SIAM J. COMPUT.
Vol. 13, No. 1, February 1984

© 1984 Society for Industrial and Applied Mathematics
0097-5397/84/1301-0001 $01.25/0

ON THE EXISTENCE OF COOK SEMANTICS*

ARIE DE BRUIN†

Abstract. In [SIAM J. Comput., 7 (1978), pp. 70–90] Cook defines the operational semantics of a programming language in the following way: a function is introduced which takes a program R and a state σ and yields a possibly infinite row of intermediate states as a result. This row is meant to be the trace resulting from executing program R starting in state σ. This function is characterized by a number of equations. However it is not immediately clear whether these equations have a solution. In this paper we show for a simple language, the most sophisticated feature of which is that it has parameterless procedures, that the corresponding equations have a unique solution. The techniques used here can also be applied to other languages described in the same way, for instance to the language in Cook's paper.

Key words. operational semantics, Cook semantics, fixed points, continuation semantics, recursive definitions, denotational semantics

1. The problem. In this paper we investigate a certain way of defining operational semantics of programming languages, which has been introduced by Cook in his paper on soundness and completeness [6]. Cook remarks that this semantics has been derived from one of the operational semantics studied in Lauer's thesis [10], and also in Hoare and Lauer [7], which is a condensed version of the thesis. This style of definition has later on been employed by de Bakker in his book on the theory of program correctness [3].

The technique is as follows: a meaning function **Comp** is described which takes a program and an initial machine state and yields a row of states as a result. This row gives the trace left by evaluating the program starting in the initial state. A terminating computation yields a finite row, and if evaluation does not terminate then the outcome is an infinite row.

We will study Cook semantics using a simple language. Before giving its syntax we introduce some notational conventions.

Rows will be indicated by angular brackets. For instance we have $\langle x_1, \cdots, x_n\rangle$ which denotes a finite row of n elements, and $\langle x_1, x_2, \cdots\rangle$ which denotes an infinite row. The empty row is denoted by $\langle\,\rangle$. Function application associates to the left, that is $fabc$ is an abbreviation of $((f(a))(b))(c)$. Correspondingly, the $\rightarrow$-operator used in forming function domains associates to the right. The above function f should have functionality definition $f{:}A \rightarrow B \rightarrow C \rightarrow D$, which should read as $f{:}A \rightarrow (B \rightarrow (C \rightarrow D))$.

We next describe the syntax of the language. We distinguish the following syntactic classes:

$P \in$ **Pvar**	Procedure variables.
$A \in$ **Atst**	Atomic statements. The structure of these statements is not specified further, but think of assignments.
$B \in$ **Bexp**	Boolean expressions. These are also considered to be atomic building blocks.
$R \in$ **Prog**	Programs. These have the form $\langle E \mid S\rangle$ and must be closed, i.e. all procedure variables in E and S are declared in E.
$E \in$ **Decl**	Declarations. These have the form $\langle P_1 \Leftarrow S_1, \cdots, P_n \Leftarrow S_n\rangle$ where all P_i are different.
$S \in$ **Stat**	Statements. This class is defined by the following *BNF*-like syntax.
$S \in$ **Stat**	Statements. This class is defined by the following *BNF*-like syntax.
	$S ::= A \mid P \mid$ if B then S_1 else $S_2 \mid S_1; S_2$.

---

* Received by the editors June 10, and in final revised form November 1, 1982.

† Faculty of Economics, Erasmus University, Rotterdam, the Netherlands.

We now turn to the semantics. There are the following semantic classes.

$\sigma \in \Sigma$ States. The internal structure of states is not specified. Notice that Σ is a set, not a cpo. There is for instance no such thing as $\perp$ in Σ.

$\tau \in \Sigma^\infty$ Rows of states. We define $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. Σ^* contains the finite sequences and the empty row and Σ^ω the infinite ones.

We define the following operators on rows of states.

$^\wedge$ Concatenation, defined by the axioms:

$\tau_1{}^\wedge\tau_2 = \tau_1$ for all $\tau_1 \in \Sigma^\omega$

$\langle\,\rangle^\wedge\tau = \tau^\wedge\langle\,\rangle = \tau$

$\langle\sigma_1, \cdots, \sigma_n\rangle^\wedge\langle\sigma'_1, \cdots, \sigma'_k\rangle = \langle\sigma_1, \cdots, \sigma_n, \sigma'_1, \cdots, \sigma'_k\rangle$

$\langle\sigma_1, \cdots, \sigma_n\rangle^\wedge\langle\sigma'_1, \cdots\rangle = \langle\sigma_1, \cdots, \sigma_n, \sigma'_1, \cdots\rangle$

κ Last element extraction, defined by

$\kappa\langle\sigma_1, \cdots, \sigma_n\rangle = \sigma_n$

$\kappa\langle\,\rangle = \kappa\tau = \bar{\sigma}$ for all $\tau \in \Sigma^\omega$, where $\bar{\sigma}$ is an arbitrary (but fixed from now on) element of Σ.

Finally we distinguish the following elementary valuations.

A: **Atst** $\to \Sigma \to \Sigma$, Meaning of atomic statements. Notice that atomic statements always terminate.

B: Bexp $\to \Sigma \to \{tt, ff\}$, Meaning of boolean expressions.

As the internal structure of **Atst** and **Bexp** has not been specified, we cannot do more than postulate the existence of functions **A** and **B** with functionalities as above.

We now have enough tools to formulate the equations which are intended to define a function **Comp**: **Prog** $\to \Sigma \to \Sigma^\infty$.

$\mathbf{Comp}\langle E|A\rangle\sigma = \langle \mathbf{A}A\sigma\rangle$

$\mathbf{Comp}\langle E|P_i\rangle\sigma = \langle\sigma\rangle^\wedge\mathbf{Comp}\langle E|S_i\rangle\sigma$, with $P_i \Leftarrow S_i$ in E.

$$\mathbf{Comp}\langle E|\text{ if } B \text{ then } S_1 \text{ else } S_2\rangle\sigma = \begin{cases}\langle\sigma\rangle^\wedge\mathbf{Comp}\langle E|S_1\rangle\sigma, & \text{if } \mathbf{B}B\sigma = tt\\ \langle\sigma\rangle^\wedge\mathbf{Comp}\langle E|S_2\rangle\sigma, & \text{otherwise}\end{cases}$$

$\mathbf{Comp}\langle E|S_1; S_2\rangle\sigma = \langle\sigma\rangle^\wedge\tau^\wedge\mathbf{Comp}\langle E|S_2\rangle(\kappa\tau)$, where $\tau = \mathbf{Comp}\langle E|S_1\rangle\sigma$.

In the sequel we will refer to this set of equations as CE, which is an abbreviation of "the Cook equations." Now there are some questions to be answered. Does there exist a function with the above properties? If so, is this function unique? We cannot provide the answers immediately because the above equations can be interpreted as a recursive definition which is not inductive.

Cook also was aware of these questions as the following quotation from [6] shows: "The definition is recursive, in the sense that **Comp** appears on the right side of the clauses. This may appear ironic in a paper on program verification, since one of the important issues in programming language semantics is interpreting recursively defined procedures. However, one does not have to understand recursive procedures in general in order to understand this specific definition. Suffice it to say that we intend **Comp** to be evaluated by "call by name," in the sense that occurrences of **Comp** are to be replaced successively by their meanings according to the appropriate clauses in the definition."

In this paper we will provide the answer to the above questions; there is a unique total function which satisfies the equations. We will show this in four different ways. The first idea is to derive from the recursive definition an inductive one which defines the elements of the outcome of **Comp** one by one. This is treated in § 2. The other techniques are based on a standard idea from denotational semantics: transform recursion into iteration. From the Cook equations an operator can be derived, and iteration of this operator yields a sequence of approximations which should tend to a limit, a function satisfying CE. In order to be able to talk about convergence, the relevant semantical domains are turned into cpo's.

However these techniques cannot be applied here straightforwardly, because the approximations will generally not converge. This phenomenon is analyzed in § 3. To make the basic idea work we have to extend the standard approach somehow and this can be done into three directions. First of all we can enrich the domain Σ^∞ by adding to it a class of finite rows marked as "not yet complete." Secondly, we can rephrase the Cook equations such that the standard approach does work. Lastly, we can make use of the fact that Σ^∞ has more structure than a cpo, it is a complete metric space. These solutions will be treated in § 4–6.

In the sequel we will need the following lemma which gives information on all total functions satisfying CE. The lemma states that a definition through a set of equations like CE is independent of the particular way we defined $\kappa\tau$ for $\tau = \langle\,\rangle$ or $\tau \in \Sigma^\omega$. This holds because CE is such that in it κ is never applied to $\langle\,\rangle$, and if κ is applied to an element of Σ^ω, then its value is irrelevant because it will be used only to determine a row which is appended to an infinite row, which means that it will be neglected.

LEMMA 1.1. *For every total function Φ in* $\mathbf{Prog} \to \Sigma \to \Sigma^\infty$ *which satisfies* CE the following holds.

1. *For all R and σ we have $\Phi R\sigma \neq \langle\,\rangle$.*
2. *If we construct a set of equations* CE′ *which is like* CE, *except for the fact that it uses another last element extraction function κ' which differs from κ only when applied to $\langle\,\rangle$ or elements from Σ^ω, then Φ is also a solution of* CE′.

2. A straightforward solution. The idea is the following. We define a new function $\mathbf{C}$ which is like $\mathbf{Comp}$ but takes besides R and σ an extra argument, a natural number n, and which yields an element from Σ. This element should then be the nth element of the row $\mathbf{Comp}R\sigma$. Now it is possible to give an inductive definition of $\mathbf{C}$. First of all we have to introduce an extra element Ω ("undefined") because in the setup, as proposed here, it is possible to ask for the third element of a row of two elements. In such cases we then deliver Ω. We define

DEFINITION 2.1. The function $\mathbf{C}: \mathbf{Prog} \to \Sigma \to \mathbb{N} \to \Sigma \cup \{\Omega\}$ is defined by induction on n as follows:

$$\mathbf{C}\langle E|A\rangle\sigma n = \begin{cases} \mathbf{A}A\sigma & \text{if } n = 1, \\ \Omega & \text{otherwise;} \end{cases}$$

$$\mathbf{C}\langle E|P_i\rangle\sigma n = \begin{cases} \sigma & \text{if } n = 1, \\ \mathbf{C}\langle E|S_i\rangle\sigma(n-1) & \text{otherwise, where } P_i \Leftarrow S_i \text{ occurs in } E; \end{cases}$$

$$\mathbf{C}\langle E|\text{if } B \text{ then } S_1 \text{ else } S_2\rangle\sigma n = \begin{cases} \sigma & \text{if } n = 1, \\ \mathbf{C}\langle E|S_1\rangle\sigma(n-1) & \text{if } n \neq 1 \text{ and } \mathbf{B}B\sigma = tt, \\ \mathbf{C}\langle E|S_2\rangle\sigma(n-1) & \text{otherwise;} \end{cases}$$

$$\mathbf{C}\langle E|S_1; S_2\rangle\sigma n = \begin{cases} \sigma & \text{if } n = 1, \\ \mathbf{C}\langle E|S_1\rangle\sigma(n-1) & \text{if } n \neq 1 \text{ and } \mathbf{C}\langle E|S_1\rangle\sigma(n-1) \neq \Omega, \\ \mathbf{C}\langle E|S_2\rangle(\mathbf{C}\langle E|S_1\rangle\sigma k)(n-k-1) & \text{if } n \neq 1,\ \mathbf{C}\langle E|S_1\rangle\sigma(n-1) = \Omega, \\ & \text{and } V := \{m | \mathbf{C}\langle E|S_1\rangle\sigma m \neq \Omega \text{ and} \\ & \mathbf{C}\langle E|S_1\rangle\sigma(m+1) = \Omega \text{ and } m < n\} \neq \varnothing, \\ & \text{where } k = \min V, \\ \Omega, & \text{otherwise.} \end{cases}$$

Because we had to be careful about little details, the above definition has an awkward appearance. It can be made more tractable by realizing that for all R and σ the infinite row $\langle \mathbf{C}R\sigma k\rangle_k$ contains either only elements from Σ, or has the form $\langle \sigma_1, \sigma_2, \cdots, \sigma_k, \Omega, \Omega, \Omega, \cdots\rangle$. This observation enables us to rephrase the case $S_1; S_2$ in the above definition as follows.

LEMMA 2.2.

$$\mathbf{C}\langle E|S_1; S_2\rangle\sigma n = \begin{cases} \sigma & \textit{if } n = 1, \\ \mathbf{C}\langle E|S_1\rangle\sigma(n-1) & \textit{if } \mathbf{C}\langle E|S_1\rangle\sigma(n-1) \neq \Omega \textit{ and } n \neq 1, \\ \mathbf{C}\langle E|S_2\rangle(\mathbf{C}\langle E|S_1\rangle\sigma k)(n-k-1) & \textit{otherwise, where } k \textit{ is such that} \\ & \mathbf{C}\langle E|S_1\rangle\sigma k \neq \Omega \textit{ and} \\ & \mathbf{C}\langle E|S_1\rangle\sigma(k+1) = \Omega. \end{cases}$$

Now, the function **Comp** defined by

$$\mathbf{Comp}R\sigma = \begin{cases} \langle \mathbf{C}R\sigma 1, \cdots, \mathbf{C}R\sigma n\rangle & \text{if } \mathbf{C}R\sigma n \neq \Omega \text{ and } \mathbf{C}R\sigma(n+1) = \Omega, \\ \langle \mathbf{C}R\sigma 1, \mathbf{C}R\sigma 2, \cdots\rangle & \text{otherwise} \end{cases}$$

satisfies CE, as one can check straightforwardly.

Finally, we show that there is exactly one total function satisfying CE by the following argument. For any function **Comp** and for any R, σ and n we can calculate, using only the clauses from CE, the nth element from the row $\mathbf{Comp}R\sigma$, like we have done in Definition 2.1. So we have that the equations CE determine, for every R and σ, every element from the row $\mathbf{Comp}R\sigma$, that is, this row must be unique, that is **Comp** must be unique. Note that the above reasoning would no longer be valid if we allowed partial functions in $\mathbf{Prog} \to \Sigma \to \Sigma^\infty$ to be solutions of CE.

3. There is a problem if we try to use the fixed point approach. It is tempting to try to use fixed point theory to answer the questions raised in § 1, because any solution of CE will be a fixed point on the operator $\Psi: D \to D$, with $D = \mathbf{Prog} \to \Sigma \to \Sigma^\infty$ defined by

$$\begin{aligned} \Psi = \lambda\Phi.\lambda R.\lambda\sigma. \\ & R \equiv \langle E|A\rangle \to \langle \mathbf{A}A\sigma\rangle, \\ & R \equiv \langle E|P_i\rangle \to \langle\sigma\rangle^\wedge\Phi\langle E|S_i\rangle\sigma, \\ & R \equiv \langle E|\text{if } B \text{ then } S_1 \text{ else } S_2\rangle \\ & \qquad \to (\mathbf{B}B\sigma = tt \to \langle\sigma\rangle^\wedge\Phi\langle E|S_1\rangle\sigma, \langle\sigma\rangle^\wedge\Phi\langle E|S_2\rangle\sigma), \\ & R \equiv \langle E|S_1, S_2\rangle \to \langle\sigma\rangle^\wedge\Phi\langle E|S_1\rangle\sigma^\wedge\Phi\langle E|S_2\rangle(\kappa(\Phi\langle E|S_1\rangle\sigma)). \end{aligned}$$

Now it is a well-known fact from denotational semantics ([12], [13], [14]; see also [15] or [3] which both give an introduction to the subject) that Ψ has a least fixed point $\mu\Psi$ if this operator is continuous. In that case $\mu\Psi$ equals the lub of the chain $\perp \sqsubseteq \Psi\perp \sqsubseteq \Psi(\Psi\perp) \sqsubseteq \cdots$.

So, if we manage to make D a cpo such that Ψ is continuous then we obtain the required existence result immediately. Again, it is well known that D is a cpo if there is an ordering $\sqsubseteq$ on Σ^∞ which makes this set a cpo. Now the intuition behind $\tau_1 \sqsubseteq \tau_2$ is that τ_2 contains more information than τ_1, or that τ_2 is a better approximation of some final result than τ_1. A technique for turning a set into a cpo that is often used is to make this set a flat cpo. That is, add a totally undefined element $\perp$ to it and define $\tau_1 \sqsubseteq \tau_2$ iff $\tau_1 = \tau_2$ or $\tau_1 = \perp$.

