result in conjunction with the results given in this paper naturally extends to yield a solution for a generalized model which includes the constraints of transaction processing, as well as exclusive and non-exclusive locking. However, efficient algorithms for the computation of the solution are not available at present.

1.1. Physical model. We let the database be composed of N items where an item is the smallest entity in the database which may be locked. Transactions are associated with lists of database items which are to be processed. The lists are partitioned into two parts, with the items in the leading part requiring exclusive locks and items in the trailing part requiring only non-exclusive locks. Thus $T = (I_1; I_2, I_3)$ signifies that transaction T calls for the processing of items I_1, I_2, I_3 in the database and, furthermore, item I_1 requires an exclusive lock, while items I_2 and I_3 require only non-exclusive locks. The items in a transaction list are not repeated. We assume that all transactions exclusively lock at least one item.

Requests for transaction processing arrive exogenously to the database. On arrival of such a request the database lock manager decides to either grant or refuse the locks required on the following basis. Let W_d and R_d be the lists of exclusively locked and non-exclusively locked items respectively, in the database at the time of arrival. Also let W_a and R_a be the lists of items required to be exclusively locked and non-exclusively locked, respectively, by the arriving transaction. The locks are granted if

$$(1) \quad (W_a \cap (R_d \cup W_d) = \emptyset) \cap (R_a \cap W_d = \emptyset),$$

and denied otherwise. If the locks are denied then the request for processing the transaction is blocked and cleared, i.e. discarded. Otherwise, the transaction is accepted for processing and the lock manager places the appropriate locks on all the items in

the accepted transaction's list in one atomic operation. The locks are not released until the entire processing of the transaction is complete, at which point all the locks involved with the transaction are simultaneously released.

All transactions that are accepted for processing are processed concurrently, i.e. in parallel.

There are p classes of transactions. The transactions in a class have the same number of items requiring exclusive locks and non-exclusive locks. For example, these numbers may be 1 and 2, respectively, in which case all transactions in this class have 3 items to process, 1 item requiring an exclusive lock and 2 additional items requiring non-exclusive locks. In general we let j_σ and k_σ denote the numbers of items requiring exclusive locks and non-exclusive locks, respectively, in each transaction of class σ , $1 \leq \sigma \leq p$. Hence there are

$$\binom{N}{j_\sigma + k_\sigma} \binom{j_\sigma + k_\sigma}{j_\sigma}$$

transactions in class σ . A typical transaction in class σ , $T_i^{(\sigma)}$, is associated with a partitioned string of items thus

$$(2) \quad T_i^{(\sigma)} = (I_{i(1)}, \dots, I_{i(j_\sigma)}; I_{i(j_\sigma+1)}, \dots, I_{i(j_\sigma+k_\sigma)}), \quad 1 \leq i \leq \binom{N}{j_\sigma + k_\sigma} \binom{j_\sigma + k_\sigma}{j_\sigma}.$$

Transaction classes may be distinguished in other respects, such as arrival rates and processing times.

We reserve the symbols σ and c to index the classes and it is understood even where it is not explicitly stated that $1 \leq \sigma \leq p$ and $1 \leq c \leq p$.

The stream of processing requests for a particular transaction of class σ is assumed to be Poisson with rate parameter λ_σ . That is, on average every $1/\lambda_\sigma$ sec a fresh request arrives for the processing of *each* transaction $T_i^{(\sigma)}$ in class σ . Hence on average a total of

$$(3) \quad \tau_\sigma \triangleq \lambda_\sigma \binom{N}{j_\sigma + k_\sigma} \binom{j_\sigma + k_\sigma}{j_\sigma}$$

requests of class σ arrive per sec. We call $\boldsymbol{\tau} = (\tau_1, \tau_2, \dots, \tau_p)$ the *total offered traffic*.

The processing time for a transaction is assumed to be an independent random variable with an *arbitrary* but common distribution for all transactions in a class. The mean processing time for an individual transaction of class σ is $1/\mu_\sigma$ sec. Finally, we let

$$(4) \quad \rho_\sigma \triangleq \tau_\sigma / \mu_\sigma, \quad 1 \leq \sigma \leq p$$

and refer to it as the *class σ loading*. We call $\boldsymbol{\rho} = (\rho_1, \rho_2, \dots, \rho_p)$ the *loading*.

Our objective is to calculate, for the system in statistical equilibrium, the mean concurrency, the blocking probabilities and throughputs of the various classes of transactions.

1.2. Summary of results.

(i) In § 2.1 we define the states and transition rates of the Markov process which exactly reflect the physical model. The state at a particular instant of time is represented as a list of transactions undergoing processing at that time. The concurrency of the state, a p -tuple, is extracted from the state representation and it is an enumeration, by class, of the transactions undergoing concurrent processing.

(ii) In Proposition 1, § 2.1 the equilibrium distribution of the states is obtained. The product form of the equilibrium distribution is a consequence of the time-reversibility of the equilibrium Markov process.

(iii) The system concurrency is the same as the concurrency of the state of the system. Many states have concurrency equal to a particular value of the system concurrency. Proposition 2, § 2.2 enumerates the number of such states.

(iv) The equilibrium distribution of system concurrency is obtained by combining Propositions 1 and 2 and is given in Proposition 3, § 2.3. The only component of the distribution requiring substantial computation is the normalization constant, also known as the partition function.

(v) The mean concurrency, throughput and blocking probabilities for each class are performance measures of the system and these are shown to be obtained from the partition function in Proposition 4, § 2.4.

(vi) Prior to Proposition 5, § 2.5, the performance measures have been calculated for given arrival rates of individual transactions. In Proposition 5 these measures are derived for given total offered traffic, τ .

(vii) Section 3 derives the asymptotics of the performance measures for $N \rightarrow \infty$, $\rho_\sigma = O(1)$, $1 \leq \sigma \leq p$. The following intuitively appealing result is obtained:

$$\left\{ \begin{array}{l} \text{Blocking prob. of} \\ \text{class } \sigma \text{ transactions} \end{array} \right\} \sim \left\{ \begin{array}{l} \text{Number of items processed by} \\ \text{individual class } \sigma \text{ transactions} \end{array} \right\} \left\{ \begin{array}{l} \text{Weighted fraction of database} \\ \text{touched by individual transactions} \end{array} \right\}$$

$$- \left\{ \begin{array}{l} \text{Number of items non-exclusively} \\ \text{locked by individual class} \\ \sigma \text{ transactions} \end{array} \right\} \left\{ \begin{array}{l} \text{Weighted fraction of database} \\ \text{touched by non-exclusively} \\ \text{locked items} \end{array} \right\}.$$

(viii) Section 4 answers the question: what are non-exclusive locks worth? For systems with only 1 transaction class, it is shown that the fractional improvement in blocking probability due to non-exclusive locking is asymptotically equal to the square of the fraction of total locks that is non-exclusive. A related statement holds for multiple-class systems.

(ix) Numerical results derived from the exact formulas and the asymptotic formulas are obtained and compared in § 5.

For the convenience of the reader, we have listed in Appendix 1 the important variables defined in the text.

1.3. A simple example. The following example is given to illuminate the model described in § 1.1. The database contains four items, i.e. $N = 4$, and the items are indexed 1-4. There are two classes of transactions, i.e. $p = 2$: transactions in class 1 consist of two items to process with one item requiring an exclusive lock and the remaining item requiring a non-exclusive lock; transactions in class 2 consist of four items to process with 1 item requiring an exclusive lock and the remaining 3 items requiring non-exclusive locks. That is,

$$(j_\sigma, k_\sigma) = \begin{cases} (1, 1), & \sigma = 1, \\ (1, 3), & \sigma = 2. \end{cases}$$

There are consequently 12 transactions in class 1 and 4 transactions in class 2. The transactions are indexed as follows:

| | | | | | | | | | | | | | |
|-----------------------|------------------------------------|--------|--------|--------------|--------|--------|--------------|--------|--------|--------------|--------|--------|--------------|
| Transaction index: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | |
| Items in transaction: | (1; 2) | (2; 1) | (1; 3) | (3; 1) | (1; 4) | (4; 1) | (2; 3) | (3; 2) | (2; 4) | (4; 2) | (3; 4) | (4; 3) | |
| | ←----- class 1 transactions -----> | | | | | | | | | | | | |
| Transaction index: | | | | 13 | | | 14 | | | 15 | | | 16 |
| Items in transaction: | | | | (1; 2, 3, 4) | | | (2; 1, 3, 4) | | | (3; 1, 2, 4) | | | (4; 1, 2, 3) |
| | ←----- class 2 transactions -----> | | | | | | | | | | | | |

The phenomenon of interference between transactions is exemplified by the fact that transaction 13 interferes with all other transactions, and transaction 1 interferes with all transactions except transactions 8, 10, 11 and 12. The detailed nature of interactions stemming from this phenomenon is shown in the state transition diagram in Fig. 1, where, for reasons of simplicity, we have assumed that the processing times are exponentially distributed.

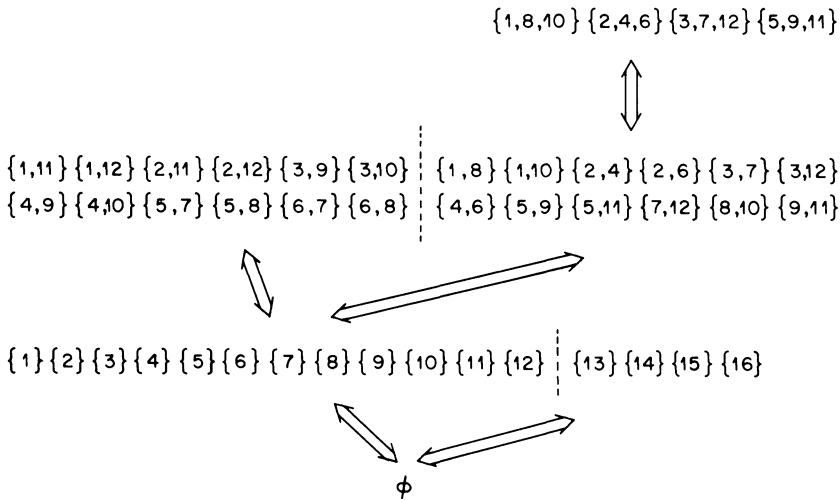


FIG. 1. State transitions in simple example of § 1.3.

In Fig. 1 we let the states of the system be specified by lists in braces of transactions under concurrent processing in the database. Thus the state $\{1, 10\}$ denotes that transactions 1 and 10 are undergoing concurrent processing. As this implies that items 1 and 4 are exclusively locked and that item 2 is non-exclusively locked, the only way in which an additional transaction may be accepted for processing is for a request for processing of transaction 8 to arrive, in which case a transition occurs, in agreement with the figure, to state $\{1, 8, 10\}$. The remaining possible transitions from state $\{1, 10\}$ are to states $\{1\}$ and $\{10\}$ which occur on completion of processing of one or the other of the pair of active transactions.

2. Equilibrium distributions, mean concurrency and other performance measures.

2.1. States, concurrency of state. In general the states are specified by lists of (nonrepeated) transactions. Thus if s is a state then it has a specification as follows:

$$(5) \quad s = \{T_{s(1)}^{(1)}, \dots, T_{s(c_1(s))}^{(1)}, T_{s(1)}^{(2)}, \dots, T_{s(c_2(s))}^{(2)}, \dots, T_{s(1)}^{(p)}, \dots, T_{s(c_p(s))}^{(p)}\}.$$

The state s is *admissible* if and only if *the exclusively locked items in the constituent transactions are mutually disjoint and the union of all exclusively locked items is disjoint from the union of all non-exclusively locked items*. We let S denote the collection of admissible states. Thus, the constituent transactions of an admissible state are non-interfering, and by the statement that the system is in state $s, s \in S$, we mean that the transactions in the specification of s have locks and are undergoing concurrent processing. *Non-admissible* states do not satisfy the above condition of non-interference among constituent transactions.

The state s in (5) corresponds to $c_1(s), c_2(s), \dots, c_p(s)$ distinct transactions of classes $1, 2, \dots, p$, respectively, being concurrently processed. It is therefore natural

to refer to

$$(6) \quad \mathbf{c}(s) \triangleq (c_1(s), c_2(s), \dots, c_p(s))$$

as the *concurrency of state s*.

Let $P(s)$ denote the equilibrium probability of state s . We claim

PROPOSITION 1.

$$(7i) \quad P(s) = \frac{\psi(s)}{G} \prod_{\sigma=1}^p (\lambda_\sigma / \mu_\sigma)^{c_\sigma(s)}$$

where G is a constant and

$$(7ii) \quad \psi(s) = \begin{cases} 1 & \text{if } s \in S, \\ 0 & \text{if } s \notin S. \end{cases}$$

Proof. The proof is based on the concept of time-reversibility in stochastic networks [9], [10]. Let s and s' be any two states with transactions lists which differ only in that the specification list of one of the states, say s' , contains an additional transaction, say of class σ . Also let $q(s, s')$ denote the transition rate from state s to state s' . Clearly,

$$(8) \quad q(s, s') = \begin{cases} \lambda_\sigma & \text{if } s \in S \text{ and } s' \in S, \\ 0 & \text{otherwise.} \end{cases}$$

Importantly, there are no transitions from inadmissible states to admissible states and, hence, the function ψ defined in (7ii) is the solution of the equation

$$(9) \quad \lambda_\sigma \frac{\psi(s')}{\psi(s)} = q(s, s').$$

The above demonstration of the existence of the function ψ allows us to invoke a theorem of Kelly's theorem [9, Thm. 3.14] to establish at once not only the product form in the equilibrium state distribution for the state dependent arrival process, but also the "insensitivity" which makes the solution valid for arbitrary distributions of the processing times. \square

While Kelly's theorem is a powerful and convenient tool for proving the result for arbitrarily distributed processing times, the reader is invited to directly verify the result for the simpler case of exponentially distributed processing times. In this case, the transition rate from state s' to state s , $s \in S$ and $s' \in S$,

$$q(s', S) = \mu_\sigma,$$

and it is straightforward to verify detailed balance [9],

$$P(s)q(s, s') = P(s')q(s', s).$$

2.2. Concurrency in the system. We are interested here in the system concurrency, which is the same as the concurrency of the state of the system. However, many states will have the same concurrency. Hence we define

$$(10) \quad \pi(\mathbf{i}) \triangleq \text{Prob} [\text{concurrency} = \mathbf{i}] = \sum_{s: \mathbf{c}(s)=\mathbf{i}} P(s).$$

That is, $\pi(\mathbf{i})$ is the equilibrium probability that i_1 transactions of class 1, i_2 transactions of class 2, \dots , i_p transactions of class p have locks and are undergoing concurrent processing.

Let \mathcal{S} denote the domain of the concurrency value \mathbf{i} in (10). Clearly, if the concurrency is \mathbf{i} then $\mathbf{i}'\mathbf{j} = i_1j_1 + i_2j_2 + \dots + i_pj_p$ items must be exclusively locked and

the minimum number of non-exclusively locked items is $\hat{k}(\mathbf{i}) = \max_{\sigma: i_{\sigma} > 0} k_{\sigma}$. Hence,

$$(11) \quad \mathcal{S} \triangleq \{\mathbf{i} \mid \mathbf{i} \text{ a } p\text{-tuple of nonnegative integers and } 0 \leq i'_j \leq N - \hat{k}(\mathbf{i})\}.$$

Combining (10) with Proposition 1 we obtain

$$(12) \quad \pi(\mathbf{i}) = \frac{C(\mathbf{i})}{G} \prod_{\sigma=1}^p (\lambda_{\sigma} / \mu_{\sigma})^{i_{\sigma}}, \quad \mathbf{i} \in \mathcal{S}$$

where

$$(13) \quad C(\mathbf{i}) \triangleq \text{cardinality of the set } \{s \in S \mid c(s) = \mathbf{i}\}, \text{ i.e., the number of admissible states with concurrency } \mathbf{i}.$$

We now claim

PROPOSITION 2.

$$(14) \quad C(\mathbf{i}) = \frac{N!}{(N - \mathbf{i}'\mathbf{j})! (\prod_{\sigma=1}^p i_{\sigma}!) \{\prod_{\sigma=1}^p (j_{\sigma}!)^{i_{\sigma}}\}} \prod_{\sigma=1}^p \binom{N - \mathbf{i}'\mathbf{j}}{k_{\sigma}}^{i_{\sigma}}, \quad \mathbf{i} \in \mathcal{S}.$$

Proof. The strings will be generated and counted in two steps: in Step 1 we will count the ways in which we can generate the exclusively locked items in the constituent transactions so that they are mutually disjoint. In Step 2 we will count the ways in which we can generate the non-exclusively locked items in the transactions from the items which are unlocked after Step 1 is complete.

Step 1. The counting is done recursively. Let

$M(l) \triangleq$ number of partitions out of N items into $(l_1 + l_2 + \dots + l_p)$ distinct, disjoint sets of distinct items in which each of l_1 sets contain j_1 items, each of l_2 sets contain j_2 items, \dots , each of l_p sets contain j_p items.

We claim that

$$(15) \quad \frac{M(l + \mathbf{e}_{\sigma})}{M(l)} = \frac{1}{(l_{\sigma} + 1)} \binom{N - l'\mathbf{j}}{j_{\sigma}}$$

where $l + \mathbf{e}_{\sigma} = (l_1, \dots, l_{\sigma} + 1, \dots, l_p)$.

To see this consider the number of ways in which we can generate partitions with an additional set containing j_{σ} items. There are $(N - l'\mathbf{j})$ items not used in any partition counted in $M(l)$. Hence there are

$$\binom{N - l'\mathbf{j}}{j_{\sigma}}$$

ways in which the items in the incremental set may be selected. Finally, the factor $1/(l_{\sigma} + 1)$ in (15) is required since the count does not take into account ordering.

The recursive relation in (15) can be solved to yield

$$(16) \quad M(\mathbf{i}) = \frac{N!}{(N - \mathbf{i}'\mathbf{j})! (\prod_{\sigma} i_{\sigma}!) \{\prod_{\sigma} (j_{\sigma}!)^{i_{\sigma}}\}}.$$

Step 2. After a partitioning performed as described in Step 1 there are $(N - \mathbf{i}'\mathbf{j})$ items which are unlocked. The non-exclusive locks for the transactions will have to be picked from this set of unlocked items and, of course, these locks may be shared. The only other restriction is that the items to be locked by each of the transactions be

distinct. Thus to each partition in Step 1 we concatenate

$$(17) \quad \prod_{\sigma=1}^p \binom{N-i'j}{k_{\sigma}}^{i_{\sigma}}$$

partitioned lists of non-exclusive locks. This concludes Step 2.

Finally, combining the results from Steps 1 and 2 we obtain

$$(18) \quad C(\mathbf{i}) = M(\mathbf{i}) \prod_{\sigma=1}^p \binom{N-i'j}{k_{\sigma}}^{i_{\sigma}}$$

as claimed in Proposition 2. \square

The reader may note that in the simple example of § 1.3 (see Fig. 1)

$$(19) \quad C(\mathbf{i}) = \begin{cases} 12, & \mathbf{i} = (1, 0), \\ 4, & \mathbf{i} = (0, 1), \\ 24, & \mathbf{i} = (2, 0), \\ 4, & \mathbf{i} = (3, 0), \end{cases}$$

which is in agreement with Proposition 2.

Equilibrium probability of system concurrency, partition function. Here we combine the expressions for $\pi(\mathbf{i})$ and $C(\mathbf{i})$ in (12) and (14) respectively and obtain forms which are convenient for further analysis.

Let us define

$$(20) \quad x_{\sigma} \triangleq \frac{\lambda_{\sigma} / \mu_{\sigma}}{j_{\sigma}! k_{\sigma}!}, \quad 1 \leq \sigma \leq p$$

and

$$(21) \quad \Phi(N, \mathbf{i}) \triangleq \frac{N!}{(N-i'j)!} \left[\prod_{\sigma=1}^p \left\{ \frac{(N-i'j)!}{(N-i'j-k_{\sigma})!} \right\}^{i_{\sigma}} \right], \quad \mathbf{i} \in \mathcal{S}.$$

From (12) and (14) it follows that

$$(22) \quad \pi(\mathbf{i}) = \frac{1}{G} \Phi(N, \mathbf{i}) \prod_{\sigma=1}^p \frac{x_{\sigma}^{i_{\sigma}}}{i_{\sigma}!}, \quad \mathbf{i} \in \mathcal{S}.$$

The constant G plays the role of a normalizing constant and henceforth we denote it by $G(N, \mathbf{x})$ and refer to it as the partition function. Hence

$$(23) \quad G(N, \mathbf{x}) = \sum_{\mathbf{i} \in \mathcal{S}} \Phi(N, \mathbf{i}) \prod_{\sigma=1}^p \frac{x_{\sigma}^{i_{\sigma}}}{i_{\sigma}!}.$$

Now observe that whenever there exists a class index σ such that $i_{\sigma} > 0$ and $N = i'j - k_{\sigma} < 0$, then $\Phi(N, \mathbf{i}) = 0$. That is, for \mathbf{i} any p -tuple of nonnegative integers,

$$(24) \quad \Phi(N, \mathbf{i}) = 0 \quad \text{if } \mathbf{i} \notin \mathcal{S}.$$

Therefore, we may rewrite (23) thus

$$(25) \quad G(N, \mathbf{x}) = \sum_{i_1=0}^{\infty} \cdots \sum_{i_p=0}^{\infty} \Phi(N, \mathbf{i}) \prod_{\sigma=1}^p \frac{x_{\sigma}^{i_{\sigma}}}{i_{\sigma}!},$$

which form we may abbreviate to

$$(26) \quad G(N, \mathbf{x}) = \sum_{\mathbf{i} \in \mathcal{Z}^p} \Phi(N, \mathbf{i}) \prod_{\sigma=1}^p \frac{x_{\sigma}^{i_{\sigma}}}{i_{\sigma}!}.$$

In summary we have

PROPOSITION 3. *The equilibrium probabilities of system concurrency are*

$$(27) \quad \pi(\mathbf{i}) = \frac{1}{G(N, \mathbf{x})} \Phi(N, \mathbf{i}) \prod_{\sigma=1}^p \frac{x_{\sigma}^{i_{\sigma}}}{i_{\sigma}!}, \quad \mathbf{i} \in \mathcal{S}$$

where x_{σ} , $1 \leq \sigma \leq p$, $\Phi(N, \mathbf{i})$ and $G(N, \mathbf{x})$ are as given in (20), (21) and (26) respectively.

2.4. Mean concurrency, throughput, blocking probabilities. Define for $1 \leq \sigma \leq p$,

$$(28) \quad \begin{aligned} \text{mean concurrency of class } \sigma &\triangleq \text{mean number of transactions of class } \sigma \\ &\text{concurrently undergoing processing,} \\ &= \sum_{\mathbf{i} \in \mathcal{S}} i_{\sigma} \pi(\mathbf{i}). \end{aligned}$$

From (27) it is easy to see that,

$$(29) \quad \text{mean concurrency of class } \sigma = \frac{x_{\sigma}}{G(N, \mathbf{x})} \frac{\partial}{\partial x_{\sigma}} G(N, \mathbf{x}).$$

For computational purposes it may also be noted that

$$(30) \quad \text{mean concurrency of class } \sigma = \frac{G_{1,\sigma}(N, \mathbf{x})}{G(N, \mathbf{x})}$$

where,

$$(31) \quad G_{1,\sigma}(N, \mathbf{x}) = \sum_{\mathbf{i} \in \mathcal{Z}^p} \Phi(N, \mathbf{i}) i_{\sigma} \prod_{c=1}^p \frac{x_c^{i_c}}{i_c!}.$$

Other important system performance measures follow directly as the following proposition notes.

PROPOSITION 4. *For $1 \leq \sigma \leq p$,*

$$(32) \quad \text{mean concurrency of class } \sigma = \frac{x_{\sigma}}{G(N, \mathbf{x})} \frac{\partial}{\partial x_{\sigma}} G(N, \mathbf{x}),$$

$$(33) \quad \begin{aligned} \text{throughput of class } \sigma \text{ transactions (trans./sec.)} \\ = \mu_{\sigma} \text{ (mean concurrency of class } \sigma \text{),} \end{aligned}$$

$$(34) \quad \begin{aligned} \text{probability of nonblocking of class } \sigma \text{ transactions} \\ = 1/\rho_{\sigma} \text{ (mean concurrency of class } \sigma \text{).} \end{aligned}$$

Equation (33) follows from (32) and an application of Little's law. Equation (34) follows from (33) and the fact that $\rho_{\sigma} = \tau_{\sigma}/\mu_{\sigma}$, where τ_{σ} (see (3)) is the total offered traffic of class σ .

2.5. Mean concurrency for given total offered traffic. Comparative performance studies frequently require isolating the effects of N , the number of items in the database. For increasing N the number of possible transactions grows as a power of N , and as $\{\lambda_{\sigma}\}$ refer to the arrival rates of requests for processing individual transactions, for fixed $\{\lambda_{\sigma}\}$ the rate of total requests for transaction processing grows similarly. It is therefore of greater interest to evaluate performance for varying N with the total offered traffic at $\tau_1, \tau_2, \dots, \tau_p$ for classes $1, 2, \dots, p$ respectively. Recall from (3) that for $1 \leq \sigma \leq p$,

$$(35) \quad \tau_{\sigma}, \text{ total offered traffic of class } \sigma \text{ (trans./sec.)} = \lambda_{\sigma} \binom{N}{j_{\sigma} + k_{\sigma}} \binom{j_{\sigma} + k_{\sigma}}{j_{\sigma}}.$$

Specifications of τ is, of course, equivalent to specification of loadings $\rho_1, \rho_2, \dots, \rho_p$ since

$$(36) \quad \rho_\sigma = \frac{\tau_\sigma}{\mu_\sigma}, \quad 1 \leq \sigma \leq p.$$

The following explicit statement is offered on account of the dependence on N of both τ and the performance measures in Proposition 4.

PROPOSITION 5. *Given total offered traffic $\tau = (\tau_1, \tau_2, \dots, \tau_p)$ or, equivalently, the loading $\rho = (\rho_1, \rho_2, \dots, \rho_p)$,*

$$(37) \quad \text{mean concurrency of class } \sigma = \frac{\rho_\sigma}{H(N, \rho)} \frac{\partial H}{\partial \rho_\sigma}(N, \rho), \quad 1 \leq \sigma \leq p,$$

where

$$(38) \quad H(N, \rho) \triangleq G(N, \mathbf{x})$$

and

$$(39) \quad \rho_\sigma = x_\sigma N! / (N - j_\sigma - k_\sigma)!, \quad 1 \leq \sigma \leq p.$$

Also,

throughput of class σ transactions (trans./sec.)
 $= \mu_\sigma$ (*mean concurrency of class σ*),
probability of nonblocking of class σ transactions
 $= 1/\rho_\sigma$ (*mean concurrency of class σ*).

The explicit expressions for $H(N, \rho)$ and $\partial H(N, \rho)/\partial \rho_\sigma$ are therefore

$$(40i) \quad H(N, \rho) = \sum_{\mathbf{i} \in \mathcal{S}} \left[\Phi(N, \mathbf{i}) \prod_{c=1}^p \left\{ \frac{(N - j_c - k_c)!}{N!} \right\}^{i_c} \right] \prod_{c=1}^p \frac{\rho_c^{i_c}}{i_c!}$$

and, for $1 \leq \sigma \leq p$,

$$(40ii) \quad \rho_\sigma \frac{\partial H}{\partial \rho_\sigma}(N, \rho) = \sum_{\mathbf{i} \in \mathcal{S}} \left[\Phi(N, \mathbf{i}) \prod_{c=1}^p \left\{ \frac{(N - j_c - k_c)!}{N!} \right\}^{i_c} \right] i_\sigma \prod_{c=1}^p \frac{\rho_c^{i_c}}{i_c!},$$

where $\Phi(N, \mathbf{i})$ is as given in (21). These expressions are important since Proposition 5 has established that all the basic performance measures of the system may be derived from it. The computational complexity for large databases, i.e. large N , is obviously formidable which provides the motivation for the asymptotic analysis in the next section.

3. Asymptotics. In this section we develop simple formulas for the mean concurrency, and thus for the other performance measures, which are asymptotically exact as $N \rightarrow \infty$, where N is the number of items in the database. The reasons are twofold. First, the asymptotic formulas yield fundamental insights into the effects of the interference phenomenon. These insights are not available from the formulas for performance measures in Propositions 4 and 5. Secondly, as we shall see in the following section, the agreement is quite good between the performance measures computed exactly and the asymptotic formulas which require very little computations.

Throughout this section we will take the point of view stated in § 2.5, i.e. the total offered traffic, or equivalently the loading, is given. Notice from (35) that the total offered traffic of class σ , τ_σ , grows like $N^{j_\sigma + k_\sigma}$ with increasing N . This observation

provides the motivation for defining

$$(41) \quad \gamma_\sigma \triangleq x_\sigma N^{j_\sigma + k_\sigma}, \quad 1 \leq \sigma \leq p.$$

We begin by developing asymptotic expansions for the partition function for the case

$$(42) \quad \gamma_\sigma = O(1), \quad 1 \leq \sigma \leq p \quad \text{as } N \rightarrow \infty.$$

Further motivation for this particular normalization comes from observing the following: the class σ loading, $\rho_\sigma \sim \gamma_\sigma$ as $N \rightarrow \infty$. That is, ρ_σ and τ_σ are of the same order of magnitude as γ_σ . (Later, in § 3.2 we will make the connection more precise.) Also, as may be intuitively expected and as is confirmed in the numerical results in § 5, it is only when (42) is true that we obtain values for the blocking probabilities that are small enough to be of practical interest.

3.1. Asymptotic formula for the partition function. Define

$$(43) \quad G(N, \boldsymbol{\gamma}) \triangleq G(N, \mathbf{x})|_{\gamma_\sigma = N^{j_\sigma + k_\sigma} x_\sigma, 1 \leq \sigma \leq p.}$$

Explicitly,

$$(44) \quad G(N, \boldsymbol{\gamma}) = \sum_{\mathbf{i} \in \mathbb{Z}^p} \frac{\Phi(N, \mathbf{i})}{N^{v(\mathbf{j}+\mathbf{k})}} \prod_{\sigma=1}^p \frac{\gamma_\sigma^{i_\sigma}}{i_\sigma!}.$$

PROPOSITION 6. As $N \rightarrow \infty$,

$$(45) \quad G(N, \boldsymbol{\gamma}) = e^{\sum \gamma_\sigma} \left[1 - \frac{\{\boldsymbol{\gamma}'(\mathbf{j}+\mathbf{k})\}^2 - (\boldsymbol{\gamma}'\mathbf{k})^2 + \sum_\sigma \gamma_\sigma (j_\sigma + k_\sigma)(j_\sigma + k_\sigma - 1)}{2N} \right] + o(1/N).$$

In the case of exclusive locking only, $\mathbf{k} = \mathbf{0}$ and the above proposition reduces to [1, Prop. 7].

The proof of Proposition 6 is in Appendix 2. It will suffice here to outline the main idea behind the estimate in (45). Let

$$(46) \quad \Psi(\mathbf{i}, 1/N) \triangleq \frac{\Phi(N, \mathbf{i})}{N^{v(\mathbf{j}+\mathbf{k})}}.$$

From (46) and the definition for $\Phi(N, \mathbf{i})$ in (21) it can be shown that

$$(47) \quad \Psi(\mathbf{i}, 1/N) = \alpha(\mathbf{i}'\mathbf{j}, 1/N) \prod_{\sigma=1}^p \{\beta_\sigma(\mathbf{i}'\mathbf{j}, 1/N)\}^{i_\sigma}$$

where $\alpha(l, 1/N)$ and $\beta_\sigma(l, 1/N)$, $1 \leq \sigma \leq p$, are functions defined for nonnegative integers l and given thus

$$(48) \quad \alpha(l, 1/N) \triangleq \begin{cases} 1 & \text{if } l = 0, \\ \prod_{m=0}^{l-1} \left(1 - \frac{m}{N}\right) & \text{if } l = 1, 2, \dots \end{cases}$$

and

$$(49) \quad \beta_\sigma(l, 1/N) = \begin{cases} 1 & \text{if } l = 0, \\ \prod_{m=l}^{l+k_\sigma-1} \left(1 - \frac{m}{N}\right) & \text{if } l = 1, 2, \dots \end{cases}$$

To recapitulate

$$(50) \quad G(N, \boldsymbol{\gamma}) = \sum_{\mathbf{i} \in \mathbb{Z}^p} \Psi(\mathbf{i}, 1/N) \prod_{\sigma=1}^p \frac{\gamma_\sigma^{i_\sigma}}{i_\sigma!}$$

where $\Psi(\cdot, \cdot)$ is constituted from the more elementary functions $\alpha(\cdot, \cdot)$ and $\beta_\sigma(\cdot, \cdot)$, $1 \leq \sigma \leq p$. The reason for making this transformation is that $\alpha(\cdot, \cdot)$ and $\beta_\sigma(\cdot, \cdot)$, and therefore $\Psi(\cdot, \cdot)$, are in forms convenient for expansions in powers of $1/N$.