However, this construction is not suited for our purposes, because we obtain a least fixed point $\mu\Psi$ which yields the right result for terminating processes, but which

yields $\perp$ for nonterminating processes. By way of an example we will evaluate some elements of the chain $\perp \sqsubseteq \Psi\perp \sqsubseteq \Psi^2\perp \sqsubseteq \cdots$ approximating $\mu\Psi$, applied to the program $\langle P|P \Leftarrow P\rangle$:

1. $\perp\langle P|P \Leftarrow P\rangle\sigma = \perp$
2. $(\Psi\perp)\langle P|P \Leftarrow P\rangle\sigma = \langle\sigma\rangle^\wedge\perp\langle P|P \Leftarrow P\rangle\sigma = \langle\sigma\rangle^\wedge\perp = \perp$
3. $(\Psi^2\perp)\langle|P \Leftarrow P\rangle\sigma = \langle\sigma\rangle^\wedge(\Psi\perp)\langle P|P \Leftarrow P\rangle\sigma = \langle\sigma\rangle^\wedge\langle\sigma\rangle^\wedge\perp\langle P|P \Leftarrow P\rangle\sigma$
   $= \langle\sigma\rangle^\wedge\langle\sigma\rangle^\wedge\perp = \perp$
4. $(\Psi^3\perp)\langle P|P \Leftarrow P\rangle\sigma = \langle\sigma\rangle^\wedge(\Psi^2\perp)\langle P|P \Leftarrow P\rangle\sigma$
   $= \langle\sigma\rangle^\wedge\langle\sigma\rangle^\wedge\langle\sigma\rangle^\wedge\perp\langle P|P \Leftarrow P\rangle\sigma = \langle\sigma\rangle^\wedge\langle\sigma\rangle^\wedge\langle\sigma\rangle^\wedge\perp = \perp$ etc.

The problem is that the ordering in a flat cpo is not refined enough: an approximation τ_1 of a final answer τ ($\tau_1 \sqsubseteq \tau$) contains either all information ($\tau_1 = \tau$) or no information at all ($\tau_1 = \perp$). Now because all finite approximations of an infinite row are necessarily unequal to this row we must have that all these approximations are equal to $\perp$. That is we get a chain $\perp \sqsubseteq \perp \sqsubseteq \cdots$ with lub $\perp$ and this is not what we want.

This analysis also shows a way out. What the sequence of approximations given above should do is yield longer and longer initial segments of the final outcome. That is, the ordering should be such that $\langle\sigma\rangle \sqsubseteq \langle\sigma, \sigma\rangle \sqsubseteq \langle\sigma, \sigma, \sigma\rangle \sqsubseteq \cdots$ is a chain with the natural lub $\langle\sigma, \sigma, \sigma, \cdots\rangle$. This leads us to trying the prefix ordering on Σ^∞: $\tau_1 \sqsubseteq \tau_2$ iff τ_1 is a prefix of τ_2. One easily checks that Σ^∞ with this ordering is a cpo with the empty row $\langle\,\rangle$ as bottom element. This ordering yields a correct approximation sequence for the program $\langle P|P \Leftarrow P\rangle$ as one easily can check. However, this approach does not work in general because Ψ is not continuous under this ordering. This stems from the fact that the operators κ and $^\wedge$ are not continuous, not even monotonic under the prefix ordering. For instance, $\langle\sigma_1\rangle \sqsubseteq \langle\sigma_1, \sigma_2\rangle$ but $\kappa\langle\sigma_1\rangle = \sigma_1$ and $\kappa\langle\sigma_1, \sigma_2\rangle = \sigma_2$ might very well be incomparable.

We can also show in a less technical way that the new approach does not work. Consider the sequence $\langle \perp (= \lambda R.\lambda\sigma.\langle\,\rangle), \Psi\perp, \Psi^2\perp, \cdots\rangle$ and apply some of the elements thereof to the program $R = \langle E|P; A_2\rangle$ (where $E = \langle P \Leftarrow A_1\rangle$) and initial state σ. We get

$$\perp R\sigma = \langle\,\rangle, \qquad (\Psi\perp)R\sigma = \langle\sigma\rangle, \qquad (\Psi^2\perp)R\sigma = \langle\sigma, \sigma, \mathbf{A}A_2\sigma\rangle,$$

$$(\Psi^3\perp)R\sigma = \langle\sigma, \sigma, \mathbf{A}A_1\sigma, \mathbf{A}A_2(\mathbf{A}A_1\sigma)\rangle,$$

and it follows that $\Psi^2\perp \not\sqsubseteq \Psi^3\perp$. Therefore the prefix ordering on Σ^∞ is such that the sequence $\langle \perp, \Psi\perp, \Psi^2\perp, \cdots\rangle$ is not a chain, and thus Ψ cannot be continuous.

If we investigate what went wrong here, we see that in evaluating $(\Psi^2\perp)R\sigma$ we apply the last element function κ to a row of states which is not yet finished; that is, we start evaluating A_2 "too early," namely in a state σ which is not the final state resulting from evaluation of P. This analysis suggests two solutions for the difficulty. The first one is to enlarge Σ^∞ so that it contains also rows of states which are marked as "not yet completed" and to let the operators κ and $^\wedge$ act in the "right" (continuous) manner on these rows. Another possibility is to rewrite Ψ in such a way that it does not use the noncontinuous operators κ and $^\wedge$ any more. Finally we observe that, though the above approximation sequence is not a chain, the right outcome has been obtained in the end. This suggests that Ψ might be continuous if we would use a more subtle notion of continuity. The next three sections will be devoted to a discussion of these possibilities.

4. Adding unfinished rows to Σ^∞. We observed that in Σ^∞ finished and unfinished rows of states must be distinguished. We will arrange this as follows: a row $\langle\sigma_1, \cdots, \sigma_n\rangle$ will be marked as unfinished by adding the element $\perp$ to it, so that we get $\langle\sigma_1, \cdots, \sigma_n, \perp\rangle$. Notice that only finite rows can possibly be unfinished; infinite rows,

which model nonterminating computations, cannot contain more information than they already do. The ordering $\langle\sigma_1, \cdots, \sigma_n, \perp\rangle \sqsubseteq \tau$ iff $\langle\sigma_1, \cdots, \sigma_n\rangle$ is a prefix of τ is natural. Furthermore, $\hat{}$ should not append its second argument if its first argument is an unfinished row. All this leads to the following list of definitions and properties.

DEFINITION 4.1.

1. $\Sigma^{\sim} = \Sigma^{*\perp} \cup \Sigma^{\infty}$, with Σ^{∞} as before, and where $\Sigma^{*\perp}$ is the set of all rows consisting of zero or more states followed by the symbol $\perp$.

2. For $\tau_1, \tau_2 \in \Sigma^{\sim}$ we define $\tau_1 \sqsubseteq \tau_2$ iff either $\tau_1 = \tau_2$ or $\tau_1 = \langle\sigma_1, \cdots, \sigma_n, \perp\rangle \in \Sigma^{*\perp}$ and $\langle\sigma_1, \cdots, \sigma_n\rangle$ is a prefix of τ_2.

3. $\Sigma_\perp = \Sigma \cup \{\perp\}$, the flat cpo derived from Σ.

4. $\kappa : \Sigma^{\sim} \to \Sigma_\perp$ is defined by

$$\kappa(\tau) = \begin{cases} \perp & \text{if } \tau \in \Sigma^{\infty}, \tau \in \Sigma^{*\perp} \text{ or } \tau = \langle\,\rangle, \\ \sigma_n & \text{if } \tau = \langle\sigma_1, \cdots, \sigma_n\rangle \in \Sigma^* \backslash \{\langle\,\rangle\}. \end{cases}$$

5. $\hat{} : \Sigma^{\sim} \times \Sigma^{\sim} \to \Sigma^{\sim}$ is defined by

$$\tau_1 \hat{}\, \tau_2 = \begin{cases} \tau_1 \quad \text{if } \tau_1 \in \Sigma^{\omega} \cup \Sigma^{*\perp}, \\ \langle\sigma_1, \cdots, \sigma_n, \sigma_1', \cdots, \sigma_k'\rangle \quad \text{if } \tau_1 = \langle\sigma_1, \cdots, \sigma_n\rangle \in \Sigma^*, \tau_2 = \langle\sigma_1', \cdots, \sigma_k'\rangle \in \Sigma^*, \\ \langle\sigma_1, \cdots, \sigma_n, \sigma_1', \cdots, \sigma_k', \perp\rangle \\ \qquad\qquad \text{if } \tau_1 = \langle\sigma_1, \cdots, \sigma_n\rangle \in \Sigma^*, \tau_2 = \langle\sigma_1', \cdots, \sigma_k', \perp\rangle \in \Sigma^{*\perp}, \\ \langle\sigma_1, \cdots, \sigma_n, \sigma_1', \cdots\rangle \quad \text{if } \tau_1 = \langle\sigma_1, \cdots, \sigma_n\rangle \in \Sigma^*, \tau_2 = \langle\sigma_1', \cdots\rangle \in \Sigma^{\omega}. \end{cases}$$

LEMMA 4.2.

1. $(\Sigma^{\sim}, \sqsubseteq)$ *is a cpo with smallest element* $\langle\perp\rangle$.

2. κ *and* $\hat{}$ *are continuous.*

Now that we have added the element $\perp$ to Σ we have to adapt the definition of a little bit.

DEFINITION 4.3. $\Psi : D \to D$, with $D = \mathbf{Prog} \to \Sigma_\perp \to_s \Sigma^{\sim}$ is defined by

$$\begin{aligned} \Psi = \lambda\Phi.\lambda R.\lambda\sigma.\sigma = \perp \to \langle\perp\rangle, & \\ & R \equiv \langle E|A\rangle \to \langle\mathbf{A}A\sigma\rangle, \\ & R \equiv \langle E|P_i\rangle \to \langle\sigma\rangle\hat{}\,\Phi\langle E|S_i\rangle\sigma, \\ & R \equiv \langle E|\text{ if } B \text{ then } S_1 \text{ else } S_2\rangle \\ & \qquad \to (\mathbf{B}B\sigma = tt \to \langle\sigma\rangle\hat{}\,\Phi\langle E|S_i\rangle\sigma, \langle\sigma\rangle\hat{}\,\Phi\langle E|S_2\rangle\sigma), \\ & R \equiv \langle E|S_1; S_2\rangle \to \langle\sigma\rangle\hat{}\,\Phi\langle E|S_1\rangle\sigma\hat{}\,\Phi\langle E|S_2\rangle(\kappa(\Phi\langle E|S_1\rangle\sigma)). \end{aligned}$$

Remarks.

1. The expression $\Sigma_\perp \to_s \Sigma^{\sim}$ denotes the cpo of all strict functions from $\Sigma_\perp$ to $\Sigma^{\sim}$, that is, all functions f for which $f\perp = \langle\perp\rangle$. This precaution is needed because otherwise Ψ would not be continuous.

2. One easily checks that the operator Ψ has the functionality as announced. That is, for all Φ in $\mathbf{Prog} \to \Sigma_\perp \to_s \Sigma^{\sim}$, $R \in \mathbf{Prog}$ and $\sigma \in \Sigma_\perp$, we have that $\Psi\Phi R\sigma \in \Sigma^{\sim}$ (that is, only the last element might be $\perp$); and also for all $\Phi \in D$, $R \in \mathbf{Prog}$ we have that $\Psi\Phi R \perp = \langle\perp\rangle$ (i.e. $\Psi\Phi R$ is strict again).

3. The fact that Ψ is a continuous operator in $D \to D$ and thus that $\mu\Psi$ exists, can proved straightforwardly.

The key lemma is

LEMMA 4.4. *For all R and $\sigma \neq \perp$ we have* $(\mu\Psi)R\sigma \in \Sigma^{\infty}$.

Proof. By contradiction. Suppose the assertion is not true. We then would have some R and $\sigma \neq \bot$ for which $(\mu\Psi)R\sigma \in \Sigma^{*\bot}$. Now $(\mu\Psi)R\sigma = \bigsqcup_i ((\Psi^i \bot)R\sigma)$ and therefore we would have that for all $i\cdot$, $\tau_i := (\Psi^i \bot)R\sigma \in \Sigma^{*\bot}$. Now intuitively $\tau_i \in \Sigma^{*\bot}$ means that this approximation of evaluation of R in σ is not good enough, because this row is not yet completed. This suggests that there is a better approximation in the chain $\langle(\Psi^i \bot)R\sigma\rangle_i$ and in fact this holds already for the next element in the chain: we have $\tau_i \in \Sigma^{*\bot} \Rightarrow \tau_{i+1} \neq \tau_i$ (to be proved by induction on i). Thus we have the following situation: $(\mu\Psi)R\sigma$ is the lub of a strictly increasing chain $\bot R\sigma \sqsubsetneq (\Psi\bot)R\sigma \sqsubsetneq \cdots$ with all $(\Psi^k \bot)R\sigma \in \Sigma^{*\bot}$. Now we have a contradiction, for such a chain must have a lub in Σ^ω.

THEOREM 4.5. *$\mu\Psi$, restricted to the domain* $\mathbf{Prog} \to \Sigma \to \Sigma^\infty$, *is the unique solution of* CE.

Proof.

1. Notice that we cannot state that $\mu\Psi$ is a solution of CE, because $\mu\Psi$ is an element of $\mathbf{Prog} \to \Sigma_\bot \to_s \Sigma^\sim$ and as such it can never be a solution of CE. Notice also that we can restrict $\mu\Psi$ to the domain $\mathbf{Prog} \to \Sigma \to \Sigma^\infty$ only by virtue of Lemma 4.4.

2. [$\mu\Psi$ is a solution.] First compare the definition of κ and $\wedge$ from § 1 with the ones in Definition 4.1 and observe that the restriction of $\wedge$ (according to 4.1.5) to $\Sigma^\infty \times \Sigma^\infty$ is the same operator as $\wedge$ in § 1, while the restriction of κ to Σ^∞ is almost the same, the only difference being the cases $\kappa\tau$ where $\tau \in \Sigma^\omega$ or $\tau = \langle\,\rangle$. If these operators would be the same then we were ready, because from Definition 4.3 we see that κ and $\wedge$ are applied only to arguments of the form $(\mu\Psi)R\sigma$ and these are in Σ^∞ by Lemma 4.4. However the values of $\kappa\langle\,\rangle$ and $\kappa\tau$ for $\tau \in \Sigma^\omega$ are irrelevant, because the fixed points of Ψ have the same properties as the ones given by Lemma 1.1 for the solutions of CE.

3. [$\mu\Psi$ is the only fixed point Ψ.] Suppose not. Then there would be a bigger fixed point Φ, that is, there would be an R and σ such that $(\mu\Psi)R\sigma \sqsubsetneq \Phi R\sigma$. This is impossible, however, because by Lemma 4.4 $(\mu\Psi)R\sigma \in \Sigma^\infty$ which means that $(\mu\Psi)R\sigma$ is a maximal element in $\Sigma^\sim$.

4. [$\mu\Psi$ is the only solution of CE.] Suppose there would be another function $\mathbf{C}: \mathbf{Prog} \to \Sigma \to \Sigma^\infty$ satisfying CE. We can extend this function to a function $\mathbf{C}': \mathbf{Prog} \to \Sigma_\bot \to_s \Sigma^\sim$ by defining $\mathbf{C}'R\sigma = \mathbf{C}R\sigma$ if $\sigma \in \Sigma$ and $\langle\bot\rangle$ if $\sigma = \bot$. One easily checks that $\mathbf{C}'$ is a fixed point of Ψ, but then $\mathbf{C}' = \mu\Psi$, a contradiction.

5. The continuation approach. In § 3 we remarked that the direct fixed point approach failed due to the fact that the operators κ and $\wedge$ are not continuous. In this section we will find a way out of this problem by restructing CE in such a way that these operators are not used any more, or at least not in a noncontinuous way. The problem stems from the clause on constructs of the form $\langle E|S_1; S_2\rangle$. The idea that we will pursue is to use continuation semantics instead of direct semantics.

Direct semantics defines the meaning of a construct in terms of the rows of states that correspond to evaluation of the constituents of the construct. Therefore the operators κ and $\wedge$ have to be used: the meaning of $\langle E|S_1; S_2\rangle$ is obtained by concatenating the rows of states corresponding to the meanings of $\langle E|S_1\rangle$ and $\langle E|S_2\rangle$. Continuation semantics uses another idea: the meaning of a construct is the row of states which is the result of evaluating the construct itself followed by evaluation of the rest of the program of which the construct is supposed to be a part. Of course, the effect of evaluation of the rest of the program cannot be obtained from the construct itself; so we have to give the meaning function another argument, a continuation which will be a function from states to rows of states describing the effect of the rest of the program. One can view this continuation as a coding of the row of statements which

are to be evaluated once the statement under consideration has been worked through. More information on continuation semantics can be found in [15].

In this setup we do not have to concatenate two rows any more while defining the meaning of $\langle E|S_1; S_2\rangle$ because the effect of evaluating S_2 can be caught by changing the continuation which describes what will happen once the whole construct has been evaluated into a continuation which describes the effect of first evaluating S_2 and then applying the original continuation. This new formed continuation is given as an argument to $\mathbf{Comp}\langle E|S_1\rangle$. All this leads to the following operator.

DEFINITION 5.1. The operator $\Psi: D \to D$, with $D = \mathbf{Prog} \to [\Theta \to \Theta]$ and $\Theta = \Sigma \to \Sigma^\infty$ is defined by

$$\Psi = \lambda\Phi.\lambda R.\lambda\theta.\lambda\sigma.$$

$$R \equiv \langle E|A\rangle \to \langle \mathbf{A}A\sigma\rangle^\wedge\theta(\mathbf{A}A\sigma),$$

$$R \equiv \langle E|P_i\rangle \to \langle\sigma\rangle^\wedge\Phi\langle E|S_i\rangle\theta\sigma,$$

$$R \equiv \langle E|\text{if } B \text{ then } S_1 \text{ else } S_2\rangle$$

$$\to (\mathbf{B}B\sigma = tt \to \langle\sigma\rangle^\wedge\Phi\langle E|S_1\rangle\theta\sigma, \langle\sigma\rangle^\wedge\Phi\langle E|S_2\rangle\theta\sigma),$$

$$R \equiv \langle E|S_1; S_2\rangle \to \langle\sigma\rangle^\wedge\Phi\langle E|S_1\rangle\{\Phi\langle E|S_2\rangle\theta\}\sigma.$$

Remarks.

1. Notice that the operator κ is not used any more. We do use the concatenation operator, but only in a continuous way: $\lambda\tau.\langle\sigma\rangle^\wedge\tau$ is continuous with respect to the prefix order on Σ^∞.

2. The fourth clause of the definition can be interpreted as follows: evaluating $\langle E|S_1; S_2\rangle$ followed by evaluation according to θ amounts to evaluation of $\langle E|S_1\rangle$ followed by [evaluation of $\langle E|S_2\rangle$ followed by evaluating according to θ].

3. The domain $[\Theta \to \Theta]$ is the cpo of all continuous functions from Θ to Θ.

4. Ψ is well defined, in the sense that for all $\Phi \in D$ we have $\Psi\Phi \in D$, or in other words: $\forall \Phi \in D\ \forall R \in \mathbf{Prog}\ \forall \theta_1 \sqsubseteq \theta_2 \sqsubseteq \cdots : \Psi\Phi R(\bigsqcup_i \theta_i) = \bigsqcup_i \Psi\Phi R\theta_i$.

5. Ψ is continuous and therefore $\mu\Psi$ exists (notice that $D = \mathbf{Prog} \to \Theta \to \Theta$ would not work).

We now define $\mathbf{Comp} = \lambda R.\lambda\sigma\ (\mu\Psi)R\{\lambda\sigma.\langle\,\rangle\}\sigma$, and the next thing to prove is that this function is a solution of CE. The proof is by cases, and the only nontrivial case is to prove that

$$(*) \qquad \mathbf{Comp}\langle E|S_1; S_2\rangle\sigma = \langle\sigma\rangle^\wedge\mathbf{Comp}\langle E|S_1\rangle\sigma^\wedge\mathbf{Comp}\langle E|S_2\rangle\sigma'$$

with σ' as usual. Now

$$\mathbf{Comp}\langle E|S_1; S_2\rangle\sigma = (\mu\Psi)\langle E|S_1; S_2\rangle\{\lambda\sigma.\langle\,\rangle\}\sigma$$

$$= \langle\sigma\rangle^\wedge(\mu\Psi)\langle E|S_1\rangle\{(\mu\Psi)\langle E|S_2\rangle\{\lambda\sigma.\langle\,\rangle\}\}\sigma,$$

and the right-hand side of (*) equals

$$\langle\sigma\rangle^\wedge(\mu\Psi)\langle E|S_1\rangle\{\lambda\sigma.\langle\,\rangle\}\sigma^\wedge(\mu\Psi)\langle E|S_2\rangle\{\lambda\sigma.\langle\,\rangle\}\sigma',$$

where $\sigma' = \kappa((\mu\Psi)\langle E|S_1\rangle\{\lambda\sigma.\langle\,\rangle\}\sigma)$.

We thus have to establish a correspondence between the old definition of composition which used κ and $^\wedge$ and the new one which uses continuations. This correspondence is phrased in the next "continuation removal" lemma, which must be clear if the idea behind continuations has been well understood.

LEMMA 5.2. *Let $\Phi \in D = \mathbf{Prog} \to [\Theta \to \Theta]$ be a fixed point of Ψ. For all R, θ and σ we have that $\Phi R\theta\sigma = \tau^\wedge\theta(\kappa\tau)$, where $\tau = \Phi R\{\lambda\sigma.\langle\,\rangle\}\sigma$.*

Proof. Two cases.

1. τ is infinite. Then $\tau^\wedge\theta(\kappa\tau)=\tau$. On the other hand, Φ is continuous in θ and thus monotonic. This means $\tau=\Phi R\{\lambda\sigma.\langle\,\rangle\}\sigma\sqsubseteq\Phi R\theta\sigma$. But τ, being infinite, is maximal, and therefore $\tau=\Phi R\theta\sigma$.

2. The case that τ is finite can be proved by induction on the length of τ.

THEOREM 5.3. *The function* **Comp** *as defined in this section is the unique solution of* CE.

Proof.

1. That **Comp** is a solution of CE follows from the remarks preceding Lemma 5.2.

2. We now prove that Ψ has exactly one fixed point. Let $\Phi\in\mathbf{Prog}\to[\Theta\to\Theta]$ be a fixed point of Ψ such that $\mu\Psi\sqsubseteq\Phi$. We can prove that for all R, θ and σ we have $(\mu\Psi)R\theta\sigma=\Phi R\theta\sigma$.

a. If $(\mu\Psi)R\theta\sigma$ is infinite then it is maximal in Σ^∞. The desired equality then follows from $(\mu\Psi)R\theta\sigma\sqsubseteq\Phi R\theta\sigma$.

b. For all finite $(\mu\Psi)R\theta\sigma$ the desired equality can be proved by induction on its length.

3. For every solution $\mathbf{C}$ of CE we define $\alpha\mathbf{C}:=\lambda R.\lambda\theta.\lambda\sigma.\mathbf{C}R\sigma^\wedge\theta(\kappa(\mathbf{C}R\sigma))$ (compare Lemma 5.2). We can show in a straightforward way that every such $\alpha\mathbf{C}$ is a fixed point of Ψ. Notice that $\alpha\mathbf{C}\in\mathbf{Prog}\to[\Theta\to\Theta]$ must hold, i.e. $\alpha\mathbf{C}$ must be continuous in its continuation parameter.

4. Suppose CE has more than one solution say $\mathbf{C}$ and $\mathbf{C}'$. Then there exist R and σ such that $\mathbf{C}R\sigma\neq\mathbf{C}'R\sigma$. But then $\alpha\mathbf{C}$ and $\alpha\mathbf{C}'$ are both fixed points of Ψ, with $(\alpha\mathbf{C})R\theta\sigma\neq(\alpha\mathbf{C}')R\theta\sigma$, which contradicts 2.

6. Σ^∞ as a metric topological space. At the end of § 3 we investigated the approximation sequence $\perp R\sigma$, $(\Psi\perp)R\sigma$, $(\Psi^2\perp)R\sigma$, $(\Psi^3\perp)R\sigma$, with $R=\langle P\Leftarrow A_1|P;A_2\rangle$ and Ψ the operator derived from CE. We observed that this sequence was not a chain though it converged (in some sense) to the right result. This phenomenon also holds for nonterminating computations like the evaluation of $\langle P\Leftarrow A_1;P|P;A_2\rangle$ in some σ. We can prove that the above observations hold in general, the key lemma is the following.

LEMMA 6.1. *If* Φ_1,Φ_2 *are such that for all* R *and* σ *the sequences* $\Phi_1R\sigma$ *and* $\Phi_2R\sigma$ *agree on their first n places, then for all* R *and* σ *we have that* $(\Psi\Phi_1)R\sigma$ *and* $(\Psi\Phi_2)R\sigma$ *agree on at least their first* $n+1$ *places.*

Proof. Straightforward by cases (if the sequence τ_1 or τ_2 has length smaller than k, we have by definition that τ_1 and τ_2 agree on their first k places iff $\tau_1=\tau_2$).

From this lemma we can deduce that for $n>m$, we have that $(\Psi^n\perp)R\sigma$ and $(\Psi^m\perp)R\sigma$ agree on at least their first m elements. Therefore we can define $\lim\langle(\Psi^k\perp)R\sigma\rangle_k$ as the sequence in Σ^∞ that agrees for every n on its first n elements with $(\Psi^n\perp)R\sigma$. Though we have not defined exactly what "convergence" means, it must be clear that, informally, the sequence $\langle(\Psi^k\perp)R\sigma\rangle_k$ converges to this limit. This convergence is uniform in R and σ in the sense that for all R and σ the first n elements in $(\Psi^n\perp)R\sigma$ are "correct."

If we define $\lim(\Psi^k\perp)$ as $\lambda R.\lambda\sigma.\lim\langle(\Psi^k\perp)R\sigma\rangle_k$ we have that $\Psi(\lim(\Psi^k\perp))=\lim(\Psi^k\perp)$. This holds, because we can prove by induction on n that for all R, σ and n the rows $\Psi(\lim\Psi^k\perp)R\sigma$ and $\lim(\Psi^k\perp)R\sigma$ agree on their first n elements (i.e. the first n elements of $(\Psi^n\perp)R\sigma$). Finally, we can show that Ψ has not more than one fixed point by the following argument. Suppose that there are two fixed points Φ_1 and Φ_2. We prove that for all R, σ the rows $\Phi_1R\sigma$ and $\Phi_2R\sigma$ agree on their first n elements (by induction on n). The case $n=0$ is immediate. If $\Phi_1R\sigma$ and $\Phi_2R\sigma$ agree on $n-1$ elements for all R and σ, then by Lemma 6.1 $(\Psi\Phi_1)R\sigma$ and $(\Psi\Phi_2)R\sigma$ agree

on n elements. But Φ_1 and Φ_2 are fixed points and thus we have that $(\Psi\Phi_i)R\sigma = \Phi_i R\sigma$ for $i = 1, 2$.

We can rephrase all this in the language of topology, by defining that τ_1, τ_2 are close to each other if they have a big common prefix (viz. Definition 6.2). This makes Σ^∞ a metric space and it is possible to formalize the above argument in terms of these topological notions. However, from Lemma 6.1, we can easily derive that the operator Ψ is a contraction (Lemma 6.8), and this means that our fixed point result can be derived in a more elegant way; it is equivalent to a well known theorem from topology. This approach is inspired by an endeavor to apply Nivat's results (see, for example, [11]) to the problem treated in this paper. We saw no way to achieve this, but the basic facts about Σ^∞ that he provided were very useful. In fact, the whole treatment given in this chapter is much in the style of Nivat's.

DEFINITION 6.2.

1. We denote, for $\tau \in \Sigma^\infty$ by $\tau[n]$ the prefix of τ consisting of the first n elements of τ, or τ itself if its length is smaller than n.
2. We define the following distance function d on Σ^∞:

$$d(\tau_1, \tau_2) = \begin{cases} 2^{-n} & \text{if } \tau_1[n-1] = \tau_2[n-1] \text{ and } \tau_1[n] \neq \tau_2[n], \\ 0 & \text{otherwise.} \end{cases}$$

LEMMA 6.3. *d is a metric, i.e. we have the familiar properties*:

$$d(\tau_1, \tau_2) = 0 \text{ iff } \tau_1 = \tau_2,$$
$$d(\tau_1, \tau_2) = d(\tau_2, \tau_1),$$
$$d(\tau_1, \tau_2) \leqq d(\tau_1, \tau_3) + d(\tau_2, \tau_3).$$

Now the metric space (Σ^∞, d) is complete.

LEMMA 6.4. *Every Cauchy sequence $\langle \tau_i \rangle_i$ in Σ^∞ converges.*

Proof [11]. For every k there is an $N(k)$ such that $d(\tau_n, \tau_m) < 2^{-k}$ for all $n, m \geqq N(k)$. Define $\tau_{(k)} = \tau_{N(k)}[k]$. Then $\tau_{(k)}$ agrees on its first k elements with every τ_n for $n \geqq N(k)$. The sequence $\langle \tau_{(k)} \rangle_k$ is a chain, say with lub τ. Now $\langle \tau_i \rangle_i$ converges to τ.

The next thing to do is to make $\mathbf{Prog} \to \Sigma \to \Sigma^\infty$ a metric space by defining

DEFINITION 6.5. $d'(\Phi_1, \Phi_2) = \text{lub}\,\{d(\Phi_1 R\sigma, \Phi_2 R\sigma) \mid R \in \mathbf{Prog}, \sigma \in \Sigma\}$. We have that $(\mathbf{Prog} \to \Sigma \to \Sigma^\infty, d')$ is a complete metric space too.

LEMMA 6.6. *The function d' is a metric, and every Cauchy sequence $\langle \Phi_k \rangle_k$ in $\mathbf{Prog} \to \Sigma \to \Sigma^\infty$ converges with limit $\lambda R.\lambda\sigma.\lim \Phi_k R\sigma$.*

Proof. Standard topology.

LEMMA 6.7. *If for all R, σ we have $d(\Phi_1 R\sigma, \Phi_2 R\sigma) \leqq 2^{-n}$, then for all R, σ: $d((\Psi\Phi_1)R\sigma, (\Psi\Phi_2)R\sigma) \leqq 2^{n-1}$.*

Proof. This is Lemma 6.1.

LEMMA 6.8. *Ψ is a contraction, in particular we have that for all Φ_1, Φ_2*

$$d'(\Psi\Phi_1, \Psi\Phi_2) \leqq \tfrac{1}{2} d'(\Phi_1, \Phi_2).$$

Proof. Follows immediately from Lemma 6.7.

THEOREM 6.9. *Ψ has exactly one fixed point.*

Proof. This is the contraction mapping theorem, viz. [5], [8].

7. Concluding remarks. In a certain sense we have worked in a direction opposite to the one Scott took when he devised his theory of computing. He wanted to exploit notions from topology such as limit and continuity, and therefore he introduced cpo's because the domains on which programs compute are in general not of a topological

kind. We found in § 3 that Σ^∞ considered as a cpo did not have enough structure to prove the desired result. However by using the inherent topology on Σ^∞ we were able to derive this result in an elegant manner (§ 6).

The above results have been derived for a rather simple paradigm language, but the techniques used here can be applied to more sophisticated languages, in particular the language used in Cook's paper [6].

The theory as it stands now cannot be applied to nondeterministic programs, and, as a consequence of this, neither to parallel programs. This is due to the fact that nondeterministic programs generate trees and not rows. However, it seems that the techniques presented here can be extended to trees as well. Part of this extension is reported on in [9].

The central theorem that we have proved four times in this paper holds also if the Cook equations have expressions in their right-hand sides which do not start with a constant one element row. Notice that we have to be careful here. For instance we cannot leave out the $\langle\sigma\rangle$ in the second clause on procedure calls in CE (§ 1) because if we had done so, then $\mathbf{Comp}\langle P \Leftarrow P|P\rangle\sigma$ would not yield an infinite row, which it should do because $\langle P \Rightarrow P|P\rangle$ specifies a nonterminating computation.

Let us investigate the consequences of changing CE such that the fourth clause is altered into

$$\mathbf{Comp}\langle E|S_1; S_2\rangle\sigma = \mathbf{Comp}\langle E|S_1\rangle\sigma^\wedge\mathbf{Comp}\langle E|S_2\rangle(\kappa(\mathbf{Comp}\langle E|S_1\rangle\sigma)).$$

The central theorem of this paper would then be much harder to prove. For instance Definition 2.1 must now be by induction on $\langle n, \text{length}(R)\rangle$ instead of n, and the same holds for induction arguments in some other proofs (for instance Lemma 5.2). Furthermore, the statement $\tau_i \in \Sigma^{*\perp} \Rightarrow \tau_{i+1} \neq \tau_i$ in the proof of Lemma 4.4 is no longer true, as the counterexample $R \equiv \langle E|A_1; A_2\rangle$ and $i = 0$ shows. A weaker version of the lemma holds through:

$$\forall R, \sigma \neq \perp \quad \exists k: \Phi_i R\sigma \in \Sigma^{*\perp} \Rightarrow \Phi_{i+k} R\sigma \neq \Phi_i R\sigma.$$

In § 6 the central Lemma 6.1 does not hold any more, and the sequences $\langle \Psi^k \Phi\rangle_k$ are no longer uniformly convergent (for arbitrary Φ) in R and σ. We have to approach the problem differently. We cannot use the lub distance on $\mathbf{Prog} \to \Sigma \to \Sigma^\infty$ any more, but we have to use the pointwise extension of convergence in Σ^∞, quite analogously to how theory has been set up for cpo's. We now give a brief sketch of how the theorem can be deduced under these new circumstances.

1. Definition. $\langle \Phi_k\rangle_k$ converges iff $\forall R, \sigma: \langle \Phi_k R\sigma\rangle_k$ converges. In that case we define $\lim \Phi_k$ as $\lambda R.\lambda\sigma.(\lim \Phi_k R\sigma)$.

2. Lemma. *Ψ is continuous, in the sense that for all converging sequences $\langle \Phi_k\rangle_k$ we have* $\lim \Psi\Phi_k = \Psi(\lim \Phi_k)$.