The leading two terms in such an expansion of $\Psi(\cdot, \cdot)$ have been obtained and these are as follows:

$$(51) \quad \Psi(\mathbf{i}, 1/N) = 1 - \frac{\{\mathbf{i}'(\mathbf{j} + \mathbf{k})\}^2 - (\mathbf{i}'\mathbf{k})^2 + \sum_\sigma i_\sigma(k_\sigma^2 - j_\sigma - k_\sigma)}{2N} + r(\mathbf{i}, 1/N).$$

An estimate of the remainder term $r(\mathbf{i}, 1/N)$ is, of course, crucial and this is derived in Appendix 2. Here it will suffice to state that the idea behind the result in Proposition 6 is the replacement of $\Psi(\cdot, \cdot)$ in (50) by its estimate given by the leading two terms on the right-hand side of (51). On summing in (50) with respect to \mathbf{i} , the estimate of the partition function in (45) is obtained. To see this observe that

$$(52) \quad \begin{aligned} & \{\mathbf{i}(\mathbf{j} + \mathbf{k})\}^2 - (\mathbf{i}'\mathbf{k})^2 + \sum_\sigma i_\sigma(k_\sigma^2 - j_\sigma - k_\sigma) \\ &= \sum_\sigma \{(j_\sigma + k_\sigma)^2 - k_\sigma^2\} i_\sigma(i_\sigma - 1) + \sum_{\sigma_1 \neq \sigma_2} \{(j_{\sigma_1} + k_{\sigma_1})(j_{\sigma_2} + k_{\sigma_2}) - k_{\sigma_1}k_{\sigma_2}\} i_{\sigma_1}i_{\sigma_2} \\ & \quad + \sum_\sigma i_\sigma(j_\sigma + k_\sigma)(j_\sigma + k_\sigma - 1). \end{aligned}$$

The reader may verify that on using (52),

$$(53) \quad \begin{aligned} & \sum_{\mathbf{i} \in \mathbb{Z}^p} \left[1 - \frac{\{\mathbf{i}'(\mathbf{j} + \mathbf{k})\}^2 - (\mathbf{i}'\mathbf{k})^2 + \sum_\sigma i_\sigma(k_\sigma^2 - j_\sigma - k_\sigma)}{2N} \right] \prod_{\sigma=1}^p \frac{\gamma_\sigma^{i_\sigma}}{i_\sigma!} \\ &= e^{\sum \gamma_\sigma} \left[1 - \frac{\{\boldsymbol{\gamma}'(\mathbf{j} + \mathbf{k})\}^2 - (\boldsymbol{\gamma}'\mathbf{k})^2 + \sum \gamma_\sigma(j_\sigma + k_\sigma)(j_\sigma + k_\sigma - 1)}{2N} \right], \end{aligned}$$

the estimate in Proposition 6.

3.2. Asymptotic formulas for mean concurrency and other performance measures. Here we combine the two results derived above, namely, Propositions 5 and 6, to obtain an asymptotically exact formula as $N \rightarrow \infty$ for the mean concurrency given the total offered traffic or, equivalently, the loading. The main step involved in the derivation of the new result is the replacement of the parameters $\{\gamma_\sigma\}$ in Proposition 6 by estimates in terms of the given loading parameters $\{\rho_\sigma\}$ which are correct up to the appropriate order of magnitude.

Observe from (39) and (41) that

$$(54) \quad \rho_\sigma = \begin{cases} \gamma_\sigma \frac{N!}{(N - j_\sigma - k_\sigma)! N^{j_\sigma + k_\sigma}}, & 1 \leq \sigma \leq p, \end{cases}$$

$$(55) \quad \rho_\sigma = \begin{cases} \gamma_\sigma \prod_{l=1}^{j_\sigma + k_\sigma - 1} \left(1 - \frac{l}{N} \right), & 1 \leq \sigma \leq p. \end{cases}$$

In particular, therefore, as $N \rightarrow \infty$

$$(56) \quad \gamma_\sigma = \rho_\sigma + O\left(\frac{1}{N}\right), \quad 1 \leq \sigma \leq p,$$

and,

$$(57) \quad \exp(\sum \gamma_\sigma) = \exp(\sum \rho_\sigma) \left[1 + \frac{1}{2N} \sum \rho_\sigma(j_\sigma + k_\sigma)(j_\sigma + k_\sigma - 1) \right] + O(1/N^2).$$

Also recall from (38), (39) and (43) that

$$H(N, \rho) = G(N, \gamma)$$

where ρ and γ are related as given in (54). Hence, we may use the expressions (56) and (57) in the asymptotic formula for $G(N, \gamma)$ derived in Proposition 6 to obtain

$$(58) \quad H(N, \rho) = \exp(\sum \rho_\sigma) \left[1 - \frac{\{\rho'(j+k)\}^2 - (\rho'k)^2}{2N} \right] + o\left(\frac{1}{N}\right),$$

as $N \rightarrow \infty$.

It can similarly be shown that

$$(59) \quad \frac{\partial H}{\partial \rho_\sigma}(N, \rho) = \exp(\sum \rho_c) \left[1 - \frac{\{\rho'(j+k)\}^2 - (\rho'k)^2}{2N} - \frac{1}{N} \{\rho'(j+k)(j_\sigma + k_\sigma) - (\rho'k)k_\sigma\} \right] + o\left(\frac{1}{N}\right), \quad 1 \leq \sigma \leq p.$$

Combining (58) and (59) gives

$$(60) \quad \frac{1}{H(N, \rho)} \frac{\partial}{\partial \rho_\sigma} H(N, \rho) = 1 - \frac{(j_\sigma + k_\sigma)\{\rho'(j+k)\} - k_\sigma(\rho'k)}{N} + o\left(\frac{1}{N}\right),$$

as $N \rightarrow \infty$.

We are now in a position to appeal to Proposition 5 to obtain the asymptotic formulas for the performance measures and the result is summarized below.

PROPOSITION 7. *Given total offered traffic $\tau = (\tau_1, \tau_2, \dots, \tau_p)$ or, equivalently, the loading $\rho = (\rho_1, \rho_2, \dots, \rho_p)$, as $N \rightarrow \infty$,*

mean concurrency of class σ

$$(61) \quad = \rho_\sigma \left[1 - \frac{1}{N} \left\{ (j_\sigma + k_\sigma) \sum_c \rho_c(j_c + k_c) - k_\sigma \sum_c \rho_c k_c \right\} \right] + o\left(\frac{1}{N}\right), \quad 1 \leq \sigma \leq p,$$

throughput of class σ transactions

$$(62) \quad = \tau_\sigma \left[1 - \frac{1}{N} \left\{ (j_\sigma + k_\sigma) \sum_c \rho_c(j_c + k_c) - k_\sigma \sum_c \rho_c k_c \right\} \right] + o\left(\frac{1}{N}\right), \quad 1 \leq \sigma \leq p,$$

probability of nonblocking of class σ transactions

$$(63) \quad = 1 - \frac{1}{N} \left\{ (j_\sigma + k_\sigma) \sum_c \rho_c(j_c + k_c) - k_\sigma \sum_c \rho_c k_c \right\} + o\left(\frac{1}{N}\right), \quad 1 \leq \sigma \leq p.$$

3.3. Discussion. The formulas in Proposition 7 are amenable to interpretations which are intuitively appealing. The formula in (63) states that for large databases with relatively low blocking,

$$(64) \quad \left\{ \begin{array}{l} \text{blocking prob. of} \\ \text{class } \sigma \text{ transactions} \end{array} \right\} \approx \left\{ \begin{array}{l} \text{number of items locked by} \\ \text{individual class } \sigma \text{ transactions} \end{array} \right\} \left\{ \begin{array}{l} \text{weighted fraction of database} \\ \text{touched by individual transactions} \end{array} \right\} - \left\{ \begin{array}{l} \text{number of items non-exclusively} \\ \text{locked by individual class} \\ \sigma \text{ transactions} \end{array} \right\} \left\{ \begin{array}{l} \text{weighted fraction of database} \\ \text{touched by non-exclusively} \\ \text{locked items} \end{array} \right\}.$$

In the above interpretation the ‘‘weights’’ are the loading parameters $\{\rho_c\}$. Thus, $(j_c + k_c)/N$ and k_c/N are the fractions of the database touched respectively by locks

of both kinds and by the non-exclusive locks only in each class c transaction. Hence, weighted fraction of database touched by individual transactions $= \sum_c \rho_c(j_c + k_c)/N$, weighted fraction of database touched by non-exclusively locked items $= \sum_c \rho_c k_c/N$.

Numerical studies indicate that the above formula is a good approximation for blocking probability up to 0.2.

When $k_c = 0, 1 \leq c \leq p$, in the formulas given in Proposition 7, all transactions require only exclusive locks and [1, Prop. 8] is recovered.

4. What are non-exclusive locks worth? The performance of the system is now compared with that of a system in which all locks are exclusive. In the latter system each transaction of class σ processes $(j_\sigma + k_\sigma)$ items, all of which are exclusively locked. The asymptotic performance for the system with exclusive locks only is obtained directly from Proposition 7. Specifically, we let the parameters of the system be

$$(65) \quad j'_\sigma = (j_\sigma + k_\sigma), \quad k'_\sigma = 0, \quad 1 \leq \sigma \leq p.$$

There are

$$\binom{N}{j_\sigma + k_\sigma} \binom{j_\sigma + k_\sigma}{j_\sigma}$$

transactions in class σ in the system with non-exclusive locks, while there are only

$$\binom{N}{j_\sigma + k_\sigma}$$

transactions in class σ in the system with only exclusive locks. (This is as it should be since each $(j_\sigma + k_\sigma)$ -exclusively locked items in the latter system has

$$\binom{j_\sigma + k_\sigma}{j_\sigma}$$

partitions into j_σ exclusively locked items and k_σ non-exclusively locked items in the former system.) The machinery for taking the above fact into account obviously exists, i.e. by having equal total offered traffic in the two systems.

To summarize,

Performance measures of a comparable system with only exclusive locks:

$$(66) \quad \text{mean concurrency of class } \sigma = \rho_\sigma \left[1 - \frac{1}{N} \left\{ (j_\sigma + k_\sigma) \sum_c \rho_c(j_c + k_c) \right\} \right] + o\left(\frac{1}{N}\right),$$

$$(67) \quad \text{throughput of class } \sigma \text{ transactions}$$

$$= \tau_\sigma \left[1 - \frac{1}{N} \left\{ (j_\sigma + k_\sigma) \sum_c \rho_c(j_c + k_c) \right\} \right] + o\left(\frac{1}{N}\right),$$

$$(68) \quad \text{probability of nonblocking of class } \sigma \text{ transactions}$$

$$= 1 - \frac{1}{N} \left\{ (j_\sigma + k_\sigma) \sum_c \rho_c(j_c + k_c) \right\} + o\left(\frac{1}{N}\right).$$

Comparison of the performance estimates in Proposition 7 with the above gives the estimated worth of non-exclusive locking. A succinct statement derived from such a comparison is

PROPOSITION 8.

$$(69) \quad \frac{B_{\sigma,E} - B_{\sigma,NE}}{B_{\sigma,E}} \sim \frac{k_{\sigma}}{j_{\sigma} + k_{\sigma}} \frac{\sum_c \rho_c k_c}{\sum_c \rho_c (j_c + k_c)}, \quad 1 \leq \sigma \leq p,$$

where $B_{\sigma,E}$ and $B_{\sigma,NE}$ are the blocking probabilities for class σ transactions in, respectively, the system with only exclusive locks and the system with non-exclusive locks.

When there is only one class of transactions the following is true: the fractional improvement in blocking probability due to non-exclusive locks is asymptotically equal to the square of the fraction of total locks that is non-exclusive.

5. Numerical results. Numerical results are presented in Figs. 2-5. All results are for the case of a single transaction class. The performance measure shown in all the figures is the probability of nonblocking, from which other performance measures may be readily computed as Proposition 5 in § 2.5 states. Also, in all cases the measure of offered traffic is the "total offered traffic," τ . As we have stated before, meaningful performance comparisons of databases of different sizes and of different transaction compositions is possible by this mechanism. For instance, in the case of different database sizes, the alternative of comparing performance for the same arrival rate of individual transactions is not meaningful since larger databases have more constituent transactions.

Figures 2-4 give exact results which have been computed by using the formula in Proposition 5. These results, therefore, have been obtained by calculating the

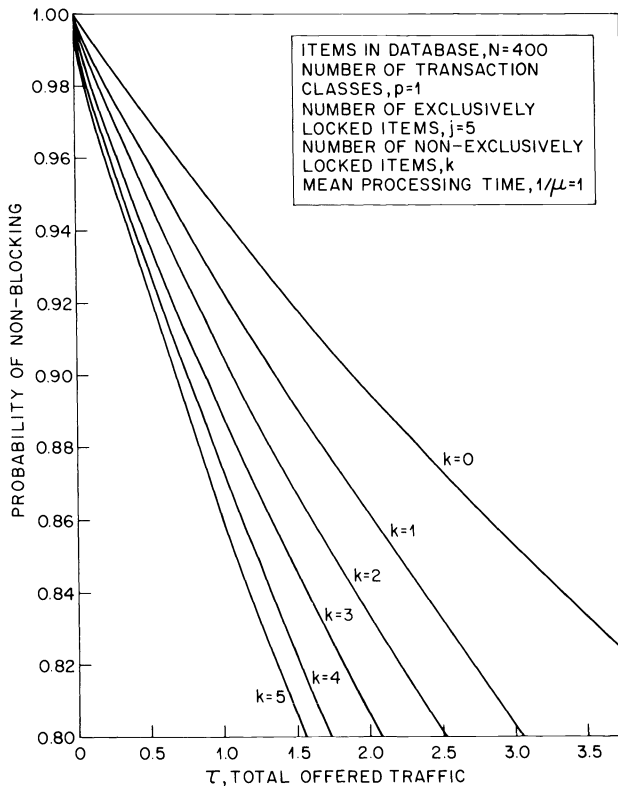


FIG. 2. Effect of the number of non-exclusively locked items.

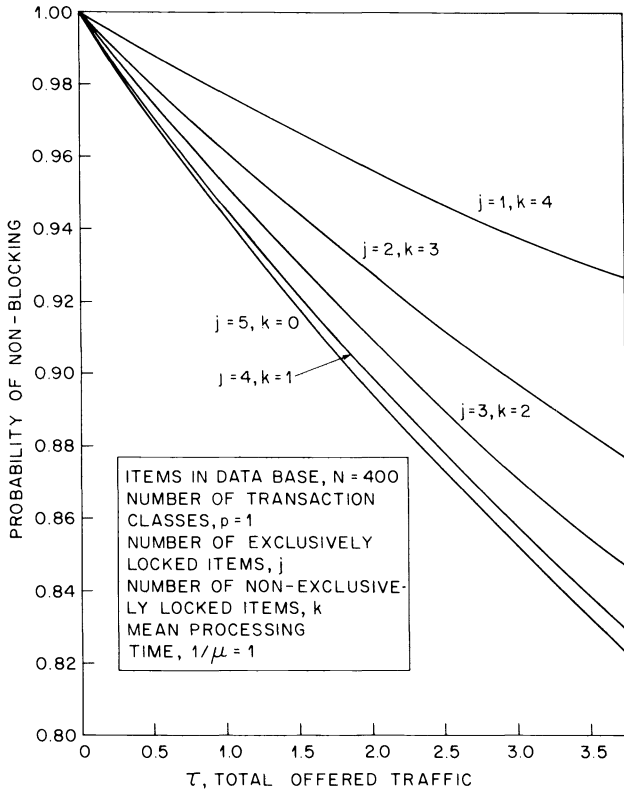


FIG. 3. Effect of the mix of exclusive and non-exclusive locks.

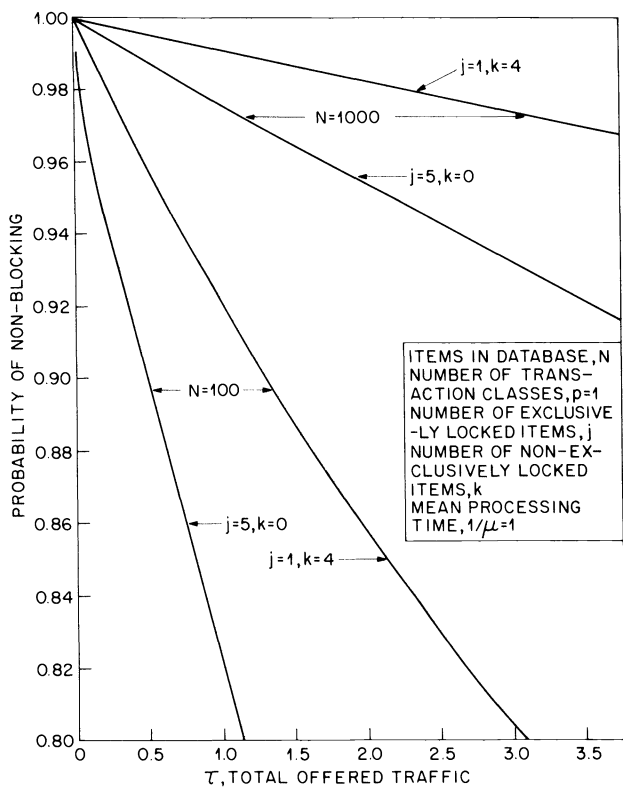


FIG. 4. Effect of the database size.

expressions in (40). Figure 5 contrasts results obtained from the exact formula with corresponding results obtained from the asymptotic formula in Proposition 7 in § 3.2.

Figure 2 shows, for a fixed database size, the effect on performance of increasing the number of non-exclusively locked items. Figure 3 shows, for the same fixed database size, the effect on performance of different proportions of exclusively locked and non-exclusively locked items in transactions, with the total number of locked transactions in each transaction held fixed. Figure 4 shows the effect on performance of database size for two fixed transaction specifications.

In Fig. 5 observe that the agreement between exact and asymptotic results is better for larger databases and for smaller values of the total offered traffic, which is as expected.