3. Lemma. $\forall R, \sigma, n\ \exists N: k \geqq N \Rightarrow \forall \Phi_1, \Phi_2: d((\Psi^k\Phi_1)R\sigma, (\Psi^k\Phi_2)R\sigma) \leqq 2^{-n}$.

This is a useful lemma, in some sense the analogue of Lemma 6.7. Notice that the N in the lemma is in general dependent on R and σ. The proof is by induction on the entity $\langle n, \text{length}(R)\rangle$. The lemma has the following useful consequences (4 and 5).

4. Lemma. *For all Φ we have that $\langle \Psi^k\Phi\rangle_k$ converges.*

5. Lemma. *The limit of $\langle \Psi^k\Phi\rangle_k$ is independent of the initial value Φ.*

6. Theorem. *The (changed) Cook equations have exactly one solution.*

Proof. There is a fixed point (for instance $\lim(\Psi^k\perp) =: \mu\Psi$), by results 2 and 4. If there was another fixed point Φ_0, then we would have that $\mu\Psi = \lim \Psi^k\Phi_0 = \lim\langle \Phi_0, \Phi_0, \cdots\rangle = \Phi_0$ (the first equality holds by result 5).

The above remarks show that it pays off (technically) to demand that the right-hand sides in CE all begin with a constant row. There are other reasons for this. The operational semantics yields a row of states which is intended as the trace left by execution of the program under consideration. Now execution of (for instance) $A_1; A_2$ can be divided into three parts: namely first determining that the statement is a composition of two other statements, secondly evaluating the first statement, and lastly evaluating the second one. It is reasonable that each stage of this evaluation has its effect on the trace. More generally, every clause in the Cook equations should add an element to the trace because it corresponds either to some elementary action, or to a decomposition of the statement being evaluated.

Related work and acknowledgments. In a letter to Cook [1], Krzysztof Apt suggested a method to compute **Comp** which is related to the technique of § 2: he proposes to define by induction on k the row **Comp**$'R\sigma k$ which should consist of the first k elements of **Comp**$R\sigma$. Having defined **Comp**$'$ he then defines **Comp**$R\sigma = \tau$ iff $\exists k$: **Comp**$'R\sigma n = \tau$ for all $n \geqq k$. He therefore defines **Comp** only for finite rows. The same holds for the results of Jeff Zucker in the appendix of [3]. He defines **Comp** as a fixed point of a set of equations derived from CE. He does this by using the recursion theorem. The technique in § 4 of adding the bottom element $\perp$ to mark a row as not yet completed has been used by Ralph Back in his analysis of unbounded nondeterminism [2]. The results in § 6 were inspired by the reading of Nivat's and others work on infinite computations, as reported on for instance in [11]. The topology on Σ^∞ was presented there, and also the proof of Lemma 6.4 can be found there.

A more elaborate version of this paper (more remarks and better worked out proofs) is registered as Mathematical Centre Report [4].

I acknowledge with pleasure the assistance of the following persons: the members of the Dutch working group on semantics, in particular Ruurd Kuiper with whom I had frequent and stimulating discussions on the material presented here. I would like to thank Jaco de Bakker and the referees for useful comments on the manuscript, and finally I would also like to thank Nizethe Kemmink and Susan Carolan for doing such a good typing job on a rather disjointed manuscript.

REFERENCES

[1] K. R. APT, personal communication.

[2] R. J. BACK, *Semantics of unbounded nondeterminism*, in Proc. 7th Colloquim on Automata, Languages and Programming, J. W. de Bakker and J. van Leeuwen, eds., Lecture Notes in Computer Science 85, Springer, New York, 1980, pp. 51–63.

[3] J. W. DE BAKKER, *Mathematical Theory of Program Correctness*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

[4] A. DE BRUIN, *On the existence of Cook semantics*, Report IW 163/81, Mathematical Centre, Amsterdam, 1981.

[5] G. CHOQUET, *Cours d'analyse, Tôme* II, *Topologie*, Masson and Cie, Paris, 1964.

[6] S. A. COOK, *Soundness and completeness of an axiom system for program verification*, this Journal, 7 (1978), pp. 70–90.

[7] C. A. R. HOARE AND P. E. LAUER, *Consistent and complementary formal theories of the semantics of programming languages*, Acta Informatica, 3 (1974), pp. 135–153.

[8] A. N. KOLMOGOROV AND S. V. FOMIN, *Elements of the Theory of Functions and Functional Analysis*, Graylock Press, Rochester, NY, 1957.

[9] R. KUIPER, *An operational semantics for nondeterminism equivalent to a denotational one*, in Proc. International Symposium on Algorithmic Languages, J. W. de Bakker and J. C. van Vliet, eds., North-Holland, Amsterdam, 1981, pp. 373–398.

[10] P. E. LAUER, *Consistent formal theories of the semantics of programming languages*, IBM Laboratory Vienna, Techn. Report TR 25-121, 1971.

[11] M. NIVAT, *Infinite words, infinite trees, infinite computations*, in Foundations of Computer Science III, Part 2: Languages, Logic, Semantics, J. W. de Bakker and J. van Leeuwen, eds., Mathematical Centre Tracts 109, Mathematical Centre, Amsterdam, pp. 1–52.

[12] D. SCOTT, *Outline of a mathematical theory of computation*, Technical monograph PRG-2, Oxford University Computing Laboratory, Programming Research Group, 1970.

[13] ———, *The lattice of flow diagrams*, Technical monograph PRG-3, Oxford University Computing Laboratory, Programming Research Group, 1971.

[14] ———, *Continuous lattices*, Technical monograph PRG-7, Oxford University Computing Laboratory, Programming Research Group, 1971.

[15] J. E. STOY, *Denotational Semantics—The Scott–Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, 1977.

SIAM J. COMPUT.
Vol. 13, No. 1, February 1984

© 1984 Society for Industrial and Applied Mathematics
0097-5397/84/1301-0002 $01.25/0

GENERALIZED SELECTION AND RANKING: SORTED MATRICES*

GREG N. FREDERICKSON† AND DONALD B. JOHNSON‡

Abstract. A more general version of the well-known selection problem is formulated, in which constraints on the input set are allowed. Selection (and also ranking) problems are solved optimally for the broad class of inputs constrained to be collections of matrices with sorted rows and sorted columns. The characterization of problem complexity includes an asymptotically significant dependency on the rank of the solution element.

Key words. Cartesian sums, complexity, data structures, lower bounds, matrices, ranking, selection, sets, sorted matrices, succinct descriptions

1. Introduction. Consider a set of elements drawn from a universe with a total order. The problem of selecting an element of given rank has received considerable attention [FG], [H], [Ki], [PY], [SPP], [Yp] since its complexity was first demonstrated to be proportional to the cardinality of the set [B]. A more general version of this problem, and one with several practical applications, allows constraints on the input set. The constraints can be specified in either of two ways. Certain total orders on the set can be forbidden by presenting the inputs in a succinct form as, for instance, the sums of the pairs in a Cartesian product of two input sets, denoted $X+Y$. Alternatively, a certain partial order can be specified which the input is known to obey because of comparisons previously made or relationships between elements that are implied by construction. Implicitly specified constraints may allow a succinct presentation of the input in this case also. We have treated the case of Cartesian sums in a previous paper [FJ1].

This paper deals with an important instance of the second case in which the input obeys any specified partial order which can be described as a collection of what we call sorted matrices. An $n \times m$ matrix X is a *sorted matrix* if each row and each column is in nondecreasing order. Individual elements x will be identified by row and column indices i and l so that $X = \{x_{il}\}$. (We use the term "set" and set notation for both sets and multisets.) A collection of sorted matrices is a set X partitioned into blocks $\{X_j\}_{j=1}^{N}$, where each block X_j is a sorted matrix of dimensions $n_j \times m_j$. We may characterize the simple (unconstrained) selection or ranking problem on N elements as applying to a collection of N trivial matrices, each one a singleton. Additionally, some problems on partially ordered blocks of elements may be translated into problems on matrices by viewing them as padded with dummy entries.

We consider both selection and the complementary problem of ranking in such sets. *Selection* in a set X determines, for a given rank k, an element that is kth in some total ordering of X. *Ranking* determines, for a given element, its rank in X.

Our algorithms for these problems are asymptotically optimal, and, in all but sets with trivial structure, are sublinear in the size of the problem, $\sum_{j=1}^{N} n_j m_j$. As we have

* Received by the editors July 11, 1981, and in revised form November 18, 1982.

† Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802. Current address: Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907. The work of this author was partially supported by the National Science Foundation under grant MCS 7909259.

‡ Computer Science Department, Pennsylvania State University, University Park, Pennsylvania 16802. The work of this author was partially supported by the National Science Foundation under grants MCS 77-21092 and MCS 80-002684.

indicated, the inputs may be succinctly presented, for example, as a collection of sorted sets defining sorted Cartesian matrices or as a collection of functions and a finite domain of n arguments. When sets are not succinctly presented, sublinear algorithms have application when the cost of inputting can be distributed over several computations. Our complexity measure charges unit cost for operations on a random access machine. We assume that any element x_i in the set X on which the problem is posed can be accessed (or constructed) at unit cost, given i.

The complexity of our problems depends not only on the matrix dimensions, but also on the rank k. For instance, selection in an $n \times m$ sorted matrix may require anywhere from constant time up to time that is $\Theta(m \log(2n/m))$, depending on the rank k. This dependence on k in our constrained problems is in marked contrast to the situation of the unconstrained problem, in which the value of k affects only the constant factor.

As stated earlier, there are several problems of practical interest in which a collection of sorted matrices arises. A p-center of a nonnegatively-weighted network is a set of p "supply" vertices chosen so as to minimize the maximum distance from any vertex to a supply point. An efficient method to locate p-centers in networks with tree topologies uses repeated selection of intervertex path lengths [CT1], [CT2], [KH], [MTZC], [FJ2]. While it is true that the tree itself is a succinct representation of all path lengths, the tree itself does not facilitate quick selection of a kth path. As we show in [FJ2], the set of all path lengths may be represented as collection of Cartesian matrices $X_i + Y_i$ where X_i and Y_i are sorted. The selection algorithms in this paper may be used to select optimally in such a collection.

Selection in $X + Y$ (unsorted) may be used to compute the Hodges–Lehmann estimator in statistics [HL], [MR], [JR], [S]. It is desired to estimate the difference in the means of the two populations from which sets of observations X and Y, respectively, have been drawn. The Hodges–Lehmann estimator for this difference is the median of $X + (-Y)$, the set $\{x - y | x \in X \text{ and } y \in Y\}$. If there are many sets of observations, and estimates are to be obtained between all pairs of them, then it will be worthwhile to sort each set of observations and to use an algorithm in this paper on the sorted Cartesian matrix for each pair.

Selection in partitions with sorted blocks solves the problem of optimum discrete distribution of effort where m concave functions $\{f_j\}$ are given [Ko], [GM], [FJ1]. It is desired to distribute discretely n units of "effort" among the $\{f_j\}$ so as to maximize the sum of the function values, i.e., maximize $\sum_{j=1}^{m} f_j(a_j)$ subject to $\sum_{j=1}^{m} a_j = n$ for nonnegative integers $\{a_j\}$. The problem is solved by selecting the nth largest element in $X = \{f_j(i+1) - f_j(i) | j = 1, \cdots, m, i = 0, \cdots, n-1\}$, where the concavity of the functions yields one sorted $n \times 1$ matrix in X for each j.

To present the inherent complexity of our problems in a reasonably simple form we factor out of our discussion certain simple problem reductions. It is reasonable to expect that any selection problem on N sorted matrices of dimension $n_j \times m_j$ will be presented in what we call *reduced* form where $m_j \leqq n_j \leqq k$, for all j, and $k \leqq \lceil \frac{1}{2} \sum_{j=1}^{N} m_j n_j \rceil$. Whenever this is not the case, a problem can be transformed in $O(N)$ time to a reduced problem with dimensions $n'_j \times m'_j$ where selection of any element with rank k' solves the given problem and $m'_j \leqq m_j$, $n'_j \leqq n_j$, and $m'_j \leqq n'_j \leqq k' \leqq k$. It will be evident from the bounds we give on the running times for our algorithms that transformation to a reduced problem never worsens asymptotic complexity and may indeed improve it. Thus, while our analyses do not depend on inputs being in reduced form, it may be assumed in using our bounds either that inputs are in reduced form or that the parameter values in the bounds should, for purposes of describing complexity, be

replaced by the parameters following reduction. Bounds in which the effect of reduction is expressed directly in terms of the given parameters are unwieldy.

The reduction process depends entirely on the parameters $\{(n_j, m_j)\}$ and k, and not on problem values. The first step rests on the observation that any problem selecting the kth smallest element may be posed as a $\bar{k}$th largest selection for $\bar{k} = \sum_{j=1}^{N} m_j n_j - k + 1$. Since selecting for kth smallest and kth largest are symmetric, replacing k with $\min\{k, \bar{k}\}$ can be done in $O(N)$ time. The entire reduction procedure is as follows.

(1) Transform so that $k = \min\{k, \bar{k}\}$.

(2) Let $m_j = \min\{m_j, k\}$, $n_j = \min\{n_j, k\}$ for each j.

(3) If, for some i, $m_i = 1$ and $n_i > \sum_{j \neq i} m_j n_j$ then replace X_i with the portion of X_i from index $k + 1 - \sum_{j \neq i} m_j n_j$ to index k, and let $k = \sum_{j \neq i} m_j n_j$.

Application of this procedure yields in $O(N)$ time a reduced problem the solution to which is a solution to the original problem.

The greater portion of our paper deals with selection. In § 2, we give a basic algorithm for selection in a sorted matrix or, more generally, in a set of sorted matrices all of the same size. In § 3, we show how to aggregate sorted matrices of different sizes into a form that may be handled by the basic algorithm. The algorithms in these sections are asymptotically optimal for selecting medians.

Algorithms for selecting optimally for any k are presented in §§ 4, 5 and 6. In § 4, we show how to select optimally in a single sorted matrix no matter what rank is desired. In § 5, a selection algorithm is given for a set of thin sorted matrices which is again optimal for all values of k. The synthesis of the ideas in these sections results, in § 6, in an algorithm for the general problem which is optimal for all collections of sorted matrices and over the whole range of k for any such collection.

Several algorithms for ranking are discussed in § 7. It appears to be conceptually easier to rank than to select in the sets we have studied although, interestingly, both have the same asymptotic complexity. As has been noted, all of our algorithms for selection and rankings are optimal to within a constant multiplicative factor over all values of the relevant parameters. Lower bounds that meet our upper bounds are established in § 8.

This paper extends preliminary results which appear in [FJ3].

2. Basic algorithm for selection in a sorted matrix. In this section we give our basic algorithm for selection and apply it to a single sorted matrix of dimensions $n \times m$ where $1 < m \leqq n$. (When $m = 1$ a solution can be found in constant time.) The running time of the basic algorithm is independent of k and will later be shown to be optimal on reduced problems where $k = \Theta(nm)$. (As described in the introduction any problem where n exceeds k can immediately be reduced in dimensions so that $m \leqq n \leqq k$.) The basic algorithm may also be applied to a set of equal-sized matrices or to an input with the essential properties of such a set. Later sections will give these results and the preprocessing of inputs into the desired form, solving optimally all selection problems posable on sorted matrices.

Our basic algorithm performs a sequence of selections on elements representing submatrices of the given problem. (See Fig. 1.) The selections are performed in pairs, yielding upper and lower bounds on the kth element and identifying submatrices that need no longer be considered. At each iteration the remaining submatrices are subdivided to allow refined bounds to be gotten by the next pair of selections. These iterations finally reduce the submatrices to single elements from among which a solution is found directly by selection.

ALGORITHM SELECT(CELLS, k)
(1) $k' \leftarrow k$
(2) **for** $p \leftarrow 1$ **to** $\log_4 S$ **do**
 (2.1) Split each cell in CELLS.
 (2.2) Let $q \leftarrow \lceil k'/(S/4^p) \rceil + B_p$. If $q \leqq |\text{CELLS}|$, select a qth element x_u in the multiset $\{\min(C) | C \in \text{CELLS}\}$. Discard $|\text{CELLS}| - q + 1$ cells from CELLS, retaining every cell C with $\min(C) < x_u$ and no cell C with $\min(C) > x_u$.
 (2.3) Let $r \leftarrow \lfloor k'/(S/4^p) \rfloor - B_p$. If $r \geqq 1$, select an rth element x_l in the multiset $\{\max(C) | C \in \text{CELLS}\}$. Discard r cells from CELLS, retaining every cell C with $\max(C) > x_l$ and no cell C with $\max(C) < x_l$. Set $k' \leftarrow k' - r(S/4^p)$.
(3) Select the k'th element in CELLS.

FIG. 1. *The basic selection algorithm.*

A submatrix of a given matrix is termed a *cell.* Associated with each cell C is a smallest element $\min(C)$, chosen as the element with smallest row and column indices, and a largest element $\max(C)$, chosen as the element with largest row and column indices. Initially there is a single cell, the matrix X, of dimensions $n \times m$ and number of elements $S = nm$. An iteration begins by splitting all remaining cells into four parts. If both dimensions of a cell are greater than one, then the cell is called *thick*, and each dimension is split in half. If one dimension is equal to one, then the cell is called *thin*, and the other dimension is split into quarters. For ease of exposition it is assumed that S is a power of 4. Hence every cell will be of size a power of 4 and each dimension greater than 1 will be divisible by 2. It is sufficiently easy to realize this restriction by means of implicit padding of matrices so that the restriction is of little practical consequence.

The structure of the matrix induces a partition of the set of remaining cells into subsets called *chains*. If the cells are thick, then two cells belong to the same chain if and only if they are in the same diagonal of the matrix of submatrices obtainable from the original matrix by partitioning it into submatrices of the same dimensions as the cells. If the cells are thin, then two cells are in the same chain if and only if they come from the same column in the original matrix. In either case, it is easily seen that if two cells C' and C'' are in the same chain, then either $\max(C') \leqq \min(C'')$ or $\max(C'') \leqq \min(C')$. Let B_p be the maximum possible number of chains after splitting cells on the pth iteration. If cells are thick, then the maximum possible number of chains is $2^{p+1} - 1$; otherwise it is m. Since $2^{p+1} - 1 < m$ if and only if cells are thick, $B_p = \min\{m, 2^{p+1} - 1\}$.

After the remaining cells have been split, two selection computations are performed. In the first, the smallest elements of the remaining cells are selected among to find a qth element x_u, where $q = \lceil k'4^p/S \rceil + B_p$ and k' is the rank to be selected for in the set of elements in the remaining cells. The value x_u is an upper bound for more than k' remaining elements. Thus all but $q - 1$ cells may be discarded, such that each cell C with $\min(C) > x_u$ is discarded, along with some of the cells C with $\min(C) = x_u$. Similarly, the largest elements of remaining cells are selected among to find an rth element x_l, where $r = \lfloor k'4^p/S \rfloor - B_p$. The value x_l is a lower bound for all but fewer than k' elements, and thus r cells may be discarded following a discarding rule similar to the above.

After $\log_4 S$ iterations, all remaining cells will be single elements, and there will be at most $O(m)$ of them. A linear-time selection algorithm is employed to select the appropriate element.

LEMMA 1. *The basic algorithm* SELECT *correctly computes a kth element in a collection of cells of equal size, a power of* 4.

Proof. We first establish that, after every iteration, a k'th element in the multiset union of the set of cells is a kth element in matrix X. This is shown by induction on the number of iterations. The above claim is certainly true after zero iterations. Consider the pth iteration, $p>0$. Each cell is of size $S/4^p$. Hence there are fewer than $\lceil k'/(S/4^p)\rceil$ cells with all values less than the k'th element. Since there are no more than B_p chains, there are no more than B_p cells with some values less than the k'th element and some values greater than or equal to the k'th element in the same cell. Thus x_u is the minimum of a cell in which there are no values smaller than the k'th element. Therefore discarding according to the rule presented will retain k' elements no larger than the k'th element and discard no element that is smaller.

By a similar argument it can be shown that the second selection retains an element that is k'th before the second selection and discards no element greater than this element. Therefore k' is adjusted correctly. So, by induction, at every iteration there is a k'th element which is kth in the original matrix. Hence the final selection in the set of singleton submatrices yields an element that is kth in the original matrix as required. □

THEOREM 1. *A selection problem on a sorted matrix of dimensions* $n\times m$, $1<m\leqq n$, *is solved by the basic algorithm in* $O(m\log(2n/m))$ *time.*

Proof. Correctness follows from Lemma 1. At most $q-1-r\leqq 2B_p$ cells remain at the end of the pth iteration of step 2. Since remaining cells are split into quarters, at most $4(2B_p)$ elements are selected among in the $(p+1)$th iteration. If a linear-time selection algorithm [B] is used, then the time per iteration is proportional to the number of cells. Thus the complexity of step 2 is at most proportional to

$$\sum_{p=1}^{\log_4 S} B_p \leqq \sum_{p=1}^{\log_4 S} \min\{2^{p+1}, m\}$$

$$= \sum_{p=1}^{\log_2 m-1} 2^{p+1} + \sum_{p=\log_4 m^2}^{\log_4 nm} m < 2m + 2m\log_4(n/m) = O(m\log(2n/m)).$$

At the completion of step 2, there will be no more than $2m$ single elements, so that step 3 will run in time $O(m)$. □

Dobkin and Munro have shown a similar result for the case $n=m$ [DM].

Since the dimensions m and n which parameterize the bound in Theorem 1 can always in constant time be made to satisfy $m\leqq n\leqq k$, they may be taken to be $\min\{m,k\}$ and $\min\{n,k\}$, respectively. The bound in Theorem 1 can be improved significantly when $k=o(nm)$. We deal with this case in § 4.

3. Selection in a collection of sorted matrices. We consider selection in a collection of $N>1$ sorted matrices $\{X_j\}$, $j=1,\cdots,N$, of dimensions $n_j\times m_j$, where $n_j\geqq m_j$ for each j. The basic algorithm in § 2 can solve this selection problem if the input matrices are first combined into a single matrix-like aggregate with all of the essential properties of the cells on which the basic algorithm operates. The aggregate is generated by repeatedly combining four smaller structures of the same size into a larger structure, until a single structure results. The aggregate and smaller structures will be split and selected among in a fashion analogous to that presented in § 2. As before, it is assumed in the presentation of the algorithm that n_jm_j is a power of 4, for all j.

In order to combine all matrices, it may be necessary to create additional matrices, with all elements of value ∞, called *dummy matrices*. Structures created by combining

are called synthetic cells. A *synthetic cell* C of size S is a collection of four structures $\{C_1, C_2, C_3, C_4\}$ of size $S/4$, where a structure is an original matrix of any shape, a dummy matrix, or a synthetic cell. It is not hard to combine matrices so as to minimize the total number of dummy matrices. If this is done, there will be no more than three dummy matrices of each size less than the size of the aggregate. It follows that the number of original and dummy matrices is one more than three times the total number of synthetic cells. The number of synthetic cells is thus less than the number of cell sizes used plus one third the number of original matrices.

Synthetic cells are handled in a fashion similar to cells representing submatrices. When split, a synthetic cell is replaced by its four components. Any dummy matrices so generated are discarded immediately. Associated with a synthetic cell C are two values, min (C) and max (C), upon which selections are performed. The value min (C) is $\min\{\min(C')|C' \in C\}$, and max (C) is similarly defined. Each synthetic cell is defined to be in a chain by itself.

With these conventions established it may be seen from the proof of Lemma 1 that the basic algorithm selects correctly in any synthetic cell. Let S be the number of elements in the largest synthetic cell, and let B_p be, as before, the maximum possible number of chains after splitting cells on the pth iteration. The value of B_p may be described as follows. For $j = 1, \cdots, N$, let B_p^j be the maximum possible number of chains in X_j on the pth iteration, if the current cell size is not larger than the size of X_j, and zero otherwise. Let B_p^0 be the maximum number of synthetic cells active on the pth iteration. Then

$$B_p = B_p^0 + \sum_{j=1}^{N} B_p^j.$$

Let S_j be the size of X_j. Then it can be seen that $B_p^j = \min\{m_j, \lceil 2^{p+1}\sqrt{S_j/S}\rceil - 1\}$.

THEOREM 2. *A selection problem in a collection of sorted matrices* $\{X_1, \cdots, X_N\}$ *in which* X_j *has dimension* $n_j \times m_j$, $n_j \geqq m_j$, *can be solved in* $O(\sum_{j=1}^N m_j \log(2n_j/m_j))$ *time.*

Proof. The method, aggregation into a single synthetic cell followed by application of the basic algorithm, has been described in the preceding paragraphs. Correctness of this method follows from Lemma 1. If the objects to be aggregated are bucket sorted according to size, aggregation can be performed in time proportional to $N + \log S$. By construction, $S < 4\sum_{j=1}^N S_j$. Thus

$$\log_4 S < \sum_{j=1}^{N} \log_4(4n_j m_j) \leqq \sum_{j=1}^{N} m_j \log_4(4n_j/m_j).$$

It follows that aggregation is $O(\sum_{j=1}^N m_j \log(2n_j/m_j))$.

In a fashion similar to that in Theorem 1, the complexity of step 2 of the basic algorithm is found to be at most proportional to

$$\sum_{p=1}^{\log_4 S} B_p = \sum_{p=1}^{\log_4 S} \sum_{j=0}^{N} B_p^j = \sum_{j=1}^{N} \sum_{p=1}^{\log_4 S} B_p^j + \sum_{p=1}^{\log_4 S} B_p^0.$$

The second sum is the total number of synthetic cells which, by previous remarks, is less than $\log_4 S + N/3$. For any $j = 1, \cdots, N$, we have

$$\sum_{p=1}^{\log_4 S} B_p^j = \sum_{p=1}^{\log_4 S} \min\{\lceil 2^{p+1}\sqrt{S_j/S}\rceil - 1, m_j\}$$

$$\leqq \sum_{p=1}^{\log_4 S_j} \min\{2^{p+1}, m_j\} = O(m_j \log(2n_j/m_j)).$$

Using the bound just obtained for $\log_4 S$, it follows that the complexity of step 2 is

$$O\left(\sum_{j=1}^{N} m_j \log_2 (2n_j/m_j)\right).$$

At the completion of step 2, there are $O(\sum_{j=1}^{N} m_j)$ single elements remaining, in which to select. □

We note that it is not necessary to combine all matrices into a single synthetic cell. The combining process need only be continued to the first point at which all structures to be combined are the same size. The basic algorithm may then be applied to this set, with the value of S set to this size. It may be verified that the time complexity of this variant is no greater, and that correctness follows from the correctness of the algorithm presented. We also note that the bound of Theorem 2 does not depend on reduced inputs though, as later results show, the bound is optimal for selecting medians in reduced inputs, i.e. when $m_j \leqq n_j \leqq k$ for all j and $k = \Theta(\sum_{j=1}^{N} m_j n_j)$. The bound of Theorem 2 can be improved when $k = o(\sum_{j=1}^{N} m_j n_j)$. This case is considered in its complete generality in § 6, after preliminary results are established in §§ 4 and 5.

4. Selection in a sorted matrix when k is small. The basic algorithm is correct for any k, but when applied to a single matrix the running time is suboptimal if $k = o(nm)$. To realize an optimal running time as a function of k, m, and n, it suffices to extract from the given matrix a certain set of submatrices guaranteed to contain all elements less than the kth and at least k elements no larger. Although these submatrices are of different shapes, they are of equal size and thus the basic algorithm may be applied directly without the combining presented in § 3. We continue our assumption that nm is a power of 4.

The submatrices are identified as follows. Let K be the smallest power of 4 no smaller than k, and let $H = \min\{\sqrt{K}, m\}$. Let X_0 be a submatrix with dimension $K/H \times H$ in the upper left corner of X. Extending downward (in the direction of increasing row number) is a series of submatrices $X_1, \cdots, X_{\log_2 H}$ of dimensions $n_j \times m_j = (2^{j-1}K/H) \times (H/2^{j-1})$ for $j = 1, \cdots, \log_2 H$. Thus X_j is the submatrix of X with index range $(2^{j-1}K/H, 2^jK/H] \times [1, H/2^{j-1}]$. If $n \leq K$, then X is assumed to be padded implicitly with elements of value ∞. If $H = \sqrt{K}$, then there is an analogous sequence $X'_1, \cdots, X'_{\log_2 H}$ to the right of X_0, of dimensions $(H/2^{j-1}) \times (2^{j-1}K/H)$ for $j = 1, \cdots, \log_2 H$. As before, implicit padding is used as necessary.

It is sufficient to confine the search for a kth element to the union of these submatrices. Since the rows and columns of X are sorted, it follows that every element discarded is no smaller than k elements that are retained. Thus a kth element in the elements retained is kth in the original matrix.

THEOREM 3. *Selection of a kth element in a sorted matrix of dimensions* $n \times m$, $n \geqq m > 1$, *can be performed in* $O(h \log(2k/h^2))$ *time, where* $h = \min\{\sqrt{k}, m\}$.

Proof. The preprocessing just described generates a problem that can be solved by the basic algorithm. Correctness follows from the above arguments of correctness for the preprocessing plus those in Lemma 1. From the proof of Theorem 2, it can be seen that selection in the collection of submatrices uses $O(\sum_{j=1}^{\log_2 H} m_j \log(2n_j/m_j))$ time. We have

$$\begin{aligned}\sum_{j=1}^{\log H} m_j \log(2n_j/m_j) &= \sum_{j=1}^{\log H} (H/2^{j-1}) \log(2^{2j-1}K/H^2) \\ &= H \log(2K/H^2) \sum_{j=1}^{\log H} 1/2^{j-1} + 2H \sum_{j=1}^{\log H} (j-1)/2^{j-1} \\ &< 2H \log(2K/H^2) + 4H.\end{aligned}$$

Since h is $\Theta(H)$ and k is $\Theta(K)$, the above expression is $O(h \log(2k/h^2))$. □

5. Selection in a collection of thin matrices when k is small. While we have shown how to take advantage of a small value of k to yield a faster algorithm for selection in a single sorted matrix, the running time of our algorithm for selection in a collection of sorted matrices is insensitive to k. In this section we give an algorithm for selection in a collection of sorted matrices that will later be shown optimal for all k. The collection of matrices on which it operates, however, is restricted to the form $\{X_j\}, j=1, \cdots, N$, where each matrix X_j is of dimension $n_j \times 1$. We call such matrices *thin matrices*. The thin matrix algorithm will be an essential component of our most general algorithm, given in the next section. We again assume that $N>1$.

An algorithm of equivalent complexity on matrices all of which are of equal size can be found in our earlier paper [FJ1] where techniques were used that could also be applied to unsorted Cartesian matrices. The approach presented here is simpler as well as more general in the context of sorted matrices. The matrices are preprocessed, yielding a problem which is solved by the basic algorithm. The preprocessing is in two steps. First the matrices are truncated to satisfy requirements on matrix sizes that reflect the value of k. Then the matrices are combined as in § 3.

Truncation is done according to the following generalization of the truncation given in [FJ1]. Let $i^* = \lceil (k+1)/(\lfloor N/2 \rfloor + 1) \rceil$. The matrices $X_1, \cdots, X_N$ are reordered so that for $a \leqq \lfloor N/2 \rfloor \leqq b$, the i^*th value in X_a is not greater than the i^*th value in X_b. If $i^* > n_j$, assume the i^*th value of X_j to be ∞. Truncate matrices $X_j, j = \lfloor N/2 \rfloor + 1, \cdots, N$, by resetting $n_j = \min\{n_j, i^*-1\}$. Repeat the process recursively with the first $\lfloor N/2 \rfloor$ matrices, until $i^* > n_j$ for all X_j.

When the discarding is completed, at least k elements remain and any element that is kth among them is a kth element in the original problem. This follows since whenever a (noninfinite) element is discarded there must be at least $(\lfloor N/2 \rfloor + 1)i^* - 1 \geqq k$ elements retained that are no larger than it.

A linear-time selection algorithm may be used among the i^*th elements to determine the reordering of the matrices. The time for the first reordering is $O(N)$, and thus the total time for the truncation is proportional to at most $N + \lfloor N/2 \rfloor + \cdots + 1$, which is $O(N)$. The resulting set of truncated matrices has an order that satisfies

$$n_j \leqq \lceil (k+1)/(N/2) \rceil 2^{\lfloor \log (N/j) \rfloor} - 1 < 4k/j - 1$$

for all j.

The truncated matrices are aggregated as in § 3 and the basic algorithm is then applied. To analyze the time required for completing the problem, the following lemma is useful. We call $\mathcal{K} = \{k_j\}$ an *N-partition* (or simply a *partition*) of k when $\mathcal{K}$ consists of N nonnegative integers that sum to k. We allow $k_j = 0$ to deal with the case $k < N$ which arises in a later section.