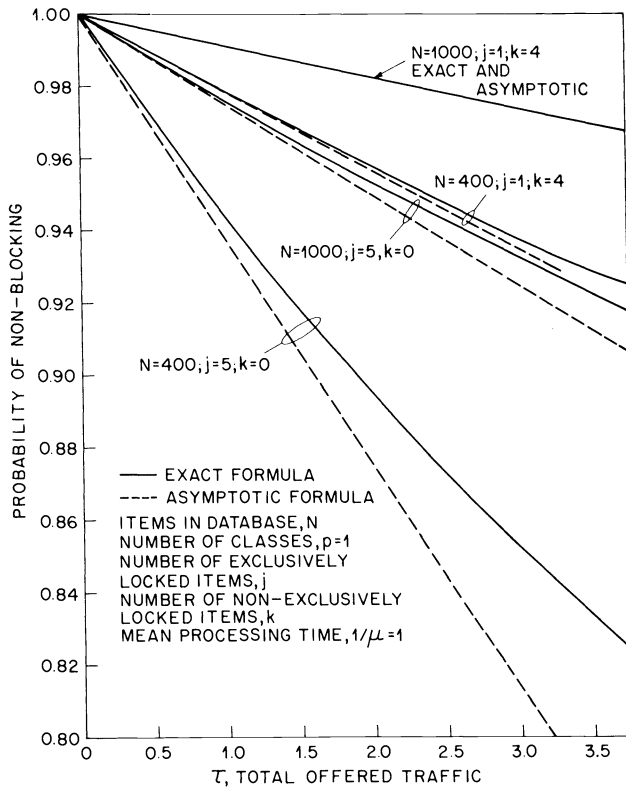


FIG. 5. Comparison of exact and asymptotic formulas.

Appendix 1. Glossary.

- N = number of items in database
- p = number of classes of transactions
- σ, c = indices of classes of transactions
- j_σ = number of exclusively locked items in each transaction of class σ
- k_σ = number of non-exclusively locked items in each transaction of class σ
- λ_σ = arrival rate (1/secs) of requests for processing of each transaction of class σ
- $1/\mu_\sigma$ = mean processing time (secs) for each transaction in class σ
- $x_\sigma = (\lambda_\sigma / \mu_\sigma) / (j_\sigma! k_\sigma!)$

$$T_i^{(\sigma)} = \text{a transaction in class } \sigma; 1 \leq i \leq \binom{N}{j_\sigma + k_\sigma} \binom{j_\sigma + k_\sigma}{j_\sigma}$$

- $s, \text{ state} = \{T_{s(l)}^{(\sigma)}, 1 \leq \sigma \leq p, 1 \leq l \leq s(c_\sigma(s))\}$
- $S = \text{collection of admissible states}$
- $\mathbf{c}(s) = (c_1(s), c_2(s), \dots, c_p(s)), \text{ concurrency of state } s$
- $P(s) = \text{equilibrium probability of state } s$
- $\pi(\mathbf{i}) = \text{prob.}[\text{concurrency} = \mathbf{i}] = \sum_{s: \mathbf{c}(s) = \mathbf{i}} P(s)$
- $C(\mathbf{i}) = \text{cardinality of } \{s | \mathbf{c}(s) = \mathbf{i}\} = \text{number of states with concurrency } \mathbf{i}$
- $\mathcal{S} = \{i | 0 \leq i' \leq N - k_\sigma, \text{ for all } \sigma \text{ such that } i_\sigma > 0\}$
- $G(N, \mathbf{x}) = \text{partition function}$
- $\Phi(N, \mathbf{i}) = \text{constituent in partition function}$
- $Z^p = \text{set of } p\text{-tuples of non-negative integers}$
- $\tau_\sigma = \text{total offered traffic of class } \sigma$
- $\rho_\sigma = \text{loading of class } \sigma$
- $\gamma_\sigma = N^{j_\sigma + k_\sigma} x_\sigma$
- $G(N, \boldsymbol{\gamma}) = G(N, \mathbf{x})$
- $H(N, \boldsymbol{\rho}) = G(N, \mathbf{x})$
- $j_\sigma = \max_{\sigma} j_\sigma$
- $k = \max_{\sigma} k_\sigma$
- $w = 1/N$

Appendix 2. Proof of Proposition 6. The following two lemmas will prove quite useful in estimating error terms in the asymptotic formula.

LEMMA 1. For any $I \geq 1$,

$$(A2.1) \quad \sum_{i' \geq I} \prod_{\sigma} \frac{\gamma_\sigma^{i'_\sigma}}{i'_\sigma!} = \frac{\exp(\sum \gamma_\sigma)}{(I-1)!} \int_0^{\sum \gamma_\sigma} e^{-w} w^{I-1} dw$$

$$(A2.2) \quad \leq \frac{\exp(\sum \gamma_\sigma)}{I!} (\sum \gamma_\sigma)^I.$$

As Lemma 1 has been proven in [1], we omit the proof. Observe that the integral in (A2.1) is the incomplete gamma function [11].

LEMMA 2. For any $I \geq 3$,

$$(A2.3) \quad \sum_{i' \geq I} \left[\{i'(j+k)\}^2 - (i'k)^2 + \sum_{\sigma} i_\sigma (k_\sigma^2 - j_\sigma - k_\sigma) \right] \prod_{\sigma} \frac{\gamma_\sigma^{i'_\sigma}}{i'_\sigma!}$$

$$(A2.4) \quad = \exp(\sum \gamma_\sigma) \left[\left[\{\boldsymbol{\gamma}'(j+k)\}^2 - (\boldsymbol{\gamma}'k)^2 \right] \frac{\int_0^{\sum \gamma_\sigma} e^{-w} w^{I-3} dw}{(I-3)!} \right. \\ \left. + \sum_{\sigma} \gamma_\sigma (j_\sigma + k_\sigma)(j_\sigma + k_\sigma - 1) \frac{\int_0^{\sum \gamma_\sigma} e^{-w} w^{I-2} dw}{(I-2)!} \right]$$

$$(A2.5) \quad \leq \exp(\sum \gamma_\sigma) \left[\left[\{\boldsymbol{\gamma}'(j+k)\}^2 - (\boldsymbol{\gamma}'k)^2 \right] \frac{(\mathbf{1}'\boldsymbol{\gamma})^{I-2}}{(I-2)!} \right. \\ \left. + \left[\sum_{\sigma} \gamma_\sigma (j_\sigma + k_\sigma)(j_\sigma + k_\sigma - 1) \right] \frac{(\mathbf{1}'\boldsymbol{\gamma})^{I-1}}{(I-1)!} \right].$$

Proof. Let the quantity in (A2.3) to be estimated be given by S and also let

$$(A2.6) \quad l_\sigma \triangleq j_\sigma + k_\sigma, \quad 1 \leq \sigma \leq p.$$

On using (52) we have

$$(A2.7) \quad S = \sum_{\sigma} (l_\sigma^2 - k_\sigma^2) \left[\sum_{i' \geq I} i_\sigma (i_\sigma - 1) \prod_c \frac{\gamma_c^{i'_c}}{i'_c!} \right] + \sum_{\sigma_1 \neq \sigma_2} (l_{\sigma_1} l_{\sigma_2} - k_{\sigma_1} k_{\sigma_2}) \left[\sum_{i' \geq I} i_{\sigma_1} i_{\sigma_2} \prod_c \frac{\gamma_c^{i'_c}}{i'_c!} \right] \\ + \sum_{\sigma} l_\sigma (l_\sigma - 1) \left[\sum_{i' \geq I} i_\sigma \prod_c \frac{\gamma_c^{i'_c}}{i'_c!} \right]$$

$$\begin{aligned}
 &= \left\{ \sum_{\sigma} (l_{\sigma}^2 - k_{\sigma}^2) \gamma_{\sigma}^2 + \sum_{\sigma_1 \neq \sigma_2} (l_{\sigma_1} l_{\sigma_2} - k_{\sigma_1} k_{\sigma_2}) \gamma_{\sigma_1} \gamma_{\sigma_2} \right\} \sum_{i'1 \geq I-2} \prod_c \frac{\gamma_c^i}{i_c!} \\
 &\quad + \left\{ \sum_{\sigma} l_{\sigma} (l_{\sigma} - 1) \gamma_{\sigma} \right\} \sum_{i'1 \geq I-1} \prod_c \frac{\gamma_c^i}{i_c!} \\
 &= \left\{ \left(\sum_{\sigma} l_{\sigma} \gamma_{\sigma} \right)^2 - \left(\sum_{\sigma} k_{\sigma} \gamma_{\sigma} \right)^2 \right\} \sum_{i'1 \geq I-2} \prod_c \frac{\gamma_c^i}{i_c!} + \left\{ \sum_{\sigma} l_{\sigma} (l_{\sigma} - 1) \gamma_{\sigma} \right\} \sum_{i'1 \geq I-1} \prod_c \frac{\gamma_c^i}{i_c!}.
 \end{aligned}$$

On now using Lemma 1, the statement made in (A2.4) follows. The statement in (A2.5) is a simple corollary obtained by observing $e^{-w} \leq 1$ for all $w \geq 0$. This concludes the proof of Lemma 2. \square

We are now ready to provide the proof of Proposition 6. We shall make the important selection

$$(A2.8) \quad I = \log_e N$$

and split the form for $G(N, \gamma)$ in (50) thus

$$(A2.9) \quad G(N, \gamma) = \sum_{i'1 \leq I-1} \Psi(i, 1/N) \prod_{\sigma} \frac{\gamma_{\sigma}^i}{i_{\sigma}!} + \sum_{i'1 \geq I} \Psi(i, 1/N) \prod_{\sigma} \frac{\gamma_{\sigma}^i}{i_{\sigma}!}.$$

On using the expression for $\Psi(i, 1/N)$ in (51) we obtain

$$(A2.10) \quad G(N, \gamma) = \sum_{i \in Z^p} \left[1 - \frac{\{i'(j+k)\}^2 - (i'k)^2 + \sum i_{\sigma} (k_{\sigma}^2 - j_{\sigma} - k_{\sigma})}{2N} \right] \prod_{\sigma=1}^p \frac{\gamma_{\sigma}^i}{i_{\sigma}!} + \varepsilon_1(N) + \varepsilon_2(N) + \varepsilon_3(N),$$

where

$$(A2.11) \quad \varepsilon_1(N) \triangleq - \sum_{i'1 \geq I} \left[1 - \frac{\{i'(j+k)\}^2 - (i'k)^2 + \sum i_{\sigma} (k_{\sigma}^2 - j_{\sigma} - k_{\sigma})}{2N} \right] \prod_{\sigma} \frac{\gamma_{\sigma}^i}{i_{\sigma}!},$$

$$(A2.12) \quad \varepsilon_2(N) \triangleq \sum_{i'1 \geq I} \Psi(i, 1/N) \prod_{\sigma} \frac{\gamma_{\sigma}^i}{i_{\sigma}!},$$

$$(A2.13) \quad \varepsilon_3(N) \triangleq \sum_{i'1 \geq I-1} r(i, 1/N) \prod_{\sigma} \frac{\gamma_{\sigma}^i}{i_{\sigma}!}.$$

Now, the leading term in the right-hand side of (A2.10) has already been calculated and it has been given in (53). The proof of Proposition 6 will therefore consist of showing that $\varepsilon_l(N) = o(1/N)$ as $N \rightarrow \infty$ for $l = 1, 2, 3$.

For $l = 1, 2$ the proof follows easily from Lemmas 1 and 2 and the fact that $\Psi(i, 1/N) \leq 1, i \in Z^p$. For instance, using Lemma 1 it follows that

$$(A2.14) \quad N \sum_{i'1 \geq I} \prod_{\sigma=1}^p \frac{\gamma_{\sigma}^i}{i_{\sigma}!} \leq e^{\sum \gamma_{\sigma}} \left[\frac{\{e(\mathbf{1}'\gamma)\}^I}{I!} \right] \rightarrow 0$$

as N (and therefore I) $\rightarrow \infty$. Hence $\varepsilon_2(N) = o(1/N)$.

It remains to show that $\varepsilon_3(N) = o(1/N)$. To do this we first need to estimate $r(i, 1/N)$. Let us write

$$(A2.15) \quad w = \frac{1}{N},$$

so that

$$(A2.16) \quad r(\mathbf{i}, w) = \frac{w^2}{2} \frac{\partial^2}{\partial w^2} \Phi(\mathbf{i}, w) \Big|_{w=a} \quad \text{for some } a \in [0, w].$$

Now it can be shown that

$$(A2.17) \quad \frac{1}{\Psi(\mathbf{i}, w)} \frac{\partial^2 \Psi(\mathbf{i}, w)}{\partial w^2} = \left\{ \sum_{l=0}^{i_j-1} \frac{l}{1-lw} + \sum_{\sigma} i_{\sigma} \sum_{l=i_j}^{i_j+k_{\sigma}-1} \frac{l}{1-lw} \right\}^2 - \left\{ \sum_{l=0}^{i_j-1} \frac{l^2}{(1-lw)^2} + \sum_{\sigma} i_{\sigma} \sum_{l=i_j}^{i_j+k_{\sigma}-1} \left(\frac{l}{1-lw} \right)^2 \right\}.$$

We shall assume that N is large enough that

$$(A2.18) \quad w < \frac{1}{\hat{j}(\hat{\mathbf{i}}\mathbf{1}) + \hat{k}}$$

where, $\hat{j} = \max_{\sigma} j_{\sigma}$ and $\hat{k} = \max_{\sigma} k_{\sigma}$, and thus avoid the singularities of the right-hand side of (A2.17). Noting that $\Psi(\mathbf{i}, w) < 1$ it follows that

$$0 < \frac{\partial^2}{\partial w^2} \Psi(\mathbf{i}, w) \Big|_{w=a} < \left\{ \sum_{l=0}^{i_j-1} \frac{l}{1-lw} + \sum_{\sigma} i_{\sigma} \sum_{l=i_j}^{i_j+k_{\sigma}-1} \frac{l}{1-lw} \right\}^2, \quad \forall a \in [0, w]$$

$$< \frac{1}{4\{1 - (\hat{j}\hat{\mathbf{i}}\mathbf{1} + \hat{k})w\}^2} [(\hat{\mathbf{i}}\mathbf{j})^2 + 2(\hat{\mathbf{i}}\mathbf{j})(\hat{\mathbf{i}}\mathbf{k}) + \hat{k}(\hat{\mathbf{i}}\mathbf{k})]^2.$$

Hence,

$$(A2.19) \quad 0 < r(\mathbf{i}, w) < \frac{w^2}{8\{1 - (\hat{j}\hat{\mathbf{i}}\mathbf{1} + \hat{k})w\}^2} [(\hat{\mathbf{i}}\mathbf{j})^2 + 2(\hat{\mathbf{i}}\mathbf{j})(\hat{\mathbf{i}}\mathbf{k}) + \hat{k}(\hat{\mathbf{i}}\mathbf{k})]^2.$$

We are now in a position to bound $\epsilon_3(N)$ in (A2.13).

$$(A2.20) \quad \epsilon_3(N) \leq \frac{1}{8N^2\{1 - (\hat{J}I + \hat{k})/N\}^2} \sum_{i_1 \leq I-1} \{(\hat{\mathbf{i}}\mathbf{j})^2 + 2(\hat{\mathbf{i}}\mathbf{j})(\hat{\mathbf{i}}\mathbf{k}) + \hat{k}(\hat{\mathbf{i}}\mathbf{k})\}^2 \prod_{\sigma} \frac{\gamma_{\sigma}^{i_{\sigma}}}{i_{\sigma}!}.$$

The assumption in (A2.18) is implied by

$$(A2.21) \quad \hat{j} \log N + \hat{k} < N$$

which may be assumed without loss of generality. Taking note of the selection $I = \log_e N$, it follows straightforwardly that

$$(A2.22) \quad N\epsilon_3(N) \rightarrow 0 \quad \text{as } N \text{ (and therefore } I) \rightarrow \infty.$$

This concludes the proof. \square

Acknowledgments. We are grateful to a colleague Ming Lai for suggesting that we investigate the worth of non-exclusive locks. We are also indebted to Peter Weinberger for the carry-over of the benefits of an earlier collaboration. Finally, we are especially grateful to F. P. Kelly of Cambridge University for the benefit of discussions concerning the insensitivity property discussed in § 2.1.

REFERENCES

[1] D. MITRA AND P. J. WEINBERGER, *Probabilistic models of database locking: solutions, computational algorithms and asymptotics*, J. Assoc. Comput. Mach., 31 (1984), pp. 855-878.

- [2] D. POTIER AND P. L. LEBLANC, *Analysis of locking policies in database management systems*, Comm. ACM, 23 (1980).
- [3] A. W. SHUM AND P. G. SPIRAKIS, *Performance analysis of concurrency control methods in database systems*, Performance '81, F. J. Kylstra ed., North-Holland, Amsterdam, 1981, pp. 1-19.
- [4] N. GOODMAN, R. SURI AND Y. C. TAY, *A simple analytic model for performance of exclusive locking in database systems*, Proc. ACM Symposium on Principles of Database Systems, Atlanta, 1983, pp. 203-215.
- [5] D. A. MENASCE AND T. NAKANISHI, *Performance evaluation of a two-phase commit based protocols for DDBs*, Proc. ACM Symposium on Principles of Database Systems, Los Angeles, 1982, pp. 247-255.
- [6] A. CHESNAIS, E. GELEMBE AND I. MITRANI, *On the modeling of parallel access to shared data*, Comm. ACM, 26 (March 1983), pp. 196-202.
- [7] D. R. RIES AND M. R. STONEBRAKER, *Locking granularity revisited*, ACM Trans. Database Systems, 4 (1979), pp. 210-227.
- [8] J. D. ULLMAN, *Principles of Database Systems*, 2nd ed., Computer Science Press, Rockville, MD, 1982.
- [9] F. P. KELLY, *Reversibility and Stochastic Networks*, John Wiley, New York, 1980.
- [10] S. S. LAM, *Queueing networks with population size constraints*, IBM J. Res. and Dev., 21 (1976), pp. 394-403.
- [11] F. W. J. OLVER, *Introduction to Asymptotics and Special Functions*, Academic Press, New York, 1974.