LEMMA 2. *For all $\mathcal{N} = \{n_j\}, j = 1, \cdots, N$, for which $0 < n_j \leqq \lfloor 4k/j \rfloor$ and $N \leqq k \leqq \lceil \frac{1}{2} \sum_{j=1}^N n_j \rceil$ there exists a partition $\mathcal{K} = \{1 \leqq k_j \leqq n_j\}$, $j = 1, \cdots, N$, of k for which $\sum_{j=1}^N \log n_j \leqq \sum_{j=1}^N \log k_j + 5N$.*

Proof. The proof is by induction on $M = \sum_{j=1}^N n_j$. The basis is where $M = \sum_{j=1}^N \lfloor 4k/j \rfloor$. A partition of k may be generated for which k_j is either $\lceil k/N \rceil$ or $\lfloor k/N \rfloor$ for each j. Then

$$\begin{aligned}\sum_{j=1}^N \log n_j \leqq \sum_{j=1}^N \log 4k/j &= N \log (k/N) + \sum_{j=1}^N \log (4N/j) \\ &< N \log (k/N) + N(2 + \log e) \\ &< N \log (k/N) + 4N < \sum_{j=1}^N \log k_j + 5N.\end{aligned}$$

For the induction step, $M<\sum_{j=1}^{N} \lfloor 4k/j \rfloor$. We assume that the lemma is true for all $\mathcal{N}'$ with $\sum_{j=1}^{N} n_j' = M+1$. Let $\mathcal{N}=\{n_j\}$ be a set with $\sum_{j=1}^{N} n_j = M$. Since $M< \sum_{j=1}^{N} \lfloor 4k/j \rfloor$ there is a set $\mathcal{N}'$ in which $n_j' = n_j$ for all j except $j=l$, for which $n_l' = n_l + 1$. Thus

$$\sum_{j=1}^{N} \log n_j = \sum_{j=1}^{N} \log n_j' + \log (n_l/n_l')$$

$$\leqq \sum_{j=1}^{N} \log k_j' + 5N + \log (n_l/n_l')$$

for some partition $\mathcal{K}'$ of k, by the induction hypothesis. If $k_l' < n_l'$, then $\mathcal{K}=\mathcal{K}'$ is a partition of k, satisfying the lemma. Otherwise, let $\mathcal{K}$ differ from $\mathcal{K}'$ only in that $k_l = k_l' - 1$ and $k_r = k_r' + 1 \leqq n_r$ for some $r \neq l$. Then

$$\sum_{j=1}^{N} \log n_j \leqq \sum_{j=1}^{N} \log k_j' + 5N + \log (k_l/k_l')$$

$$< \sum_{j=1}^{N} \log k_j + 5N. \qquad \square$$

Let $\tilde{\mathcal{K}} = \{\tilde{k}_j\}$ be a partition of k where $\tilde{k}_j \leqq n_j$ for all j. $\tilde{\mathcal{K}}$ is called a *maximizing partition of* k if $\sum_{\tilde{k}_j>0} \log \tilde{k}_j$ is maximized. As the following theorem states, the running time of the thin matrix algorithm is bounded by a quantity proportional to $\sum_{j=1}^{N} \log (\tilde{k}_j + 1)$. The theorem states this bound in a form compatible with later results.

THEOREM 4. *Selection of a kth element in a set of* $N>1$ *sorted matrices* $\{X_j\}$, $j = 1, \cdots, N$, *of dimensions* $n_j \times 1$ *can be solved in time* $O(N + \sum_{\tilde{k}_j>0} \log \tilde{k}_j)$, *where* $\tilde{\mathcal{K}} = \{\tilde{\mathcal{K}}_j\}$ *is a maximizing partition of k.*

Proof. The recursively applied preprocessing costs $O(N)$ time. Upon completion, no more than $N' = \min\{k, N\}$ nonempty matrices remain. Let the resulting sizes be $n_j' \leqq n_j$ for all j under an appropriate reindexing. By Theorem 2, the algorithm of § 3 will require $O(N' + \sum_{j=1}^{N'} \log n_j')$ time to complete the solution. From Lemma 2, this quantity is $O(N + \sum_{j=1}^{N'} \log k_j')$ for some partition $\mathcal{K}'$ of k for which $1 \leqq k_j' \leqq n_j'$ for all j. Since $n_j' \leqq n_j$ for a suitable reindexing of the given $\{n_j\}$, $\sum_{j=1}^{N} \log k_j' \leqq \sum_{\tilde{k}_j>0} \log \tilde{k}_j$ for $\tilde{\mathcal{K}}$, a maximizing partition of k for which $\tilde{k}_j \leqq n_j$. $\square$

Later results show the bound in Theorem 4 to be optimal on reduced inputs. Therefore, the reduction as described in the introduction (since it takes $O(N)$ time) should be a first step in solving any problem on thin matrices. it is useful also to note that this bound is a restriction to thin matrices of the bound we obtain in the next section for the general problem,

$$O(N + \textstyle\sum_{h_j>0} m_j \log (2\tilde{k}_j/h_j^2)),$$

where $h_j = \min\{\sqrt{\tilde{k}_j}, m_j\}$, $j = 1, \cdots, N$.

6. Selection in a collection of sorted matrices for general *k*. The preceding sections have developed ideas that can now be brought together to give an asymptotically optimal algorithm for selecting a kth element in any set of sorted matrices $\{X_j\}$, $j = 1, \cdots, N$, of dimensions $n_j \times m_j$. As in the preceding sections we assume that $N > 1$ and $m_j \leqq n_j$. The algorithm follows the same pattern as before: preprocessing of the matrices, aggregation, and application of the basic algorithm to the aggregated problem. (See Fig. 2.) The preprocessing is done in three stages. First the matrices are cut down to dimensions $n_j' \times m_j'$, where $\sum_{j=1}^{N} (n_j' + m_j') \leqq 2k$, by applying the thin matrix algorithm to the set of all first rows and first columns of the given matrices.

ALGORITHM GENERAL_SELECT ($\{X_j\}, k$)

(1) Using the thin matrix algorithm, split the first row and first column of each matrix on a $2k$th element in this set so as to discard elements no smaller than the $2k$th element, retaining exactly $2k$ elements. In each X_j retain the origin-containing $n'_j \times m'_j$ submatrix induced by this splitting. Transpose matrices implicitly as necessary so that $n'_j \geqq m'j$ for all j.

(2) In the matrices that remain from step 1 define index set $Q = \{(s, t, j) | \ s$ and t are powers of 2 with $s \leqq n'_j, t \leqq m'_j, j = 1, \cdots, N\}$. Element $q \in Q$ has weight $w_q = st$. Let q^* be a $6k$th weighted element in Q according to w_q relative to a lexicographic order on the pairs (x_q, w_q). Let $\hat{k}_j = \sum \{w_q | q$ is in X'_j and does not follow q^* in the lexicographical order$\}$ for $j = 1, \cdots, N$.

(3) Use $\hat{k}_j$ in each X'_j to construct the set of logarithmically-dimensioned matrices as in § 4.

(4) Aggregate as in § 3. Then apply the basic algorithm to all the submatrices from step (3).

FIG. 2. *The general algorithm to select a kth element in any collection of N sorted matrices $\{X_j\}$ with dimensions $n_j \times m_j$.*

Next, consider a partition $\{k_j\}$ of k, for which a kth element in $\bigcup_i X_i$ has (virtual) rank k_j in X_j, for each j. We find upper bounds $\{\hat{k}_j\}$ for the $\{k_j\}$. The upper bounds $\{\hat{k}_j\}$, obtained by weighted selection in the matrices of dimensions $n'_j \times m'_j$, have the property $\sum_{j=1}^{N} \hat{k}_j = O(k)$ and therefore serve as estimates of $\{k_j\}$. The upper bounds can thus be used in the final stage of preprocessing to identify sets of submatrices in each X_j, as in § 4, to which the search for a kth element may be confined. It is this entire collection of submatrices from each given matrix that is aggregated for submission to the basic algorithm.

If necessary, the first step of preprocessing cuts the given matrices down to dimensions $n'_j \times m'_j$ where $\sum_{j=1}^{N} (n'_j + m'_j) \leqq 2k$. If $\sum_{j=1}^{N} (n_j + m_j) > 2k$, then a $2k$th element is selected in the set of all first rows and first columns of the given matrices. These rows and columns are thin matrices and thus the thin matrix algorithm of the preceding section can be used. The $2k$th element is then inserted into each first row and each first column to discard enough columns and rows that begin with elements at least as big as the $2k$th element so as to leave exactly $2k$ rows and columns altogether. When necessary the submatrices X'_j which remain are viewed as transposed to ensure that $n'_j \geqq m'_j$ for each j. A kth element in these submatrices will be a kth element in the original problem since no element smaller than the $2k$th is discarded and, when any element is discarded, at least $2k$ elements are retained.

Let $\mathcal{K} = \{k_j\}$ be a partition of k induced in $\{X_j\}$ by a kth element. The second step of the preprocessing generates a set $\{\hat{k}_j\}$ of upper bounds such that $k_j \leqq \hat{k}_j$ and $\sum_{j=1}^{N} \hat{k}_j < 6k$. This is accomplished by performing a weighted selection on a subset of elements from $\{X'_j\}$ with the following index set: $Q = \{(s, t, j) | \ s$ and t are powers of 2 with $s \leqq n'_j$ and $t \leqq m'_j, j = 1, \cdots, N\}$. The element x_q, with index $q = (s, t, j)$ is in the sth row and tth column of X'_j. It is accorded weight $w_q = st$ and is used to represent all elements in the cell (submatrix) C_q consisting of elements in rows s through $2s - 1$ and columns t through $2t - 1$. (The matrices are viewed as padded out to dimensions that are powers of 2.) Let q^* index a $6k$th weighted element, relative to a lexicographic order on the pairs (x_q, w_q). The weighted selection may be carried out in time linear in the cardinality of Q, using an algorithm from [JM].

Let Q' be the subset of all indices preceding q^* in the lexicographic order. By the property of the order, if (s, t, j) is in Q', then every (s', t', j) is in Q', where $s' \leqq s$ and $t' \leqq t$. Let $Q_L \subseteq Q'$ be the set of indices of cells containing some elements larger than x_{q^*}. Of these, cells in the first row or first column of any matrix account for weight no greater than $2k/2 = k$. Each remaining index (s, t, j) in $Q_L \cup \{q^*\}$ can be paired with index $(s/2, t/2, j)$ in $Q' - Q_L$, representing a cell with no elements greater

than x_{q^*}. Hence there are at least $(6k-k)/5=k$ elements no greater than x_{q^*} in cells indexed by Q'. Since there are no elements smaller than x_{q^*} in cells indexed by $Q-Q'$, a kth element in the cells index by Q' will be a kth element in $\{X_j\}$.

Let $\hat{k}_j$ be the sum of the weights, of cells in X_j indexed in Q'. A kth element in $\{X_j\}$ can have rank no greater than $\hat{k}_j$ in X_j, for each j. Thus in matrix X_j, the set of logarithmically dimensioned matrices based on $\hat{k}_j$ is formed, as was done for a matrix in § 4. This completes the preprocessing. Then the submatrices from all the X_j in $\{X_j\}$ are aggregated, and the basic algorithm is applied.

LEMMA 3. *Let* $\{(m_j, n_j)\}$, $j=1,\cdots,N$, *satisfy* $1\leqq m_j\leqq n_j$ *and* $\sum_{j=1}^{N}(m_j+n_j)\leqq 2k$. *There exists a partition* $\{k_j\}$ *of* k *for which* $k_j\leqq m_jn_j$ *and* $(\log m_j)(\log n_j)=O(h_j\log(2k_j/h_j^2))$ *for all* j, *where* $h_j=\min\{\sqrt{k_j}, m_j\}$.

Proof. It suffices to treat the case when $\sum_{j=1}^{N}(m_j+n_j)=2k$. Let $k_j=(m_j+n_j)/2$. Then $k_j\leqq m_jn_j$. Note also that $m_j\leqq k_j\leqq n_j<2k_j$. Thus

$$\begin{aligned}(\log m_j)(\log n_j)&<2(\log h_j)(\log 2k_j)\\&=2(\log h_j)(\log(2k_j/h_j^2))+4(\log h_j)^2\\&=O(h_j\log(2k_j/h_j^2)).\end{aligned}$$ □

We now extend the definition of a maximizing partition of k to a general set of sorted matrices. Call $\tilde{\mathcal{K}}=\{\tilde{k}_j\}$ a maximizing partition of k if $\tilde{\mathcal{K}}$ maximizes the expression $\sum_{h_j>0}(h_j\log(2\tilde{k}_j/h_j^2))$, where $h_j=\min\{\sqrt{\tilde{k}_j}, m_j\}$ for $j=1,\cdots,N$.

THEOREM 5. *Selection of a kth element in a collection of sorted matrices* $\{X_j\}$, $j=1,\cdots,N$, *in which* X_j *has dimensions* $n_i\times m_j$, $n_j\geqq m_j$, *can be solved in* $O(N+\sum_{h_j>0}h_j\log(2\tilde{k}_j/h_j^2))$ *time where* $\tilde{\mathcal{K}}=\{\tilde{k}_j\}$ *is a maximizing partition of k, and* $h_j=\min\{\sqrt{\tilde{k}_j}, m_j\}$, $j=1,\cdots,N$.

Proof. The algorithm of Fig. 2 has been shown to be correct in the preceding discussion. We now show that it runs within the stated bound.

By Theorem 4, selection in the set of $2N$ thin matrices will use time $O(2N+\sum_{j=1,\tilde{k}_j>0}^{2N}\log\tilde{k}_j')$, where $\tilde{\mathcal{K}}'=\{\tilde{k}_j'\}$ is a maximizing partition for $2k$. For a suitable indexing of $\tilde{\mathcal{K}}'$ compatible with the definition of a maximizing partition, this quantity is $O(N+\sum_{j=1}^{N}\log\tilde{k}_j')$ and in fact, within a constant factor, $\{\tilde{k}_j'\}_{j=1}^{N}$ can be taken to be a maximizing partition of k. To show that the application of the thin matrix algorithm is bounded asymptotically by the expression in the theorem it suffices to let $h_j'=\min\{\sqrt{\tilde{k}'}, m_j\}$. Then

$$\log\tilde{k}_j'=2\log\sqrt{\tilde{k}_j'}\leqq 2h_j'\log(\sqrt{\tilde{k}_j'}/h_j')=h_j'\log(\tilde{k}_j'/h_j'^2).$$

The selection in $N'\leqq N$ matrices using index set Q will use no more than $O(\sum(\log m_j')(\log n_j'))$ time. By Lemma 3, there is a partition $\{k_j'\}$ of k for which $(\log m_j')(\log n_j')=O(h_j'\log(2k_j'/h_j'^2))$ where $h_j'=\min\{\sqrt{k_j'}, m_j'\}$ for each j. Since $m_j'\leqq m_j$, the selection in Q is bounded by the expression in the statement of the theorem.

The aggregation and final selection can be shown to use $O(\sum_{\hat{k}_j>0}\hat{h}_j\log(2\hat{k}_j/\hat{h}_j^2))$, where $\hat{h}_j=\min\{\sqrt{\hat{k}_j}, m_j\}$, by an argument similar to that in Theorems 2 and 3. Since $\sum_{j=1}^{N}\hat{k}_j<6k$, this expression is bounded asymptotically by the expression in the statement of the theorem. □

The bound in Theorem 5 reduces to the bound for medians in single matrices (Theorem 1) when $N=1$, $k=\Theta(nm)$, and $m>1$, and the bound for medians in collections (Theorem 2) when $k=\Theta(\sum n_jm_j)$; it reduces to the bound for small k in a single matrix (Theorem 3) when $N=1$ and $m>1$; it reduces to the bound for collections of thin matrices (Theorem 4) when $m_j=1$ for all j. The bound of Theorem 5 is shown

in § 8 to be optimal on reduced inputs. Since reduction costs $O(N)$, it is reasonable to assume inputs will be reduced.

7. Ranking in a collection of sorted matrices. Ranking in a collection of sorted matrices may be done by ranking in each matrix independently and summing the ranks. Let t be the value to be ranked. We encode the value t as an element identified by index by creating a singleton matrix with the value t in it. By this reduction, ranking can be seen to be the complementary operation to selection. This reduction is also useful in deriving the lower bound in § 8.

To rank in a thin matrix, we use one-sided binary search [BY]. A one-sided binary search starts at the beginning of a sorted array instead of the middle, probing at indices $1, 2, 4, 8, \cdots, i$ until an element x_i greater than or equal to the test element t is found. The search is completed by performing an ordinary binary search in the range of indices $i/2$ to i to find i^* such that $x_{i^*} \leqq t < x_{i^*+1}$. The cost of a one-sided binary search is $O(\log(i^*+1))$.

Since it is not known on which side of the median t falls, the one-sided binary search should be applied in an interleaved fashion starting at both index 1 and index n_j in each thin matrix. A similar interleaving strategy may be applied to thick sorted matrices as we see below. To simplify presentation, we give only the portion for the case where t precedes the median.

In a sorted matrix X, the idea of one-sided binary search may be applied as follows. Our search first finds the insertion position in the first column, i.e., the largest i, such that $x_{i_1 1} \leqq t$. Then moving across row i_1 it finds the largest $x_{i_1 j_2}$ for which $x_{i_1 j_2} \leqq t$ at a cost of $O(\log(j_2 - j_1 + 2))$, etc., following the step-shaped boundary induced in X by the value t and finding the points at which the boundary "changes direction." Thus at a cost proportional to no more than

$$\sigma = \log(j_1+1) + \sum_{p=1}^{l-1} (\log(j_{p+1} - j_p + 2) + \log(i_p - i_{p+1} + 2))$$

a sequence of indices $(1 = i_1, j_1), (i_1, j_2), (i_2, j_2), (i_2, j_3), \cdots, (i_l, j_l = 1)$ is found from which the rank can be determined as $k = \sum_{p=1}^{l} (i_p - i_{p+1}) j_p$. We notice that $l \leqq \min\{\sqrt{2k}, m\}$.

THEOREM 6. *Ranking in a sorted matrix with dimensions n and m, $m \leqq n$, can be done in $O(h \log(2k/h^2))$ time when the element ranked has rank $k \leqq \lceil nm/2 \rceil$ and $h = \min\{\lceil \sqrt{k} \rceil, m\}$.*

Proof. We can maximize the contribution to σ of the i_p's when $j_l = l \leqq m$, giving

$$\sigma \leqq j_l \log 3 + 2 \cdot \sum_{p=1}^{l} \log(i_p - i_{p+1} + 2).$$

If $j_l = l$, then $k = \sum_{p=1}^{l} p(i_p - i_{p+1})$. We may then apply Lemma 4 below, to yield the desired result. □

It is important to note that $h \log(2k/h^2)$ is always $O(\sqrt{k})$, since $h \leqq \sqrt{k}$. When $k = \Theta(mn)$ the bound of Theorem 6 is $O(m \log(2n/m))$. Also, by symmetry, the algorithm may rank for kth smallest and $(nm-k)$th largest simultaneously, so that k in the above bound may be taken to be $\min\{k, nm-k\}$.

LEMMA 4. *Let a set of real numbers $\{a_i\}_{i=1}^{l}$, with $a_i \geqq 1$, satisfy $\sum_{i=1}^{l} i a_i = k$, for any $l \leqq m$. Then $\sum_{i=1}^{l} \lceil \log a_i \rceil = O(h \log(2k/h^2))$, where $h = \min\{\lceil \sqrt{k} \rceil, m\}$.*

Proof. It can be shown by induction that, for any fixed integer l, $\prod_{i=1}^{l} i a_i = \pi_l$ is maximized by $a_i = k/(il)$ for $i = 1, \cdots, l$. Thus

$$\pi_l = O\left(\frac{k^l}{(l^l) l!}\right).$$

By Stirling's formula, $\pi_l = O((1/\sqrt{2\pi l})(ke/l^2)^l)$ so $\ln \pi_l = O(l \ln (ke/l^2))$. Since $l<\sqrt{2k}$ and $l=O(h)$, we get $\ln \pi_l = O(h \ln (2k/h^2))$. Since $\sum_{i=1}^{l} \lceil \log a_i \rceil < l+\sum_{i=1}^{l} \log a_i$, and $l \leqq \min\{\lceil\sqrt{k}\rceil, m\}$, we obtain our result. □

To rank in a collection of sorted matrices, let t be identified by index in a singleton matrix. To allow for the case where k may exceed the median, let k be the rank of t in $\{X_j\}$ and let $\bar{k}$ be equal to $(\sum_{j=1}^{N} n_j)-k+1$. Then $\tilde{\mathcal{K}}$ is a maximizing partition of $\min\{k, \bar{k}\}$ if $\tilde{\mathcal{K}}$ maximizes the expression $\sum_{h_j>0}(h_j \log(2\tilde{k}_j/h_j^2))$, where $h_j = \min\{\sqrt{\tilde{k}_j}, m_j\}$ for $j=1, \cdots, N$. The general result for collections now follows immediately.

THEOREM 7. *Ranking of an element t in a collection of sorted matrices* $\{X_1, \cdots, X_N\}$, *where t is identified by index in a singleton matrix can be solved in* $O(N+\sum_{h_j>0}(h_j \log(2\tilde{k}_j/h_j^2)))$ *where* $\tilde{\mathcal{K}}=\{\tilde{k}_j\}$ *is a maximizing partition of* $\min\{k, \bar{k}\}$ *with respect to* $\{h_j\}$, *as defined above.*

8. Optimality. In this section we give lower bounds on running times for any algorithm that solves the selection or ranking problems treated in previous sections. These lower bounds coincide asymptotically with the running times of our algorithms, thus establishing the optimality of these algorithms in the context of a general comparison model of computation.

The lower bounds which we present are based on counting arguments in decision trees, extending the results of [FJ1] to the problems addressed here. We consider two varieties of decision trees. If algorithms are restricted to making comparisons between single input values, as is the case with every algorithm in this paper, then the lower bounds hold for any input domain with a total order. If comparisons are allowed between linear functions over the input values then our lower bounds hold for any dense domain on which a total order is defined on the set of all linear functions of values in the domain. Bounds on other problems have been obtained in a similar manner by [PY], [Yo], [JK], [FG], [FW].

Any finitely presented algorithm which solves a problem by means of the comparisons which we allow may be represented by a family of decision trees, one tree for all inputs of a given problem size and a given set of constraints on the total orders allowed for the inputs. (Certain algorithms which are not finitely presented may have this representation also.) Each tree contains interior nodes, representing comparisons, which have three children, one for each of the outcomes $<$, $=$ and $>$. Leaves of trees for selection or ranking are labeled with a single answer of the form "the kth element has value t." A selection tree is uniform for a given k, and at each leaf t is given by index in the input. A ranking tree is uniform for a t given by index, and at each leaf the correct k is given. A leaf that is reached over a path from the root on which only $<$ and $>$ branches are taken is called *strict*, as is also the path to a strict leaf.

For any input $X=\{X_i\}$ regardless of its structure, a *configuration* of X with respect to t drawn from the same domain is a set I_X for which $\{i|x_i<t\}\subset I_X \subset \{i|x_i \leqq t\}$. When all members of X are distinct, as we may assume in arguments for lower bounds, then I_X is uniquely specified by X and t. We call an input with distinct members a *simple* input.

We need two results which we present together in the following lemma.

LEMMA 5. *Any algorithm that can determine that value t is kth in inputs where all values are distinct*

(a) *using only elementary comparisons and over any input domain with a total order, or*

(b) *using only comparisons between linear forms of the input and over any dense input domain with a total order defined on the comparands*

must expend $\Omega(\log P)$ *comparisons, where* P *is the number of distinct input configurations allowed by the parameters and constraints of the problem.*

Proof. It is easily shown by an adversary argument that any algorithm must determine $x_i < t$, $x_i = t$, or $x_i > t$ for every i. Thus for part (a), there must be P leaves and some path of length $\lceil \log P \rceil$, because if two simple inputs with distinct configurations were to choose the same leaf it will be asserted that $x_i < t$ and $x_i \geqq t$ for some single input value x_i.

For part (b), assume some strict leaf is reached by two simple inputs X' and X'' the configurations of which are distinct. Hence the decision tree identifies the jth element in each as equal to t. Since the inequalities specified by the path to the leaf in question define an open and convex set in the input domain, every linear combination of the inputs X' and X'' also select the same leaf. As in [FJ1] it is easily shown that there is a possible input Y in an ε-neighborhood of a linear combination of X' and X'' for which the configuration I_Y satisfies $|I_Y| = k+1$, contradicting the correctness of the tree. It follows that each strict leaf accepts inputs with exactly one configuration. Therefore there are at least P leaves and some path of length $\lceil \log P \rceil$. □

It is convenient to define the following. Given a maximizing partition $\tilde{\mathscr{K}}$ of $k \leqq \lceil \frac{1}{2} \sum_{j=1}^{N} n_j m_j \rceil$, a partition $\mathscr{K} = \{k_j\}$ can be constructed which satisfies $\lfloor \tilde{k}_j/2 \rfloor \leqq k_j \leqq \lceil n_j m_j/2 \rceil$ for all j. Any partition so constructed is *asymptotically maximizing* since it maximizes the expression $N + \sum_{h_j>0} h_j \log(2k_j/h_j^2)$ to within a constant factor. Also, let the largest thin matrix be identified by index i_L.

LEMMA 6. *Selection of a kth element in a collection of thin sorted matrices* $\{X_1, \cdots, X_N\}$, *of dimensions* $n_j \times 1$, *requires time*

$$\Omega\Big(N + \sum_{\substack{\tilde{k}_j > 0 \\ j \neq i_L}} \log \tilde{k}_j\Big)$$

where $\tilde{\mathscr{K}} = \{\tilde{k}_j\}$ *is a maximizing partition of k.*

Proof. Let $\{x_j\}$ be indexed so that $i_L = 1$ and $k_j \geqq k_{j+1}$, $j = 1, \cdots, N-1$, for an asymptotically maximizing partition $\{k_j\}$. Let $\hat{j} = \min\{j \mid \sum_{l=j+1}^{N} k_l \leqq (k-k_1)/2\}$ and $\sigma = \sum_{l=\hat{j}+1}^{N} k_l$. Configurations of the form $\{(i_j, j) \mid j = 1, \cdots, N\}$ are achievable for $0 \leqq i_j \leqq k_j$, $j > \hat{j}$, and $d \leqq i_{\hat{j}} \leqq d + (k-k_1)/2 - \sigma$, for d a function of $\sum_{j<\hat{j}} i_j$. (To be specific, $d = \max\{0, k - \sum_{j<\hat{j}} n_j - \sum_{j>\hat{j}} i_j\}$.) Thus the number P of achievable configurations satisfies $P \geqq ((k-k_1)/2 - \sigma) \prod_{j>\hat{j}} (k_j + 1)$. Since this expression is no smaller than either $\prod_{j=2}^{\hat{j}-1} (k_j + 1)$ or $k_{\hat{j}} + 1$, we have $P^3 \geqq \sum_{j=2}^{N} (k_j + 1)$, from which it follows that

$$\log P = \Omega\Big(N + \sum_{\substack{\tilde{k}_j > 0 \\ j \neq i_L}} \log \tilde{k}_j\Big),$$

since $\{k_j\}$ is asymptotically maximizing. □

LEMMA 7. *Selection of a kth element in a sorted matrix* X *of dimension* $n \times m$, $1 < m \leqq n$, *requires time* $\Omega(h \log(2k/h^2))$, *where* $h = \min\{\sqrt{k}, m\}$ *and* $k \leqq \lceil nm/2 \rceil$.

Proof. We construct a basic "step-shaped" configuration as follows. If $m = 2$, an initial configuration will have k elements in column 1. A valid configuration may be obtained by moving up to $\lfloor k/2 \rfloor$ elements into column 2. If $m = 3$, an initial configuration will have $\lfloor 2k/3 \rfloor$ elements in column 1 and $\lceil k/3 \rceil$ elements in column 2. This allows moving up to $\lfloor k/3 \rfloor$ elements from column 1 to column 3. So, for the cases $m = 2$ and $m = 3$, $P = \Omega(k)$ and $\log P = \Omega(\log k)$, which is $\Omega(h \log(2k/h^2))$.

For $m \geqq 4$, let $a = \min\{m, \lceil \sqrt{2k} \rceil\}$, $s = \lfloor 2k/a^2 \rfloor$, and $b = \lfloor 2k/a \rfloor - \lceil sa/4 \rceil$. The basic configuration has $b + s(a+1-i)$ elements in each column, $i = 1, \cdots, \lfloor a/2 \rfloor - 1$,

and no more than $b+s(a+1-i)$ elements in each column, $i=\lfloor a/2\rfloor, \cdots, a$. Thus it is possible to move up to s elements from any column $i=1, \cdots, \lfloor a/2\rfloor-1$ to some column with index greater than $\lfloor a/2\rfloor$, and this may be done independently for each of the first $\lfloor a/2\rfloor-1$ columns. Therefore $P\geqq s^{\lfloor a/2\rfloor-1}$ and $\log P=\Omega(h\log(2k/h^2))$ as required. We now verify that the basic configuration claimed exists for all values of k, m, and n for which $1\leqq k\leqq\lceil nm/2\rceil$ and $n\geqq m\geqq 4$.

It is sufficient to show the following for $m\geqq 4$,

$$\sum_{i=1}^{\lfloor a/2\rfloor-1}(is+b)\leqq k\leqq\sum_{i=1}^{a}(is+b).$$

To obtain the first inequality, we obtain

$$b=\lfloor 2k/a\rfloor-\lceil sa/4\rceil\leqq\lfloor k/\lfloor a/2\rfloor\rfloor-\lceil s\lfloor a/2\rfloor/2\rceil\leqq k/(\lfloor a/2\rfloor-1)-s\lfloor a/2\rfloor/2,$$

which implies

$$s\lfloor a/2\rfloor(\lfloor a/2\rfloor-1)/2+b(\lfloor a/2\rfloor-1)\leqq k,$$

which is the first inequality in closed form. As to the second, since $a\geqq 2$ and $k/a\geqq 1/2$,

$$b=\lfloor 2k/a\rfloor-\lceil sa/4\rceil\geqq k/a-s(a+1)/2,$$

which implies

$$k\leqq\frac{sa(a+1)}{2}+ab,$$

the closed form for the second inequality. □

We may now combine these results to obtain lower bounds for the general problems.

THEOREM 8. *Selection of a kth element in a collection of sorted matrices* $\{X_1,\cdots,X_N\}$ *in which matrix* X_j *has dimensions* $n_j\times m_j$, $m_j\leqq n_j\leqq k$, *is of complexity*

$$\Theta\Big(n+\sum_{h_j>0}h_j\log(\tilde{k}_j/h_j^2)\Big)$$

where $\tilde{\mathcal{K}}=\{\tilde{k}_j\}$ *is a maximizing partition of* $k\leqq\lceil\sum_{j=1}^N m_jn_j/2\rceil$, *and* $h_j=\min\{\sqrt{\tilde{k}_j}, m_j\}$ *for* $j=1,\cdots,N$.

Proof. The upper bound has been established in Theorem 5. For the lower bound, consider an asymptotically maximizing partition $\mathcal{K}=\{k_j\}$ of k. Let $I\subset\{1,\cdots,N\}$ be the set indexing the thin matrices of the problem, with i_L indexing the largest. Consider selection of the $(\sum_{j\in I}k_j)$th element in $\{X_j\}_{j\in I}$. By Lemma 6, there are P_T configurations satisfying $\log P_T=\Omega(\sum_{k_j>0,j\neq i_L}\log k_j)$. For each remaining matrix X_j with $k_j>0$, there are P_j configurations, where $\log P_j=\Omega(h_j\log(2k_j/h_j^2))$, and $h_j=\min\{\sqrt{k_j}, m_j\}$. Since the configurations of the collection of thin matrices and of the individual thick matrices are independent, we may multiply to get a total number of configurations. Since the problem is reduced, $k_{i_L}\leqq\lceil\frac{1}{2}n_{i_L}\rceil\leqq\lceil k/2\rceil$, and thus we get $\log P=\Omega(\sum_{h_j>0}h_j\log(2k_j/h_j^2))$. The $\Omega(N)$ term arises for case (a) of Lemma 5 because each matrix must be examined, and for case (b) it follows from Rabin's results [R]. □

COROLLARY 1. *The algorithm of reduction, as given in the introduction, followed by Algorithm* GENERAL_SELECT (*see Fig.* 2) *is optimal on any selection problem on sorted matrices.*

Proof. This result follows directly from Theorem 8 and the fact that reduction to satisfy $m_j\leqq n_j\leqq k\leqq\lceil\frac{1}{2}\sum_{j=1}^N m_jn_j\rceil$ can be accomplished in $O(N)$ time. □

THEOREM 9. *Ranking an element t in a collection of sorted matrices* $\{X_1, \cdots, X_N\}$, *where t is identified by index in a singleton matrix, requires* $\Omega(N + \sum_{h_j > 0} h_j \log(2\bar{k}_j / h_j^2))$ *time, where* $\mathscr{K} = \{k_j\}$ *is the collection of ranks induced by t,* $\bar{k}_j = \min\{k_j, m_j n_j - k_j\}$, *and* $h_j = \min\{\sqrt{\bar{k}_j}, m_j\}$, $j = 1, \cdots, N$.

Proof. The lower bound is constituted independently for each matrix, thick or thin. For a thin matrix, there are $\lfloor k_j/2 \rfloor$ positions i, $\lceil k_j/2 \rceil < i \leqq k_j$. Hence searching for an element with rank in this range will require $\Omega(\log(k_j + 1))$, for $k_j \leqq \lceil n_j/2 \rceil$. For a thick matrix, configurations may be constructed and counted as in Lemma 7, but using actual values k_j. The additive term N arises as in the proof of Theorem 8. □

REFERENCES

[BY] J. L. BENTLEY AND A. C. YAO, *An almost optimal algorithm for unbounded searching*, Inform. Proc. Letters, 5 (1976), pp. 82–87.

[B] M. BLUM, R. W. FLOYD, V. R. PRATT, R. L. RIVEST AND R. E. TARJAN, *Time bounds for selection*, J. Comput. Syst. Sci., 7 (1972), pp. 448–461.

[CT1] R. CHANDRASEKARAN AND A. TAMIR, *Polynomially bounded algorithms for locating P-centers on a tree*, Math. Prog., 22 (1982), pp. 304–315.

[CT2] ———, *An* $O((n \log P)^2)$ *algorithm for the continuous P-center problem on a tree*, SIAM J. Alg. Disc. Meth., 1 (1980), pp. 370–375.

[DM] D. DOBKIN AND J. I. MUNRO, private communication.