THE KNUTH-BENDIX COMPLETION PROCEDURE AND THUE SYSTEMS*

DEEPAK KAPUR[†] AND PALIATH NARENDRAN[†]

Abstract. The Knuth-Bendix completion procedure for term rewriting systems in many cases provides a decision procedure for equational theories and has been found to have many applications in various areas. We discuss the application of the Knuth-Bendix procedure to Thue systems. We use the notion of a reduced Thue system and show that for every Church-Rosser Thue system, there is a unique reduced Church-Rosser Thue system equivalent to it. Furthermore, the Knuth-Bendix completion procedure, when applied to a Thue system T , always produces the finite reduced Church-Rosser Thue system equivalent to T whenever such a system exists. Similar results can also be proved for almost-confluent Thue systems. Using properties of reduced Church-Rosser systems, we develop conditions under which a class of special Thue systems have equivalent finite Church-Rosser systems. In addition, we show that the completion procedure always terminates on finite parenthesized Thue systems, from which the termination of the completion procedure over ground-term-rewriting systems can be shown immediately. From the results discussed in this paper, we also obtain the termination of the Knuth-Bendix completion procedure for commutative Thue systems (commutative monoids) as a simple corollary.

Key Words. Thue systems, Knuth-Bendix completion procedure, Church-Rosser systems, rewriting systems, almost-confluent systems, ground terms, commutative monoids

1. Introduction. There has been considerable interest recently in rewriting or transformation systems because of their applications to theorem proving, reasoning about specifications and programs, abstract data types, program transformation and synthesis, algebraic simplification, etc. [5],[7],[16],[17],[22],[24]. The main reason for this interest is the usefulness of the *Church-Rosser property*, which, in general, can be interpreted as implying that the order of applications of transformations makes no difference. Along with the *uniform termination property*, which ensures that every sequence of transformations eventually reaches a result that cannot be further transformed (called a “normal form”), we get a decision procedure for the equational theory induced by the transformations. Note that now any sequence of transformations on an object that produces a normal form will do; the Church-Rosser property ensures that all possible sequences of transformations on the object would produce the same normal form. Transformation systems that have both uniform termination and Church-Rosser properties are called *canonical* systems.

In most cases, however, the systems of transformations or rules that arise may not be canonical. This is where the notion of a completion procedure comes into play; we can attempt to get a canonical set of rules by adding and/or deleting existing rules without altering the underlying theory. Knuth and Bendix [17] introduced a completion procedure for term-rewriting systems in which objects under consideration are (first-order) terms or expressions and transformations are

*Received by the editors October 24, 1983, and in revised form October 1, 1984. A preliminary version of this paper appeared in the *Third Conference on Foundations of Software Technology and Theoretical Computer Science*, held in Bangalore, India, in December, 1983. This paper was typeset at General Electric Corporate Research and Development using *Troff* software developed for the Unix operating system.

[†]Computer Science Branch, General Electric Corporate Research and Development, Schenectady, New York 12345.

simply rewriting of terms using the rules corresponding to axioms of an algebraic system. Their completion procedure looks for representative terms that have *two* normal forms and then tries to “patch up” by adding a new rule involving the two normal forms. This method has been quite successful in many practical cases: free groups, commutative semigroups, and polynomial ideals are but a few of them. The Knuth-Bendix completion procedure has some other applications as well, such as establishing consistency of equational theories and proving inductive properties using what has become known in the literature as the *inductionless induction* method [23],[24].

In this paper, we discuss the application of the Knuth-Bendix completion procedure on Thue systems. Thue systems are rewriting systems specified using equations over strings (thus there are no variables, but concatenation satisfies the associativity property). It is our belief that understanding the behavior of the Knuth-Bendix completion procedure on Thue systems will also provide some insight into its behavior on term-rewriting systems. Further, Thue systems have recently been studied in their own right by Book and others in the context of formal language theory, monoid presentation, and word problems for finitely presented monoids. The application of the Knuth-Bendix completion procedure for such systems has not been studied so far.

We give a set of transformations that when applied on a Church-Rosser Thue system yields an equivalent reduced (or minimal) Church-Rosser system. We show that these transformations on Church-Rosser Thue systems themselves have the Church-Rosser property, and thus, for every Thue system that has a finite equivalent Church-Rosser Thue system, there is a unique finite reduced Church-Rosser Thue system equivalent to it.¹ Using properties of reduced Church-Rosser systems, we also develop conditions under which a class of special Thue systems have equivalent finite Church-Rosser systems. We also show that a version of the Knuth-Bendix completion procedure, when applied to such a Thue system, always terminates and results in the finite reduced Church-Rosser Thue system equivalent to the original system. We discuss how the completion procedure can be modified to generate almost-confluent Thue systems.

Using concepts and results of the paper, we are also able to prove, in a straightforward manner, that the procedure always halts when applied to commutative Thue systems (i.e., presentations of commutative monoids) [18],[20]. We further exhibit a sufficient condition for the input Thue systems that, when satisfied, guarantees that the procedure will eventually terminate. More specifically, it is shown that if the left-hand sides of rules satisfy a certain condition regarding overlaps, then the Knuth-Bendix completion procedure terminates. Ground-term-rewriting systems can be translated to Thue systems that satisfy this condition, so the result applies to ground-term-rewriting systems, thus giving another proof of termination of the Knuth-Bendix procedure for ground-term-rewriting systems (see also [7]). We are thus also able to give a uniform treatment of some the known results in rewrite rule theory.

The paper is organized as follows: The next section gives definitions and properties of Thue systems. We introduce a new property of Thue systems, called

¹We have recently learned that Nivat and Benois [26] were the first to make this observation. A similar result was also obtained by Lankford and Ballantyne for term-rewriting systems, as reported in [18].

lexicographic confluence, which subsumes the Church-Rosser property. The lexicographic confluence property is used in later sections to study the Knuth-Bendix completion procedure. We prove some properties of the set of irreducible strings of a Thue system. In § 3, we introduce the notion of a reduced Thue system. We prove that if a Thue system T has an equivalent Church-Rosser system, then there exists a unique reduced Church-Rosser system equivalent to it. We also show that for a special Thue system that can be homomorphically mapped into a reduced Church-Rosser system, an equivalent Church-Rosser system exists if and only if the original system itself is Church-Rosser. In § 4, we discuss the application of the Knuth-Bendix completion procedure to Thue systems. First we prove that if there exists a Church-Rosser system equivalent to a Thue system T , then the Knuth-Bendix completion procedure, when applied on T , terminates with a reduced lexicographic confluent, and hence Church-Rosser, system equivalent to T . We also prove the termination of the completion procedure on commutative Thue systems (commutative monoids) as a corollary. Later, we modify the completion procedure to generate reduced almost-confluent systems. Section 5 discusses conditions under which the completion procedure terminates; this is used to show the termination of the completion procedure on parenthesized Thue systems and hence ground-term-rewriting systems.

2. Definitions. Let Σ be a finite alphabet. Σ^* is the monoid freely generated by Σ , or, in other words, the set of all finite strings over Σ . The empty string, which is the identity in the monoid, is denoted by λ , where λ is a symbol not in Σ . The length function on strings, denoted by $|u|$, can be defined as usual: $|\lambda| = 0$, $|ua| = |u| + 1$, where a is in Σ .

A Thue system T is a binary relation on Σ^* . The Thue congruence generated by T is the reflexive transitive closure \leftarrow_T^* of the relation \leftarrow_T , which is defined as follows: for any u and v in Σ^* such that $\langle u, v \rangle$ is in T or $\langle v, u \rangle$ is in T , and any x, y in Σ^* , $xuy \leftarrow_T xvy$. Two strings, w and z , are congruent mod T if $w \leftarrow_T^* z$. Two Thue systems T_1 and T_2 are equivalent if and only if they generate the same congruence relation. A Thue system T is commutative if and only if for each $\langle xy, z \rangle$ in \leftarrow_T^* , $\langle yx, z \rangle$ is also in \leftarrow_T^* . A Thue system T is called special if and only if for every $\langle u, v \rangle$ in T , either u or v is λ . Henceforth, we shall omit the subscript T whenever it is understood from the context.

Every element of a Thue system T is called an equation of T . Some equations of T can be oriented into rules depending upon the length of the sides of each equation. An equation $u = v$ is oriented into a rule $u \rightarrow v$ if $|u| > |v|$, or $v \rightarrow u$ when $|v| > |u|$; such a rule is called length-reducing or simply a reduction.

An equation $u = v$ in which u and v are of the same length cannot be oriented based on the length of u and v ; it is written as $u | - | v$. Such an equation is called a length-preserving rule. Later, we will discuss another way of orienting rules for the case when a total ordering is introduced on Σ^* . In that case, it would also be possible to uniquely orient equations whose two sides have strings of the same length.

Based on the above classification of the rules, a Thue system T can be partitioned into two components: 1. a subset of length-preserving rules, which will be called LP , and 2. the remaining subset of reductions, which will be called R . Only the rules in the R subset of T will be used for reducing (rewriting) strings unless stated otherwise.

For any x , if there are u and v , as well as a rule $l \rightarrow r$ in R of T such that $x = u l v$, then $x \rightarrow u r v$, read as x reduces to $u r v$ using the rule $l \rightarrow r$ of R . The reflexive transitive closure of this relation is the *reduction relation* generated by R of T (also called the *reduction relation* generated by T) on Σ^* ; this is denoted by \rightarrow^* whereas \rightarrow^+ stands for the transitive closure of \rightarrow .

Two strings x and y are called *joinable* (under \rightarrow) if and only if there exists a z such that $x \rightarrow^* z$ and $y \rightarrow^* z$. Strings x and y are *almost-joinable* if and only if there exist u and v such that $x \rightarrow^* u$, $y \rightarrow^* v$, and $u \mid \mid^* v$.

A Thue system T is called *Church-Rosser* if for each u, v , such that $u \rightarrow^* v$, u , and v are joinable. A Thue system T is called *confluent* if for each u, v, w , such that $u \rightarrow^* v$, $u \rightarrow^* w$, v , and w are joinable.

Note that the Church-Rosser property and the confluence property, though closely related, are not the same in case of Thue systems because Thue congruence \leftarrow^* and the reflexive, symmetric, and transitive closure of the reduction relation \rightarrow , in general, do not coincide, as the latter does not take into account length-preserving rules. A Church-Rosser system is confluent but the converse does not hold, as shown by the simple example $\{ \langle ab, cd \rangle, \langle ab, a \rangle, \langle cd, c \rangle \}$, which is confluent but not Church-Rosser.

A Thue system T is called *almost confluent* if for each u, v such that $u \rightarrow^* v$, u and v are almost joinable.

For a commutative Thue system, a string x can be expressed as a k -tuple $\langle x_1, \dots, x_k \rangle$ (sometimes called a Parikh vector), where $\Sigma = \{a_1, \dots, a_k\}$ and x_i is the number of times the letter a_i appears in x . A rule of a commutative Thue system can be expressed as a rule relating two such k -tuples.

2.1. Total ordering on strings and lexicographic confluence. We will introduce another property called "lexicographical confluence" that is not in the literature but, we think, is useful from theoretical as well as practical points of view.

Let $<$ be a total ordering on strings in Σ^* such that the following two properties hold:

1. $|x| < |y| \Rightarrow x < y$, and
2. $x < y \Rightarrow$ for any u, v , $u x v < u y v$.

The above properties are closely related to the properties of simplification orderings introduced by Dershowitz [6].

A family of total orderings satisfying the above two properties is the size and lexicographic ordering on strings induced by a total ordering on Σ defined as follows:

$x < y$ if and only if either

1. $|x| < |y|$ or
2. $|x| = |y|$, $x = a x'$, $y = b y'$, $a, b \in \Sigma$, and either $a < b$ or $a = b$ and $x' < y'$.

Such a total ordering $<$ on Σ^* can be used to orient equations whose two sides are of the same length. So, given a Thue system T , equations in the length-preserving component can also be oriented using $<$. Every rule in T is thus used for reduction. The symbol \rightarrow will be used to denote this reduction relation also, as long as it is evident from the context that the whole T is being used for reduction. We shall use \rightarrow' to specify the reduction relation induced by

T to distinguish it from the reduction relation \rightarrow induced by R . However, in § 5 we shall drop the prime. The reflexive, symmetric, and transitive closure of the relation \rightarrow' and the Thue congruence \leftrightarrow^* are the same.

A Thue system T is called *lexicographically confluent* (with respect to $<$) if and only if for every u, v , and w such that $u \rightarrow'^* v$ and $u \rightarrow'^* w$, there is a z such that $v \rightarrow'^* z$ and $w \rightarrow'^* z$. We abbreviate a “lexicographically confluent system” to a “lex-confluent system.”

THEOREM 2.1. *If T is Church-Rosser then T is lex-confluent.*

Proof. For every length-preserving rule $u \mid\mid v$ in a Church-Rosser system T , there is a z such that $u \rightarrow^* z$ and $v \rightarrow^* z$. So, irrespective of the way the length-preserving rules are oriented for \rightarrow' , T is lex-confluent. \square

Note that if T is lex-confluent, its R component (the set of length-reducing rules) need not be confluent, as the following example illustrates:

$$T = \{ a b \rightarrow c, a b \rightarrow d, c \mid\mid d \} .$$

T is also not Church-Rosser, as c and d do not reduce to the same string.

For any Thue system T , there need not exist an equivalent finite or infinite Church-Rosser system, but there is always an equivalent infinite lex-confluent system which can be obtained trivially from the congruence relation generated by T .

Once an ordering on strings is defined, it can be extended to rules in a natural way as follows:

$$l_1 \rightarrow r_1 < l_2 \rightarrow r_2 \text{ if and only if } l_1 < l_2 \text{ or } l_1 = l_2 \text{ and } r_1 < r_2 .$$

This ordering on rules is used later in some proofs.

2.2. Irreducible sets for Thue systems. For a Thue system T , x is *irreducible* (mod T) if and only if x cannot be reduced further using the R component of T ; x is *minimal* if and only if x is one of the smallest strings in its congruence class induced by T . Clearly, every minimal string is irreducible; but an irreducible string need not be minimal.

Let y be an irreducible string obtained by reducing x using rules of R ; y is also called a *normal form* of x . Let \bar{x} stand for a normal form of x under R . If a string x has a unique normal form under R , the normal form of x is also called its *canonical form*. It can be shown that every string has a unique normal form in a Church-Rosser system and thus a string is irreducible if and only if it is minimal.

Let $IRR(T)$ be the set of irreducible strings of T . Equivalent Thue systems can have different IRR sets; for example, $T_1 = \{ abc \rightarrow ab, abc \rightarrow c \}$ and $T_2 = \{ abc \rightarrow c, ab \rightarrow c \}$ are equivalent, but ab is in $IRR(T_1)$ but not in $IRR(T_2)$.

THEOREM 2.2. *If $T_1 \subseteq T_2$ and for every rule $l \rightarrow r$ in $T_2 - T_1$, l is reducible in T_1 , then $IRR(T_1) = IRR(T_2)$.*

Proof. That $IRR(T_2) \subseteq IRR(T_1)$ is obvious. Since every string reducible modulo T_2 is reducible modulo T_1 , $IRR(T_1) \subseteq IRR(T_2)$. \square

It can be easily seen that equivalent Church-Rosser (lex-confluent, almost-confluent) systems have the same IRR set. The following also holds.

THEOREM 2.3. *Let T and T' be two equivalent Thue systems. If T is Church-Rosser (lex-confluent) and $IRR(T) = IRR(T')$, then T' is also Church-Rosser (lex-confluent).*

Proof. By contradiction. Assume T' is not Church-Rosser (lex-confluent), then there exists an x such that it has two distinct irreducible forms, say w_1 and w_2 , in $IRR(T')$. Since $w_1 \xrightarrow{*} w_2$ and T is Church-Rosser, they have a common descendant, implying that at least one of w_1 and w_2 is reducible in T ; so, $IRR(T) \neq IRR(T')$, which is a contradiction. \square

A similar theorem about almost-confluent Thue systems is proved in [14].

For a Church-Rosser system, all irreducible elements are minimal and minimal elements are unique in their equivalence classes. These two properties can serve as an alternative characterization of Church-Rosser systems. For an almost-confluent system, only the first property holds.

3. Testing for the Church-Rosser property and critical pairs. Nivat and Benois [26] were the first to give a test for the Church-Rosser property and confluence of finite Thue systems. Book and O'Dunlaing [4] showed that this problem is tractable and gave a polynomial time algorithm. Kapur, Krishnamoorthy, McNaughton, and Narendran [13] improve on Book and O'Dunlaing's upper bound and give an $O(|T|^3)$ algorithm for this problem.

The conditions that a Thue system T must satisfy to be Church-Rosser can be stated as follows: We define *critical pairs* from rules; for T to be Church-Rosser, the two strings in each critical pair must be joinable.

1. For a length-preserving rule $l \rightarrow r$ in T , the critical pair is $\langle l, r \rangle$; l and r must be joinable.

2. Substring case: For rules $l_1 \rightarrow r_1$, $l_2 \rightarrow r_2$ in R , if l_2 is a substring of l_1 , then for every u and v such that $l_1 = u l_2 v$, both $p = u r_2 v$ and $q = r_1$, which form a critical pair $\langle p, q \rangle$, must be joinable.

3. Overlap case: For rules $l_1 \rightarrow r_1$, $l_2 \rightarrow r_2$, if $l_1 = u x$, $l_2 = x v$, then for every such u, v , and x , both $p = r_1 v$ and $q = u r_2$, which also form a critical pair $\langle p, q \rangle$, must be joinable. The rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ are called *overlapping* rules.

A critical pair $\langle p, q \rangle$ is called *nontrivial* if and only if $\bar{p} \neq \bar{q}$, where \bar{p}, \bar{q} are, respectively, normal forms of p and q ; otherwise, the critical pair $\langle p, q \rangle$ is called *trivial*. We assume the existence of an algorithm for computing a normal form of a string, which we will denote by $normal_form(x, R)$. A Thue system is Church-Rosser if and only if all its critical pairs are trivial. A pair $\langle p, q \rangle$ is in *reduced form* with respect to R if p and q cannot be reduced further by R . Henceforth, whenever we refer to a nontrivial critical pair $\langle p, q \rangle$, we assume that p and q are irreducible.

The test for lex-confluence is similar to that of the Church-Rosser property; we do not have case 1 because length-preserving rules are also oriented.

From the above definition of critical pairs, we have

LEMMA 3.1. *For any critical pair $\langle p, q \rangle$ of a Thue system T , T is equivalent to $T \cup \{\langle p, q \rangle\}$.*

4. Reduced Thue systems. A Thue system T is called *reduced* if for every rule in R and LP , neither its left-hand side (lhs) nor its right-hand side (rhs) can be rewritten using the remaining set of rules in R . Thus for every rule $l \rightarrow r$ in a reduced Thue system T , r is irreducible in T and l is irreducible in $T - \{l \rightarrow r\}$. In addition, we have

PROPOSITION 4.1. *For each rule $l \rightarrow r$ in a reduced Thue system T , every proper substring of l is irreducible in T .*

4.1. Church-Rosser Thue systems remain Church-Rosser under reduction. The definition implies that a reduced Church-Rosser system T cannot have an LP component. The following theorem states that Church-Rosser systems have an interesting property, namely that the transformation of stepwise reduction of Church-Rosser Thue systems itself has the Church-Rosser property.

THEOREM 4.2. *Given a Church-Rosser Thue system T , there is a unique reduced Church-Rosser system equivalent to T , effectively obtainable from T .*

We first prove that there is a finite reduced Church-Rosser Thue system equivalent to any finite Church-Rosser system T' and that this reduced system can be effectively obtained from T' . This proof is based on the following two lemmas, which state that the reduction of the lhs and rhs of rules in a Church-Rosser system does not affect the Church-Rosser property of the system. Later, we show the uniqueness of a finite reduced Church-Rosser Thue system.

LEMMA 4.3: *For a Church-Rosser Thue system T , if it has a rule $w_1 \rightarrow w_2$ whose lhs can be reduced using the remaining set of rules in T , then the system $T' = T - \{w_1 \rightarrow w_2\}$ is equivalent to T and is also Church-Rosser.*

Proof. Since w_1 can be reduced, there is a rule $u_j \rightarrow v_j$ in T' such that $w_1 = l_1 u_j r_1$, and $w_1 \rightarrow l_1 v_j r_1$. Since T is Church-Rosser, there is some irreducible z such that $w_2 \rightarrow^* z$ as well as $l_1 v_j r_1 \rightarrow^* z$. Since $|l_1 v_j r_1| < w_1$, the rule $w_1 \rightarrow w_2$ of T is not applied in the reduction of either w_2 or $l_1 v_j r_1$ in T , thus implying that they reduce to z in T' also. Thus, w_1 and w_2 are congruent mod T' , which means that T and T' are equivalent. The fact that T' is Church-Rosser follows from Theorems 2.2 and 2.3. \square

LEMMA 4.4: *If there is a rule $w_1 \rightarrow w_2$ in a Church-Rosser system T , such that w_2 can be reduced using other rules, then $T' = T - \{w_1 \rightarrow w_2\} \cup \{w_1 \rightarrow \text{normal-form}(w_2, T - \{w_1 \rightarrow w_2\})\}$ is equivalent to T and is Church-Rosser.*

Proof. For any reduction sequence $w_2 \rightarrow^+ z$ in T , since the rule $w_1 \rightarrow w_2$ cannot be applied, $w_2 \rightarrow^+ z$ in T' also, which establishes the equivalence of T and T' . It follows easily from Theorem 2.3 that T' is Church-Rosser. \square

Proof of Theorem 4.2. Our algorithm consists of applying the two lemmas as often as possible. Clearly this procedure terminates in a reduced equivalent Church-Rosser system.

But we must also prove that two equivalent reduced Church-Rosser systems T_1 and T_2 are the same. To this end, let $w_1 \rightarrow w_2$ be a rule in T_1 which is not in T_2 (this must be possible, otherwise we are done). Since T_2 is Church-Rosser, $w_1 \rightarrow^+ w_2$ in T_2 because w_2 is an irreducible string of T_1 and also T_2 ($IRR(T_1) = IRR(T_2)$ by Theorem 2.2). So, there is a rule in T_2 that applies to w_1 . Every proper substring of w_1 is irreducible (Lemma 4.1); the rule must be $w_1 \rightarrow w'$, where w' is also irreducible. We have w' and w_2 being equivalent in T_2 , which is impossible because both are irreducible. \square

The above proof easily extends to lexicographically confluent systems; instead of using the ordering on strings induced by their length in the proof of Lemma 4.3, a total ordering $<$ on strings as defined in §§ 2.2 is used.

THEOREM 4.5. *If there is a Church-Rosser Thue system equivalent to a reduced lex-confluent system T , then T itself is Church-Rosser.*

Proof. Since T is reduced, for every rule $l \rightarrow r$ in T , r is irreducible. In particular, if T has a rule $l' \rightarrow r'$, such that $|l'| = |r'|$, then r' is irreducible. But if there is a Church-Rosser system equivalent to T , then neither l' nor r' can be irreducible, implying that T cannot have rules whose two sides are the same length. Thus T is Church-Rosser. \square

Thus an algorithm for reducing a Church-Rosser (or lex-confluent) system T is 1. repeatedly throw away a rule in T whose lhs is reducible by the remaining rules, and 2. for every rule in T whose rhs can be reduced using the remaining rules, replace the rhs by its normal form. Later we discuss an extension of this algorithm for almost-confluent Thue systems.

4.2. Special Church-Rosser systems. We discuss below conditions characterizing special Church-Rosser systems that extend some of the results reported in [2],[3]. These conditions are based on homomorphically mapping a special Thue system into a special reduced Church-Rosser Thue system.

Let Δ be a set of generators possibly different from Σ . Let ϕ be a homomorphism from Σ^* to Δ^* ; so, $\phi(\lambda) = \lambda'$, where λ' is the identity of the monoid Δ^* , and $\phi(uv) = \phi(u)\phi(v)$. Furthermore, we require that the length function on Δ^* , also denoted as $\|\cdot\|$, satisfy the following property: $\|u\| > \|v\|$ if and only if $\|\phi(u)\| > \|\phi(v)\|$. The *version of T under ϕ* , denoted by T^ϕ , is defined as

$$T^\phi = \{ (\phi(u), \phi(v)) \mid (u, v) \in T \}.$$

Again, note that $x \rightarrow^* y \pmod T$ implies $\phi(x) \rightarrow^* \phi(y) \pmod{T^\phi}$. A homomorphism ϕ is said to be *length-preserving* if $\|\phi(a)\| = 1$ for all $a \in \Sigma$.

THEOREM 4.6. *Let T be a special Thue system and ϕ be a length-preserving homomorphism such that T^ϕ is a reduced Church-Rosser system. Then T has an equivalent Church-Rosser system if and only if T itself is Church-Rosser.*

Proof. The “if” part is trivial. We prove the “only if” part by contradiction.

Assume T is not Church-Rosser. Then there must be rules $x \rightarrow \lambda$ and $y \rightarrow \lambda$ such that one of their critical pairs is nontrivial. There are two possibilities:

1. $x = uv$ and $y = vw$ for some $u, v, w \neq \lambda$ and the critical pair is $\langle u, w \rangle$. Clearly $u \neq w$. Note that $\phi(u) \rightarrow^* \phi(w) \pmod{T^\phi}$. And, $\phi(u)$ must be equal to $\phi(w)$, since T^ϕ is a reduced Church-Rosser system and hence both $\phi(u)$ and $\phi(w)$ must be irreducible mod T^ϕ . Also, $\|u\| = \|w\|$. Now if T has an equivalent Church-Rosser system, then there must be a z shorter than both u and w such that $u \rightarrow^* z \rightarrow^* w$. This is obviously impossible since both $\phi(u)$ and $\phi(v)$ are irreducible mod T^ϕ .

2. $x = uvv$ for some u, v such that $uv \neq \lambda$ and the critical pair is $\langle uv, \lambda \rangle$. This case is evidently an impossibility, since $\phi(x)$ must be irreducible mod $T^\phi - (\phi(x), \lambda)$. \square

From the above theorem, we immediately get the following results, already reported in [3],[4].

A Thue system T is called *homogeneous* if and only if for every $\langle u, v \rangle, \langle x, y \rangle$ in T , either $\|u\| = \|x\|$ and $\|v\| = \|y\|$, or $\|u\| = \|y\|$ and $\|v\| = \|x\|$.

COROLLARY 4.7. *Let T be a homogeneous special Thue system. Then T has an equivalent Church-Rosser system if and only if T itself is Church-Rosser.*

Proof. Define $\Delta = \{t\}$ and $\phi(a) = t$ for all a in the alphabet of T . A single-rule special Thue system with a rule $t^n \rightarrow \lambda$ is reduced Church-Rosser. \square

COROLLARY 4.8. *If T is a special Thue system that has only a single rule, then T has an equivalent Church-Rosser system if and only if T is itself Church-Rosser.*

From the above theorem, we also get a generalization of Corollary 4.8 that suggests how to test whether a class of special Thue systems has equivalent Church-Rosser systems by mapping them to their commutative versions.

The *commutative version* of a Thue system T , denoted by T^C , is T with the commutative law built into it. As mentioned earlier, this can be represented as a

set of relations between k -tuples of integers (often referred to in the literature as Parikh-vectors) where k is the cardinality of Σ . For string w , let \bar{w} stand for its Parikh-vector. Thus,

$$T^C = \{ (\bar{u}, \bar{v}) \mid (u, v) \in T \} .$$

Examples. Let $\Sigma = \{a, b\}$.

$$1. T = \{ (aba, bb) \}. T^C = \{ ((2,1), (0,2)) \}.$$

$$2. T = \{ (aba, \lambda), (baa, \lambda) \}. T^C = \{ ((2,1), (0,0)) \}.$$

Clearly, for all $x, y, x \leftrightarrow^* y \text{ mod } T$ implies $\bar{x} \leftrightarrow^* \bar{y} \text{ mod } T^C$.

We can define a homomorphism ϕ from T to its commutative version T^C as follows: Δ in this case is the set of k basis vectors, where a basis vector is a k -tuple in which exactly one component is nonzero and is 1. The concatenation operation on Δ^* is the vector addition; the identity is the zero vector, and the length of a vector is the sum of all components in the vector. It can be easily verified that ϕ is indeed a homomorphism that maps strings to vectors; furthermore, ϕ is length-preserving. Using Theorem 4.6 above, we have the following:

THEOREM 4.9. *Let T be a special Thue system such that T^C is a reduced Church-Rosser system. Then T has an equivalent Church-Rosser system if and only if T itself is Church-Rosser.*

5. The Knuth-Bendix completion procedure.

5.1. How to obtain a reduced Church-Rosser Thue system.

5.1.1. Completing a Thue system to get an equivalent lex-confluent system. For a finite Thue system T that is not lex-confluent, it is possible to generate an equivalent lex-confluent system by adapting the Knuth-Bendix completion procedure for term-rewriting systems to Thue systems. (Note that because we will discuss mostly lex-confluence in this section, \rightarrow represents the reduction relation generated by all the rules, including the rules whose sides are of the same length.) If the test for lex-confluence fails, all nontrivial critical pairs generated during the lex-confluence test are added to T and the test for lex-confluence on the modified T is repeated. This transformation is performed until the resulting system is lex-confluent.

An inefficient version of the Knuth-Bendix completion procedure for Thue systems is given below in which redundant rules in T are not deleted. Later, a more efficient version will be presented in which redundant rules in T are deleted. The function $CP(T)$ generates all nontrivial critical pairs of T in normal form; if $CP(T)$ is empty, then T is lex-confluent. We do not need to generate critical pairs of rules in T_{i+1} that were also in T_i .

```
Knuth-Bendix Procedure (T):
  i := 0;
  T0 := Normalize(T);
  CE := CP(T0);
  while CE ≠ null do
    Ti+1 := Normalize(Ti ∪ CE);
    i := i+1;
    CE := CP(Ti)
  endwhile
  output(Ti);
```

```

Normalize (T):
  unmark all rules in T.
  while T has an unmarked rule  $l \rightarrow r$  do
     $T' := T - \{l \rightarrow r\}$ ;
     $\langle l', r' \rangle := \langle \text{normal\_form}(l, T'), \text{normal\_form}(r, T') \rangle$ ;
    if  $\langle l, r \rangle \neq \langle l', r' \rangle$  and  $l' \neq r'$ 
      then
         $T := T \cup \{l' \rightarrow r'\}$ ;
        mark  $l' \rightarrow r'$ 
      endif,
    mark  $l \rightarrow r$ 
  endwhile
  return T;

```

Note that the procedure *Normalize* does not delete any rules. It merely adds rules that are obtained from further reducing the two sides of rules already in T without removing the original rule. The procedure *Normalize* is nondeterministic if strategies for 1. picking an unmarked rule and 2. computing a normal form of a string are not specified.

There is a possibility of the Knuth-Bendix completion procedure going on forever. We discuss below the conditions under which the Knuth-Bendix procedure is guaranteed to terminate.

Let T_∞ be T_k if the above procedure terminates after k iterations; otherwise, the union of T_i for all i .

LEMMA 5.2. *T and Normalize(T) are equivalent.*

Proof. Since T is a subset of *Normalize(T)*, it needs to be shown that the extra rules in *Normalize(T)* do not introduce any extra congruence classes. Each rule $l \rightarrow r$ in *Normalize(T)* which is not in T is obtained by reducing some rule $l' \rightarrow r'$ in T . Since $l = \text{normal_form}(l', T - \{l' \rightarrow r'\})$ and $r = \text{normal_form}(r', T - \{l' \rightarrow r'\})$, and l' and r' are congruent in T , l and r are also congruent in T . \square

From Lemmas 3.1 and 5.2, T is equivalent to T_i for each i . And also, T_∞ is equivalent to T .

LEMMA 5.3. *For any i , if $IRR(T_{i+1}) = IRR(T_i)$, then T_i is lex-confluent.*

Proof. By contradiction. If T_i is not lex-confluent, then there is a nontrivial critical pair $\langle p, q \rangle$ where $p \neq q$ and p and q are in $IRR(T_i)$. Since the rule $p \rightarrow q$ or $q \rightarrow p$ is included in T_{i+1} , we get $IRR(T_i) \neq IRR(T_{i+1})$, which is a contradiction. \square

It is obvious from the above lemma that the *IRR* set keeps decreasing in every iteration of the Knuth-Bendix procedure (since $T_i \subseteq T_{i+1}$, $IRR(T_{i+1}) \subseteq IRR(T_i)$). The procedure thus terminates when the *IRR* set becomes stable; i.e., for some i , $IRR(T_i) = IRR(T_{i+1})$. It can be noted that the set of reducible strings of a Thue system T form an ideal in the free semigroup Σ^* . For commutative semigroups, the irreducible set of strings always becomes stable because of the *finite ascending chain* (also called the *Noetherian*) condition for its ideals [9]. So, we get the result that for commutative Thue systems, the Knuth-Bendix procedure will always terminate as a corollary of the above lemma. A similar argument is used to show the termination of Buchberger's algorithm for finding a Grobner basis for polynomial ideals over a field [5], as well as polynomial ideals over a Euclidean ring [12] (also see references in Mayr and Meyer [20] as well as [11] for similar observations).

The correctness of the Knuth-Bendix completion procedure follows from the following theorem.

THEOREM 5.4. T_∞ is lex-confluent.

Proof. By contradiction. If T_∞ is not lex-confluent, it could fail because of any of the above two tests for lex-confluence. This is impossible because if any of the tests fails in an iteration i , then the corresponding nontrivial critical pairs are added in the $i+1^{\text{th}}$ iteration. \square

For every finite Thue system T , the Knuth-Bendix procedure thus gives an equivalent lex-confluent system T_∞ . If the Knuth-Bendix procedure terminates, then T_∞ is finite; otherwise, as we shall see, T_∞ is infinite.

5.1.2. The completion procedure without deletion of redundant rules.

THEOREM 5.5. Given a finite Thue system T , if there exists a finite lex-confluent Thue system equivalent to T , then the Knuth-Bendix procedure terminates with a finite (not necessarily reduced) lex-confluent Thue system T' , which is equivalent to T .

Proof. Assume that there exists a finite lex-confluent Thue system equivalent to T . Then using the results of § 3, there is a reduced finite lex-confluent system T' equivalent to T . Furthermore, T_∞ is equivalent to T' and both are lex-confluent, so the set of irreducible strings $IRR(T_\infty)$ and $IRR(T')$ are the same. It needs to be shown that $T_\infty = T_k$ for some k .

LEMMA 5.6. For every rule $l \rightarrow r$ of T' , there is some T_i having $l \rightarrow r$.

Proof. By contradiction. Assume that the statement is not true. Pick the smallest rule $l \rightarrow r$ in T' that does not appear in any T_i ; so, $l \rightarrow r$ is also not in T_∞ . Let j be the maximum over the iteration numbers in which the rules in T' less than $l \rightarrow r$ get added; i.e., T_j has all rules of T' less than $l \rightarrow r$.

Since T' and T_∞ are equivalent as well as lex-confluent, l and r are congruent in T_∞ , implying that l and r must reduce to the same string in T_∞ . Since r is in $IRR(T') = IRR(T_\infty)$, l must reduce to r in T_∞ , implying that T_∞ has a rule $l' \rightarrow r'$ that reduces l . By Proposition 4.1, no proper substring of l is reducible, implying that T_∞ must have a rule $l \rightarrow r'$, and both in T_∞ and T' , $r' \rightarrow^* r$. Let j' be the iteration number when $l \rightarrow r'$ gets added. In T' , $r' \rightarrow^* r$ using rules smaller than the rule $l \rightarrow r$ as $r' < l$, which are in T_∞ . So, either in the i^{th} iteration, where $i = \max(j, j')$, or the $i+1^{\text{th}}$, r' would be reduced to r and the rule $l \rightarrow r$ would be added, leading to a contradiction. \square

We now complete the proof of Theorem 5.5. Since a rule once added never gets deleted, Lemma 5.6 implies that T_∞ contains T' . But T' is finite, so after finitely many iterations of the loop in the procedure, all rules of T' get added into T_∞ . But T' is lex-confluent, so after the iteration, say k^{th} , in which the last rule of T' is added to T_∞ , the test for lex-confluence would succeed, implying that the procedure would terminate before the loop is executed the $(k+1)^{\text{st}}$ time.

5.1.3. The completion procedure with deletion of redundant rules. The above version of the Knuth-Bendix procedure is clearly inefficient, because the original rules from which the simplified rules are obtained in the normalization process are not discarded. However, its proof of termination is easier. An optimized version of the Normalize procedure follows in which redundant rules are discarded.

Normalize' (T):

unmark all rules in T.

while T has an unmarked rule $l \rightarrow r$ do


```

T' := T - { l → r };
< l', r' > := < normal_form(l, T'), normal_form(r, T') >;
if l' = r'
then T := T'
else
  T := T' ∪ { l' → r' };
  mark l' → r'
endif
endwhile
return T;

```

We first illustrate the completion procedure on a simple example:

$$T = \{ bb \rightarrow a, bab \rightarrow a, baa \rightarrow aab \},$$

where $a < b$.

First iteration: $ba \rightarrow ab$ and $aab \rightarrow ab$ are added; $bab \rightarrow aab$ is deleted and $bab \rightarrow a$ is replaced by $aa \rightarrow a$, which results in the deletion of $aab \rightarrow ab$. $T_1 = \{ bb \rightarrow a, aa \rightarrow a, ba \rightarrow ab \}$.

After this, the system T_1 is lex-confluent.

Using the above procedure, a stronger version of Theorem 5.5 can be proved. The termination proof requires an additional step, which is to show that once desired rules get added, they are never deleted. We first show that $\text{Normalize}'(T)$ is equivalent to T and later that the *IRR* set of the system obtained using $\text{Normalize}'$ is the same as the *IRR* set of the system obtained using Normalize .

LEMMA 5.7. $\text{Normalize}'(T)$ is equivalent to T .

Proof. It is sufficient to show that in $\text{Normalize}'$, the deletion of a rule does not affect the Thue system. There are two situations in which a rule gets deleted: (a) For a rule $l \rightarrow r$ in T , l and r reduce to the same string in $T - \{l \rightarrow r\}$. In this case, congruence relations generated by T and $T - \{l \rightarrow r\}$ is the same.

(b) For a rule $l \rightarrow r$ in T , if $l' \rightarrow r' \neq l \rightarrow r$, where $l' = \text{normal_form}(l, T - \{l \rightarrow r\})$ and $r' = \text{normal_form}(r, T - \{l \rightarrow r\})$, then $l \rightarrow r$ is replaced by $l' \rightarrow r'$. In $T - \{l \rightarrow r\}$, l' is congruent to l and r' is congruent to r , so $T - \{l \rightarrow r\} \cup \{l' \rightarrow r'\}$ is equivalent to T .

Note that T_i in the optimized version of the Knuth-Bendix procedure is always a subset of T_i of the unoptimized version. Furthermore, since $\text{Normalize}'(T) \subseteq \text{Normalize}(T)$, $\text{IRR}(\text{Normalize}'(T)) \supseteq \text{IRR}(\text{Normalize}(T))$.

LEMMA 5.8. $\text{IRR}(\text{Normalize}(T)) = \text{IRR}(\text{Normalize}'(T))$.

Proof. If a rule $l \rightarrow r$ is deleted in $\text{Normalize}'$, it means that l is reducible by some other rule $l' \rightarrow r'$. Thus, if a string is reducible by $l \rightarrow r$, then it is also reducible by $l' \rightarrow r'$. \square

From Lemmas 3.1 and 5.7, we get that T_i is equivalent to T . So, T_∞ is equivalent to T . Further, we have

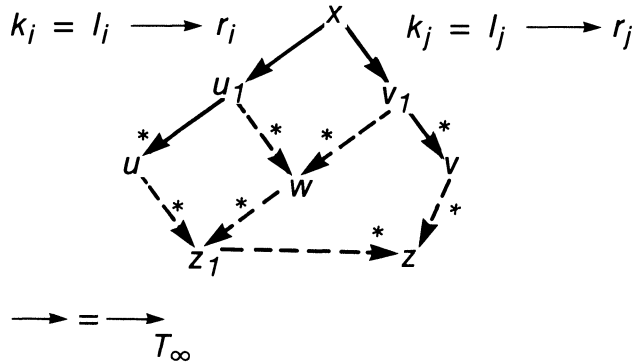
LEMMA 5.9. T_∞ is lex-confluent.

Proof. We need to show that for any string $x, u, v, x \xrightarrow{*} u$ in m steps, $m \geq 0$, and $x \xrightarrow{*} v$ in n steps, $n \geq 0$, there is w such that $u \xrightarrow{*} w$ and $v \xrightarrow{*} w$. Since T_∞ is Noetherian, the proof has the structure of the proof of Lemma 2.4 in [10], which states that a Noetherian relation is confluent if and only if it is locally confluent.

For $m = 0$ or $n = 0$, the above trivially holds. So, we assume that both m and $n > 0$. Let $x \rightarrow_{T_\infty} u_1$ by a rule $l_i \rightarrow r_i$ and $x \rightarrow_{T_\infty} v_1$ by a rule $l_j \rightarrow r_j$ such that $u_1 \rightarrow_{T_\infty}^* u$ and $v_1 \rightarrow_{T_\infty}^* v$. Let T_{k_i} and T_{k_j} be the Thue systems generated in the Knuth-Bendix procedure having $l_i \rightarrow r_i$ and $l_j \rightarrow r_j$, respectively, such that $|k_i - k_j|$ is minimum.

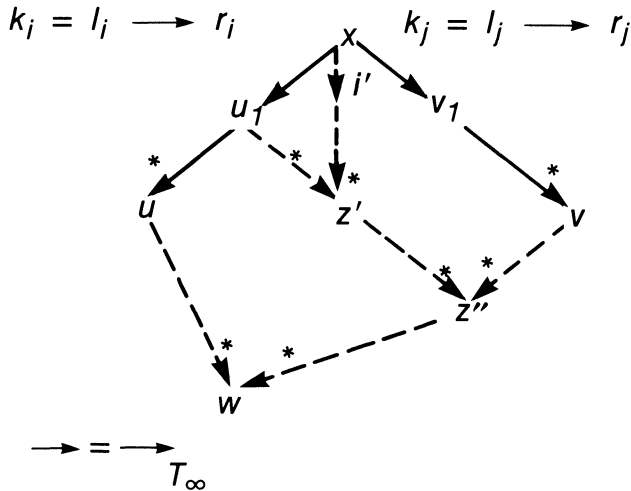
Let x be the smallest string under the ordering $<$ being used for confluence such that there is no w with $u \rightarrow_{T_\infty}^* w$ and $v \rightarrow_{T_\infty}^* w$ and derive a contradiction by induction on $|k_i - k_j|$.

Basis: $|k_i - k_j| = 0$, in the iteration after $\max(k_i, k_j)$, all nontrivial critical pairs of $l_i \rightarrow r_i$ and $l_j \rightarrow r_j$ are generated, giving us the following diagram. But then both $u_1 < x$ and $v_1 < x$, which is a contradiction.



Inductive step: Assume for all $k < |k_i - k_j|$, to show for $|k_i - k_j|$:

Without any loss of generality, assume that $k_j > k_i$. The rule $l_i \rightarrow r_i$ disappears in some iteration i' , such that $k_i < i' \leq k_j$. So, there is z such that in $T_{i'}$, l_i and r_i reduce to z ; this implies that x and u_1 reduce to some z' in \rightarrow_{T_∞} , since $|k_j - i'| < |k_j - k_i|$, by the inductive hypothesis, there is a z'' such that $z' \rightarrow_{T_\infty}^* z''$ and $v \rightarrow_{T_\infty}^* z''$. Since $u_1 < x$, there is a w such that $u \rightarrow_{T_\infty}^* w$ and $z'' \rightarrow_{T_\infty}^* w$, which is a contradiction.



□

A similar result about the application of the Knuth-Bendix completion procedure on term-rewriting systems is proved in [11]. The above proof is simpler because we are considering Thue systems and also, the version of our completion procedure is not as efficient as in [11]. Using Lemma 5.9, we can prove a stronger version of Theorem 5.5.

THEOREM 5.10. *Given a finite Thue system T , if there exists a finite reduced lex-confluent Thue system T' equivalent to T , then the Knuth-Bendix procedure using the optimized Normalize procedure terminates with T' .*

One way to prove the above theorem is to show a stronger version of Lemma 5.6, that every rule of T' gets added in some iteration of the Knuth-Bendix procedure and never gets deleted afterward (Lemma 5.6 only states that every rule of T' gets added to some T_i in the i^{th} iteration). Theorem 5.10 then follows, because in the iteration k in which the last rule of T' is added, T_k would include T' and every rule other than those in T' would be deleted by the optimized Normalize procedure.

LEMMA 5.11. *Every rule in T' gets added in some iteration i and never gets deleted afterward.*

Proof. By contradiction. Similar to the proof of Lemma 5.6, let $l_j \rightarrow r_j$ be the smallest rule of T' with respect to the ordering $<$ that either

- (a) never gets added in any iteration (so it is not in T_∞), or
- (b) gets added in iteration i but later is deleted in iteration $i' > i$.

That is, all rules less than $l_j \rightarrow r_j$ in T' get added by some iteration m and never get deleted.

Case (a). The proof of this part is the same as the proof of Lemma 5.6, so it is omitted.

Case (b). Once the rule $l_j \rightarrow r_j$ is added into T_i , the only way it can get deleted in i^{th} iteration, $i' > i$, is when Normalize' reduces l_j (because r_j is irreducible). And, the only way l_j can be reduced is if there is a rule $l_j \rightarrow r'_j$ in $T_{i'}$. Furthermore, this rule is still unmarked; otherwise, its lhs would have been reduced using the rule $l_j \rightarrow r_j$. There are two subcases: 1. l_j reduces to r_j , implying $r'_j \rightarrow^* r_j$ in $T_{i'}$: Normalize' would then replace the rule $l_j \rightarrow r'_j$ later by $l_j \rightarrow r_j$, meaning that $l_j \rightarrow r_j$ does not get deleted in i' , which is a contradiction. 2. l_j reduces to l'_j , implying that $r'_j \rightarrow^* l'_j$ in $T_{i'}$: In this case also, Normalize' would first replace $l_j \rightarrow r_j$ by $l'_j \rightarrow r_j$. Later, Normalize' would also replace the rule $l_j \rightarrow r'_j$ by $l_j \rightarrow r_j$, again meaning that the rule under consideration does not get deleted in i' , which is a contradiction. \square

Using Theorem 4.5, which relates reduced finite lex-confluent systems and reduced finite Church-Rosser systems, and Theorem 5.10, we have:

THEOREM 5.12. *Given a finite Thue system T , if there exists a finite reduced Church-Rosser system T' equivalent to T , then the optimized Knuth-Bendix procedure terminates with T' .*

Huet in [11] discussed a version of the Knuth-Bendix completion procedure that is more efficient than the optimized version given above. Kapur and Sivakumar [16] have given an improvement over Huet's version. The proof discussed above extends to these versions of the Knuth-Bendix procedure, giving us a result similar to Theorem 5.12.

5.2 Reduced almost-confluent Thue systems. Nivat and Benois [26] also gave conditions under which a Thue system is almost-confluent. For length-reducing rules, the conditions are similar to the conditions for the Church-Rosser

property, except that for every critical pair $\langle p, q \rangle$, p and q must be almost-joinable, and not necessarily joinable. In addition, interaction between length-reducing rules and length-preserving rules are also considered.

For every rule, we define *critical strings* whose interaction must be considered to check for the almost-confluence property: For a length-reducing rule $l \rightarrow r$, the critical string is l ; for a length-preserving rule $b \mid\mid s$, both b and s are critical strings. For every pair of rules such that at least one rule is length-reducing, the following two conditions must be met: Let (l_i, r_i) and (l_j, r_j) be such a pair of rules. Then if l_i and l_j are critical strings, then

(a) If $l_i = uv$, $l_j = vw$ for some u, v, w where $|u|, |v|, |w| > 0$ (i.e., the two rules properly overlap), then for every such u, v , and w , the critical pair $(u r_j, r_i w)$ must be almost-joinable.

(b) If $l_i = u l_j w$ for some u, w then for every such u, w , the critical pair $(r_i, u r_j w)$ must be almost-joinable.

Thus, one has to consider all possible overlaps between

- (1) left-hand sides of length-reducing rules, and
- (2) left-hand side of a length-reducing rule and *both* sides of a length-preserving rule.

It is shown in [14] that the test for the almost-confluence property is PSPACE-complete.

Just as in the case of Church-Rosser systems, given an almost-confluent Thue system, we can obtain a reduced almost-confluent Thue system from it by getting rid of redundancies.

1. For every length-reducing rule $l \rightarrow r$ in R , if r is reducible by other rules in R , then replace the rule by $l \rightarrow r'$, where r' is a normal form of r under R .

2. (a) For every length-reducing rule $l \rightarrow r$ in R , if l is reducible by other rules in R , then delete the rule from R . (b) For every length-preserving rule $b \mid\mid s$, if either b or s is reducible by a rule in R , then delete the rule from T .

As stated in § 3, the above algorithm is a slight extension of the algorithm for obtaining a reduced Church-Rosser system; for a proof of correctness of the algorithm, see [14].

5.2.1. A normal form for almost-confluent systems. Using the above transformations, we can obtain an equivalent reduced almost-confluent system from every almost-confluent system. Note that unlike in the case of reduced Church-Rosser Thue systems, we do not have a unique reduced almost-confluent Thue system because of the length-preserving component. The following theorem tells us this property of reduced almost-confluent systems.

THEOREM 5.13. *Let T_1 and T_2 be two equivalent reduced almost-confluent systems.*

(a) *$LP(T_1)$ and $LP(T_2)$ are equivalent; i.e., the congruence relations generated by the length-preserving components of T_1 and T_2 are the same.*

(b) *For every rule $l \rightarrow r$ in $R(T_1)$, there exists a rule $l \rightarrow r'$ in $R(T_2)$ such that $r \mid\mid^* r'$ and vice versa.*

Proof. (a) Assume there exists a rule $(x \mid\mid y)$ in T_1 such that not $x \mid\mid^* y$ in T_2 . Because T_2 is almost-confluent, this implies that x and y are not irreducible and hence not minimal in T_2 . Thus x and y are reducible in T_1 also, which contradicts the fact that T_1 is reduced. This implies that $LP(T_1) = LP(T_2)$ are equivalent.

(b) b and s are almost-joinable in T_2 . Because all right-hand sides in a reduced system are irreducible, s is irreducible both in T_1 and T_2 . Since no proper substring of b can be reducible, there must be a rule $(b \rightarrow t)$, t irreducible. Therefore s and t are equivalent and irreducible, and this implies $s \mid\mid^* t$ (by Lemma 2.2). \square

It is possible to get rid of more redundancies in the LP component of a reduced almost-confluent Thue system to obtain an LP component that is minimal in the following sense: For any length-preserving rule $b \mid\mid s$, b and s are not related by other length-preserving rules in LP . Thus, for any length-preserving rule $b \mid\mid s$ in a reduced almost-confluent system T , if b and s are equivalent using the remaining length-preserving rules in $LP(T)$, then we can delete $b \mid\mid s$ from $LP(T)$.

In addition, if we assume a total ordering on strings that is an extension of the ordering induced by the length of strings (the size and lexicographic ordering discussed in § 2 is an example of such an ordering), we can orient the length-preserving rules also and use them as reductions for generating normal forms of reduced almost-confluent systems. In that case, the right-hand side of every rule in R can also be normalized with respect to the length-preserving rules. Despite these transformations, it is still not always possible to obtain a unique reduced almost-confluent system equivalent to a given almost-confluent system. For example, consider the following equivalent almost-confluent systems:

$$T_1 = \{ cbc \mid\mid bba, cd \mid\mid ab, db \mid\mid bc \},$$

$$T_2 = \{ abb \mid\mid bba, cd \mid\mid ab, db \mid\mid bc \}.$$

Both T_1 and T_2 are reduced. Even if the length-preserving rules are oriented using the length and lexicographic ordering induced by the ordering $a < b < c < d$ on the alphabet, T_1 and T_2 remain reduced.

As shown in § 4, for Church-Rosser systems, the transformations for obtaining a reduced Church-Rosser system have the Church-Rosser property; however, for almost-confluent systems, the transformations for obtaining a reduced almost-confluent system only have the almost-confluence property in the following sense: Given two equivalent almost-confluent systems, the above transformations for obtaining an equivalent reduced almost-confluent system may result in two distinct equivalent reduced almost-confluent systems. The following example illustrates this:

$$T_3 = \{ cde \rightarrow ab, cde \rightarrow ba, ab \mid\mid ba \}.$$

The system T_3 is almost-confluent but is not reduced. Two different reduced almost-confluent systems can be obtained from T_3 by applying the above transformations depending upon which of two rules, $cde \rightarrow ab$ or $cde \rightarrow ba$, is considered first.

5.2.2. A completion procedure for almost-confluence. Given a Thue system T which is not almost-confluent, it is possible to generate from T a reduced almost-confluent system equivalent to T , if such a system exists, by a completion procedure which is in the spirit of the Knuth-Bendix completion procedure discussed earlier. In the test for almost-confluence, if for any pair of rules the conditions are not met, i.e., the normal forms of two strings generated from the superposition are not equivalent by the length-preserving rules, then we modify

the system by adding a rule that ensures that the critical pair under consideration is almost-joinable; in this way, we keep adding new rules to both R and LP components of T whenever the need arises until the almost-confluence test is met.

Knuth-Bendix Procedure (T):

```

i := 0;
T0 := Normalize(T);
CE := CP(T0);
while CE ≠ null do
  Ti+1 := Normalize(Ti ∪ CE);
  i := i + 1;
  CE := CP(Ti)
endwhile
output(T);

```

Normalize(T):

```

unmark all rules in T.
while T has an unmarked a rule  $l \rightarrow r$  do
  T' := T - { < l, r > };
  < l', r' > := < normal_form(l, T'), normal_form(r, T') >;
  if almost_joinable(l', r', T')
    then T := T'
  else
    T := T' ∪ { < l', r' > };
    mark < l', r' >
  endif
endwhile
return T;

```

In the above procedures, the procedure CP generates all nontrivial critical pairs (which do not reduce to normal forms equivalent by $LP(T)$). The procedure `normal_form` generates a normal form of x using the length-reducing rules in T , whereas `almost_joinable` checks whether two strings are equivalent using the length-preserving and length-reducing rules of T . If strings in a non-trivial critical pair after normalization are of the same length, then the corresponding rule is added to the length-preserving component of T and is subsequently used to check for the almost-joinability condition. This completion procedure to generate almost-confluent systems is thus different from other uses of the Knuth-Bendix completion procedure discussed in the literature, because in this case the simplification theory generated by the length-preserving rules is also being extended; some of the new rules being generated are used as reduction while others are used as simplifications.

The procedure discussed above is not necessarily efficient, because critical pairs among various rules are being checked for again and again; an efficient implementation can be designed based on a version of the procedure given in [11],[16].

The results discussed in § 5.1 can be extended to almost-confluent Thue systems. Using the techniques and proofs developed there, we can show that for a given Thue system T , if there exists a finite almost-confluent Thue system T' , then the Knuth-Bendix completion procedure terminates with a reduced almost-confluent Thue system T'' equivalent to T .

Examples.

$$1. T = \{a \mid b, bab \rightarrow b\}.$$

After the first iteration of the completion procedure, three rules are added: $baa \rightarrow b$, $aab \rightarrow b$, and $bbb \rightarrow b$. In the next iteration, three rules are added: $aaa \rightarrow b$, $bba \rightarrow b$, $abb \rightarrow b$. Subsequently, the rule $aba \rightarrow b$ is added, which makes the final system almost-confluent. The result is

$$\{a \mid b, aaa \rightarrow b, aab \rightarrow b, aba \rightarrow b, baa \rightarrow b, \\ abb \rightarrow b, bab \rightarrow b, bba \rightarrow b, bbb \rightarrow b\}.$$

$$2. T = \{baa \mid aab, bab \rightarrow a, bb \rightarrow a\}.$$

First iteration: $aa \rightarrow a$, $ab \mid ba$ added and $baa \mid aab$ deleted.

Second iteration: $aba \rightarrow ab$ added.

After this, the system

$$\{aa \rightarrow a, bb \rightarrow a, bab \rightarrow a, aba \rightarrow ab, ab \mid ba\}$$

is almost-confluent.

6. Termination of the Knuth-Bendix procedure on parenthesized Thue systems. Unfortunately it is undecidable whether there exists a finite Church-Rosser Thue system equivalent to a finite Thue system T [27]. Although the results in the previous section ensure that for Thue systems for which this question is decided in the affirmative, the Knuth-Bendix procedure is guaranteed to terminate, there is no way to say a priori by examining a given Thue system, whether the Knuth-Bendix procedure will terminate. Below, we discuss a property of Thue systems that can be checked and for which the Knuth-Bendix procedure terminates.

We define

$$DS(T) = \{x \mid l_i \rightarrow^* x \text{ for some } l_i \rightarrow r_i \text{ in } T\}.$$

(The letters DS represent “derived strings.”)

Two not necessarily distinct strings x and y are called *nonoverlapping* if and only if there do not exist nonempty strings u , v , and w such that $x = uv$ and $y = vw$. This definition can be extended to a set of strings by requiring that every pair of identical and nonidentical strings from the set are nonoverlapping.

THEOREM. 6.1. *For a Thue system T , if $DS(T)$ is nonoverlapping, then the Knuth-Bendix procedure will terminate, generating a reduced lex-confluent system T' equivalent to T .*

Proof. Because of nonoverlapping of the left-hand sides of rules in T , all critical pairs generated in the first iteration must be due to the lhs of some rule being a substring of the lhs of another. These are less than the largest lhs of rules in T in the total ordering $<$ on strings being used. Furthermore, all critical pairs generated can be obtained by reducing lhs's of rules. The condition that all strings in $DS(T)$ are nonoverlapping implies strings in all critical pairs generated in any iteration are $<$ the greatest lhs in rules in T , which are finitely many. So, the extra rules that can be generated by the Knuth-Bendix completion procedure are only finitely many, implying that the Knuth-Bendix procedure will necessarily terminate. \square

A Thue system T is called *parenthesized* if and only if every string in T is properly parenthesized with respect to “(” and “).” Some examples of parenthesized strings are (a) , $(a b (c b) (b c) c)$, etc. This definition can be generalized so that two arbitrary strings can be used instead of “(” and “).”

For a parenthesized Thue system T , it is obvious that $DS(T)$ is nonoverlapping, so we have

COROLLARY 6.2. *For a parenthesized Thue system, the Knuth-Bendix completion procedure always terminates.*

Note that a term-rewriting system in which rules involve only ground terms is parenthesized, so it has the property that all its lhs's are nonoverlapping. So for ground-term-rewriting systems, the Knuth-Bendix procedure always terminates once we have an ordering on ground terms that satisfies the replacement and subterm properties (the analog for ground terms of the two properties of an ordering on strings discussed in §§ 2.1). As in parenthesized systems, critical pairs generated in the completion procedure are never going to be greater in size than the largest (with respect to the ordering on ground terms) lhs in a ground-term-rewriting system, and there are only finitely many such terms since the ordering is well founded. According to Dershowitz [7], Lankford was the first to observe that the completion procedure always terminates on ground-term-rewriting systems.

The condition stated above is a particular case of the following general condition: If the set of terms that can possibly serve as superpositions for generating critical pairs is finite, then the Knuth-Bendix completion procedure will always terminate, assuming that it does not abort because the two terms in a nontrivial critical pair cannot be made into a rule.

7. Conclusions. We have introduced the notion of a reduced Thue system. For reduced Thue systems, we have shown a number of properties. It was proved that if there exists a Church-Rosser system equivalent to a Thue system, then there is a unique reduced Church-Rosser Thue system equivalent to it. Using properties of reduced Church-Rosser systems, we have developed conditions under which a class of special Thue systems have equivalent finite Church-Rosser systems.

By using the fact that finite Church-Rosser (lex-confluent) Thue systems themselves have a canonical form, we have shown that if there exists a finite Church-Rosser (lex-confluent) system equivalent to a finite Thue system, then the Knuth-Bendix procedure is guaranteed to terminate with a reduced Church-Rosser (lex-confluent) system equivalent to the original system. The proof depends upon another crucial property, which is that all nontrivial critical pairs generated by the Knuth-Bendix completion procedure can be oriented to give rules; this property is easy to ensure for strings. We have also extended the completion procedure to generate reduced almost-confluent systems. This completion procedure is different from the completion procedures discussed in the literature, as in this case new rules are being added into the set of length-reducing rules, as well as into the set of length-preserving rules.

In addition, we have discussed two methods for showing the termination of the completion procedure. The first method uses the structure of the set of normal forms of a reduction system. For a class of reduction systems for which there cannot exist an infinite ascending chain of sets of reducible strings (by an

ascending chain, we mean that for every i , S_i properly contains S_{i+1}), the completion procedure would terminate because the sets of normal forms keep decreasing in every iteration of the completion procedure. The termination of the completion procedure on commutative Thue systems is shown using this method.

The second method is based on the property that can be informally stated as follows: For systems for which strings appearing in the critical pairs that can be generated by the Knuth-Bendix completion procedure constitute a finite set, the Knuth-Bendix procedure is guaranteed to terminate. The termination of the Knuth-Bendix procedure for ground rewriting systems turns out to be a corollary of this result.

To extend these results to term-rewriting systems, 1. a suitable canonical form for term-rewriting systems needs to be developed (the results in [21] are an attempt in that direction) and 2. orderings on terms have to be devised such that nontrivial critical pairs generated during the Knuth-Bendix completion procedure can always be oriented so that the Knuth-Bendix procedure does not have to be aborted because a critical pair cannot be converted to a rule. It will also be useful to extend the methods discussed above for showing termination of the completion procedure on term-rewriting systems. This would result in a nice characterization of a class of decidable theories.

Acknowledgments. We thank Ron Book, M.S. Krishnamoorthy, Bob McNaughton, Dave Musser and the referees for comments and suggestions which have improved the presentation of the paper.

REFERENCES

- [1] R. BOOK, *Confluent and other types of Thue systems*, J. Assoc. Comput. Mach., 29 (Jan. 1982), pp. 171-182.
- [2] ———, *A note on special Thue systems with a single defining relation*, Mathematical Systems Theory, 16 (1983), pp. 57-60.
- [3] ———, *Homogeneous Thue systems and the Church-Rosser property*, Discrete Mathematics, 48 (1984), pp. 137-145.
- [4] R. BOOK AND C. O'DUNLAIG, *Testing for the Church-Rosser property*, Theoretical Computer Science, 16 (1981), pp. 223-229.
- [5] B. BUCHBERGER AND R. LOOS, *Algebraic simplification*, Computing Suppl. 4, Springer Verlag (1982), pp. 11-43.
- [6] N. DERSHOWITZ, *Ordering for term rewriting systems*, Theoretical Computer Science, 17 (1982), pp. 279-301.
- [7] ———, *Applications of the Knuth-Bendix Completion Procedure*, Laboratory Operation, Aerospace Corporation, Aerospace Report No. ATR-83(8478)-2, 15 May 1983.
- [8] ———, *Existence and Construction of Rewrite Systems*, Laboratory Operation, Aerospace Corporation, Aerospace Report No. ATR-82(8478)-3, 1 December 1982.
- [9] S. EILENBERG AND M.P. SCHUTZENBERGER, *Rational sets in commutative monoids*, Journal of Algebra, 13 (1969), pp. 173-191.
- [10] G. HUET, *Confluent reductions: abstract properties and applications to term rewriting systems*, J. Assoc. Comput. Mach., 27 (1980), pp. 797-821.
- [11] ———, *A Complete Proof of Correctness of the Knuth-Bendix Completion Algorithm*, Journal of Computer and System Sciences, 23 (1981), pp. 11-21.
- [12] A. KANDRI-RODY AND D. KAPUR, *Computing the Grobner basis of a polynomial ideal over integers*, Third MACSYMA Users Conference, Schenectady, NY (July 1984), pp. 436-451.

- [13] D. KAPUR, M.S. KRISHNAMOORTHY, R. MCNAUGHTON, AND P. NARENDRAN, *An $O(T^3)$ algorithm for testing the Church-Rosser property of Thue systems*, to appear in Theoretical Computer Science.
- [14] D. KAPUR AND P. NARENDRAN, *Almost-Confluence and Related Properties of Thue Systems*, Report No. 83CRD258, General Electric Corporate Research and Development, Schenectady, NY, November, 1983.
- [15] — — —, *A finite Thue system with a decidable word problem and without equivalent finite canonical Thue system*, Theoretical Computer Science, 35 (1985), pp. 337-344.
- [16] D. KAPUR AND G. SIVAKUMAR, *Experiments and architecture of RRL, a Rewrite Rule Laboratory*, in Proceedings of an NSF Workshop on the Rewrite Rule Laboratory (6-9 Sept. 1983), General Electric Report 84GEN008, Schenectady, NY, April 1984, pp. 33-66.
- [17] D.E. KNUTH AND P.B. BENDIX, *Simple word problems in universal algebras*, in Computational Problems in Abstract Algebras (ed. J. Leech), Pergamon Press, 1970, pp. 263-297.
- [18] D.S. LANKFORD AND G. BUTLER, *Experiments with Computer Implementations of Procedures which Often Derive Decision Algorithms for the Word Problem in Abstract Algebra*, Technical Report, MTP-7, Louisiana Tech. University, August 1980.
- [19] P. LESCANNE, *Computer Experiments with the REVE Term Rewriting System Generator*, Proc. 10th ACM Symposium on Principles of Programming Languages (1983), pp. 99-68.
- [20] E. MAYR AND A.R. MEYER, *The Complexity of the Word Problems for Commutative Semigroups and Polynomial Ideals*, MIT-LCS-TM-199, Lab. for Computer Science, MIT, Cambridge, June 1981.
- [21] Y. METIVIER, *About the rewriting systems produced by the Knuth-Bendix completion algorithm*, Information Processing Letters, 16 (1983), pp. 31-34.
- [22] D.R. MUSSER, *Abstract data type specification in the AFFIRM system*, IEEE Trans. Software Engg., Vol. 6 (1980), pp. 24-31.
- [23] — — —, *On proving inductive properties of abstract data types*, Proc. 7th ACM Symposium on Principles of Programming Languages (1980), pp. 154-162.
- [24] D.R. MUSSER AND D. KAPUR, *Rewrite rule theory and abstract data type analysis*, in Computer Algebra: EUROCAM '82 (ed. J. Calmet), Lecture Notes in Computer Science, Springer Verlag, Vol. 144, (1982) pp. 77-90.
- [25] P. NARENDRAN, *Church-Rosser and related Thue systems*, Ph.D. Dissertation, Dept. of Mathematical Sciences, Rensselaer Polytechnic Institute, Troy, NY (1984).
- [26] M. NIVAT (with M. Benois), *Congruences parfaites et quasi- parfaites*, Seminaire Dubreil, 25^e Annee (1971-1972) 7-01-09.
- [27] C. O'DUNLAING *Undecidable questions related to Church-Rosser Thue systems*, Theoretical Computer Science, 23 (1983), pp. 339-345.