[FJ1] G. N. FREDERICKSON AND D. B. JOHNSON, *The complexity of selection and ranking in* $X + Y$ *and matrices with sorted columns*, J. Comput. Syst. Sci., 24 (1982), pp. 197–208.

[FJ2] ———, *Finding kth paths and p-centers by generating and searching good data structures*, J. Algorithms, 4 (1983), pp. 61–80.

[FJ3] ———, *Generalized selection and ranking*, Proc. 12th Annual ACM Symposium on Theory of Computing, Los Angeles (April 1980), pp. 420–428.

[FW] M. L. FREDMAN AND B. WEIDE, *On the complexity of computing the measure of* $\bigcup [a_i, b_i]$, Comm. ACM, 21 (1979), pp. 227–238.

[FG] F. FUSSENEGGER AND H. N. GABOW, *A counting approach to lower bounds for selection problems*, J. Assoc. Comput. Mach., 26 (1979), pp. 540–543.

[GM] Z. GALIL AND N. MEGIDDO, *A fast selection algorithm and the problem of optimum distribution of effort*, J. Assoc. Comput. Mach., 26 (1979), pp. 58–64.

[HL] J. L. HODGES AND E. L. LEHMANN, *Estimates of location based on rank tests*, Ann. Math. Statist., 34 (1963), pp. 598–611.

[H] L. HYAFIL, *Bounds for selection*, this Journal, 5 (March 1976), pp. 109–114.

[JK] D. B. JOHNSON AND S. D. KASHDAN, *Lower bounds for selection in* $X + Y$ *and other multisets*, J. Assoc. Comput. Mach., 25 (1978), pp. 556–570.

[JM] D. B. JOHNSON AND T. MIZOGUCHI, *Selecting the Kth element in* $X + Y$ *and* $X_1 + X_2 + \cdots + X_m$, this Journal, 7 (1978), pp. 147–153.

[JR] D. B. JOHNSON AND T. A. RYAN, JR., *Fast computation of the Hodges Lehmann estimator—Theory and practice*, Proc. Ann. Statistical Assoc. 138th Ann. Meeting, San Diego, 1978, pp. 1–2.

[KH] O. KARIV AND S. L. HAKIMI, *An algorithmic approach to network location problems* I: *The p-centers*, SIAM J. Appl. Math., 37 (1979), pp. 513–538.

[Ki] D. G. KIRKPATRICK, *A unified lower bound for selection and set partitioning problems*, J. Assoc. Comput. Mach., 28 (1981), pp. 150–165.

[Ko] B. O. KOOPMAN, *The optimum distribution of effort*, J. Oper. Res. Soc. Amer., 1 (1953), pp. 52–63.

[MR] J. W. MCKEAN AND T. A. RYAN, JR., ALGORITHM 516: *An algorithm for obtaining confidence intervals and point estimates based on ranks in a two-sample location problem*, ACM Trans. Math. Software, 3 (June 1977), pp. 183–185.

[MTZC] N. MEGIDDO, A. TAMIR, E. ZEMEL AND R. CHANDRASEKARAN, *An* $O(n \log^2 n)$ *algorithm for the kth longest path in a tree with applications to location problems*, this Journal, 10 (1981), pp. 328–337.

[PY] V. R. PRATT AND F. F. YAO, *On lower bounds for computing the ith largest element*, Proc. 14th Annual Symposium Switching and Automata Theory, Iowa City, 1973, pp. 70–81.

[R] M. O. RABIN, *Proving simultaneous positivity of linear forms*, J. Comput. Syst. Sci., 6 (1972), pp. 639–650.

[SPP] A. SCHÖNHAGE, M. PATERSON AND N. PIPPENGER, *Finding the median*, J. Comput. Syst. Sci. 13 (1976), pp. 184–199.

[S] M. I. SHAMOS, *Geometry and statistics: problems at the interface*, in Algorithms and Complexity: New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York, 1976, pp. 251–280.

[YO] A. C.-C. YAO, *On the complexity of comparison problems using linear functions*, Proc. 16th Annual ACM Symposium Foundations of Computer Science, Berkeley, CA, 1975, pp. 85–89.

[Yp] C. K. YAP, *New upper bounds for selection*, Comm. ACM, 19 (1976), pp. 501–508.

SIAM J. COMPUT.
Vol. 13, No. 1, February 1984

© 1984 Society for Industrial and Applied Mathematics
0097-5397/84/1301-0003 $01.25/0

LINEAR TIME ALGORITHMS FOR TWO- AND THREE-VARIABLE LINEAR PROGRAMS*

M. E. DYER†

Abstract. $O(n)$ time algorithms for linear programming problems with two or three variables and n constraints are described. The approach uses convexity, dominance of linear functions and linear-time median finding algorithms. The algorithms improve the previously known best bounds of $O(n \log n)$ time for both of these problems.

Key words. linear programming, computational complexity, geometric algorithm

1. Introduction. Algorithms for linear programming (LP) have recently been a subject of great interest. Khachiyan [6] has shown that LPs are solvable in time polynomial in the total number of digits in the problem description, and this result clearly places LP in the class of "easy" computational problems. However it may be argued that the model within which Khachiyan's algorithm is polynomial is not entirely satisfactory for LP, since it deals at a very low level with arithmetic precision in the data. A more appropriate model is perhaps the RAM with real arithmetic, commonly used in the analysis of geometric algorithms (see, for example, [11]). In this model the atomic operations are arithmetic, comparisons and accesses of real numbers. (Since real numbers are essentially infinite objects it may be observed that this approach does beg some important questions about the nature of the computations, but this will not be considered here.) Within this model, we seek an LP algorithm for which the complexity is polynomial only in the numbers of constraints and variables. However at present no such algorithm is known. Even if such an algorithm is discovered, it will still, of course, be of interest to minimize the degree of the polynomial involved.

This paper considers LP under the real-arithmetic RAM model in the simplest nontrivial cases, where the numbers of variables are either two or three. For such a problem with n constraints, it is shown that algorithms requiring only $O(n)$ time and space are available. The approach uses convexity, dominance of linear functions, and well-known ideas for the design of linear time algorithms.[1] The paper is divided into two main sections. Section 2 deals with the two-variable case, and then § 3 generalizes this to the three-variable problem.

2. Two-variable LP. This section examines the solution of the two-variable LP problem:

$$\text{maximize} \quad c_1x_1+c_2x_2,$$

$$\text{subject to} \quad a_{i1}x_1+a_{i2}x_2 \leqq b_i \qquad (i=1,2,\cdots,n),$$

where c_1, c_2, a_{i1}, a_{i2}, b_i are given constants.

The simplex method can be used to solve this problem in time $O(n^2)$. However, it has been known for some time (see [11]) that an $O(n \log n)$ time solution is possible, based on determining all the $O(n)$ vertices of the constraint set. From results concerning convex hull determination [13], it would seem that this approach is also $\Omega(n \log n)$,

* Received by the editors April 2, 1982, and in revised form December 2, 1982.

† Department of Mathematics and Statistics, Teesside Polytechnic, Middlesbrough, Cleveland, United Kingdom TS1 3BA.

[1] A similar approach to LP problems has been discovered independently by N. Megiddo.

though Bentley and Shamos [1] have shown that an $O(n)$ expected time algorithm, under various distributions for the inputs, is obtainable in this way.

Here we continue this work by showing that an $O(n)$ worst case algorithm is available for this problem. Such a method is clearly optimal to within a constant factor. The method is based on minimizing convex functions defined as the maximum of a set of linear functions on the real line, R. The section is divided as follows. Section 2.1 considers the fundamental minimax problem just described, and presents the basic pairing algorithm. Then § 2.2 concerns the solution of the LP when a point satisfying the constraints is known. Finally § 2.3 considers how such a point may be determined. All algorithms work in linear time.

Note that the problem considered here is not closely related to the "two-variable per constraint" linear programming problem (see, e.g., [12]) which has also been referred to as two-variable LP.

2.1. The pairing algorithm. Suppose a function F is defined on R by:

$$F(x) = \max_{1 \leqq i \leqq n} \{\alpha_i x + \beta_i\}$$

where $(\alpha_i x + \beta_i)$ are given lines (i.e., linear functions of x). It is well known that F is a convex function [10]. Consider the problem of determining

$$F(x_0) = \min_{x \in R} F(x).$$

For notational convenience, it will be assumed that we are working in the extended real line $\bar{R}$, i.e., R augmented with the symbols $+\infty$ and $-\infty$ (see [10]). In any closed interval of $\bar{R}$, we will say that the ith of the n lines $(\alpha_i x + \beta_i)$ is *dominated* in I if for all $x \in I$ there is a j such that $\alpha_j x + \beta_j \geqq \alpha_i x + \beta_i$ and either $(\alpha_j, \beta_j) \neq (\alpha_i, \beta_i)$ or $(\alpha_j, \beta_j) = (\alpha_i, \beta_i)$ and $j < i$. Thus, a line is dominated if there is another line which lies above it at all points of I. Otherwise line i is undominated in I. It is straightforward to show that F has the following form. There is a partition of $\bar{R}$ into $(m+1) \leqq n$ closed intervals $I_r = \{x : z_r \leqq x \leqq z_{r+1}\}$ $(r = 0, 1, \cdots, m)$, with $z_0 = -\infty$, $z_{m+1} = +\infty$. In each I_r there is a single undominated line $(\alpha_r^* x + \beta_r^*)$. Then $F(x) = \alpha_r^* x + \beta_r^*$ $(x \in I_r)$. By the convexity of F, $\{\alpha_r^*\}$ is an increasing sequence. Now x_0 is clearly one of the z_r since F is linear (and hence monotonic) between such values. First, let us exclude the cases $x_0 = \pm\infty$. From the above, it follows that $x_0 = +\infty$ if and only if $\max \alpha_i < 0$, and $x_0 = -\infty$ if and only if $\min \alpha_i > 0$. These conditions can be detected in $O(n)$ time and hence it will be assumed that x_0 is finite (or equivalently $\min \alpha_i \leqq 0 \leqq \max \alpha_i$). It follows that F is as shown in Fig. 1 in the neighborhood of x_0.

Clearly, as far as the determination of x_0 is concerned, only two lines are relevant, those two which intersect at x_0 and are undominated in every neighborhood of x_0. Any line which is dominated in any neighborhood of x_0 is irrelevant for this purpose and could as well be removed altogether from the definition of F. This observation, together with the convexity of F, provides the basis for the following algorithm, in which lines are successively deleted from the definition of F by dominance. In the algorithm, F is assumed to be represented by a list of length n, giving the α_i, β_i. Each element of the list has a third data field x_i which is used in the algorithm. Note that as deletions occur from the F-list, n is reduced. Thus, in the algorithm description, n is to be read as referring to the current length of this list at the commencement of Step 1 of the iteration.

Step 0. Check if $\min \alpha_i > 0$. If so, set $x_0 \leftarrow -\infty$, $F(x_0) \leftarrow -\infty$ and stop. Check if $\max \alpha_i < 0$, if so, set $x_0 \leftarrow +\infty$ and stop.

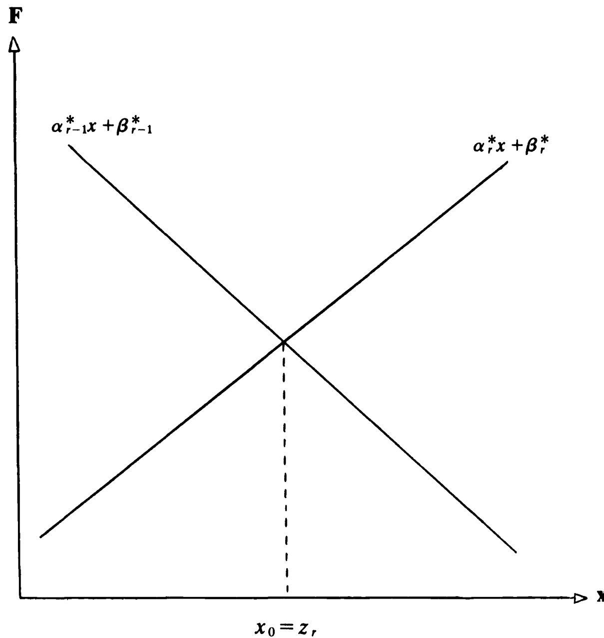


FIG. 1

Step 1. Proceed through the list, dividing the n lines into $\lfloor\frac{1}{2}n\rfloor$ pairs of successive lines. For each pair $(\alpha_i x+\beta_i)$, $(\alpha_{i+1}x+\beta_{i+1})$ perform the following operations. If the lines are parallel, i.e. $\alpha_i=\alpha_{i+1}$, then if $\beta_i<\beta_{i+1}$, delete line i, otherwise delete line $(i+1)$. (One or the other of the lines is dominated on R.) Otherwise, calculate the (unique) intersection x_i of lines i, $(i+1)$. Thus $x_i\leftarrow(\beta_i-\beta_{i+1})/(\alpha_{i+1}-\alpha_i)$. ($x_{i+1}$ will be left undefined.) Observe that if the *left half* of the pair i, $(i+1)$ is the line corresponding to $\min(\alpha_i,\alpha_{i+1})$ and the *right half* is that corresponding to $\max(\alpha_i,\alpha_{i+1})$, then the right half dominates the left half on the interval $\{x\geqq x_i\}$, and the left half dominates the right half on $\{x\leqq x_i\}$ (cf. Fig. 1).

Step 2. Let $k\leqq\lfloor n/2\rfloor$ be the number of values x_i created in Step 1. If $k=0$, return to Step 1. Note that $\lfloor\frac{1}{2}n\rfloor$ lines will have been deleted from the list. Otherwise, using a linear-time median finding algorithm [2] determine the median x^* of the k values x_i (i.e., x^* is the $\lceil\frac{1}{2}k\rceil$th x_i in ranked order).

Step 3. Evaluate F and its left and right gradients λ,ρ at x^*. That is, determine

$$F(x^*)=\max_{1\leqq j\leqq n}\{\alpha_j x^*+\beta_j\}$$

and

$$\lambda=\min\{\alpha_j\colon \alpha_j x^*+\beta_j=F(x^*)\},$$

$$\rho=\max\{\alpha_j\colon \alpha_j x^*+\beta_j=F(x^*)\}.$$

Clearly, $\lambda\leqq\rho$ and equality is possible. By the convexity of F, these can be used to determine whether $x_0<x^*$, $x_0=x^*$, or $x_0>x^*$.

Step 4. If $\lambda\leqq 0\leqq\rho$, x^* is minimal. Thus, set $x_0\leftarrow x^*$, $F(x_0)\leftarrow F(x^*)$ and stop. Else if $\lambda>0$, then $x_0<x^*$ and thus $x_0<x_i$ for all $x_i\geqq x^*$. Thus the right half of pair i, $(i+1)$ is dominated in a neighborhood of x_0. Therefore, go through the F-list deleting the

right half of all pairs $i, (i+1)$ having $x_i \geqq x^*$. Clearly $\lceil \frac{1}{2}k \rceil$ lines are deleted, and thus it follows that, in Steps 1 and 4, at least $\lceil \frac{1}{2} \lfloor \frac{1}{2}n \rfloor \rceil = \lceil \frac{1}{4}(n-1) \rceil$ lines are deleted. Return to Step 1. Similarly if $\rho < 0, x_0 > x^*$. In this case, delete the left half of $i, (i+1)$ for $x_i \leqq x^*$ and return to Step 1. This completes the description of the algorithm.

The algorithm will eventually terminate in Step 4, since at least $\lceil \frac{1}{4}(n-1) \rceil$ lines are deleted at each iteration of Steps 1 to 4. Thus, at least one line is deleted when $n > 2$. When $n = 2$, the algorithm must terminate since x^* is then x_0, the unique intersection of the two remaining undominated lines. An examination of each of Steps 1 to 4 shows that they can be carried out in $O(n)$ time, where n is used here as in the algorithm. In each such iteration, approximately a quarter (at least) of the F list is deleted. Therefore, returning to the original meaning of n as the initial number of lines, only $O(\log n)$ iterations can occur. Since Step 0 clearly takes $O(n)$ time, the total time for the algorithm is

$$O(n + \tfrac{3}{4}n + (\tfrac{3}{4})^2 n + \cdots) = O(n).$$

It is also obvious that any algorithm for this problem is $\Omega(n)$ and thus the pairing algorithm is essentially optimal.

2.2. Solving the linear program. We return now to the solution of the LP:

$$\begin{aligned} &\text{maximize} && c_1x_1 + c_2x_2, \\ &\text{subject to} && a_{i1}x_1 + a_{i2}x_2 \leqq b_i \qquad (i = 1, 2, \cdots, n), \end{aligned}$$

when a point x_1', x_2' satisfying all the inequalities is given. (The determination of such a point is considered in the next section.)

First, note that if $c_1 = c_2 = 0$, then x_1', x_2' is an optimal point, and hence this case can be detected and discarded in constant time. It is, therefore, assumed that $c_2 \neq 0$, if necessary after applying the transformation

$$x_1' \leftrightarrow x_2', \quad c_1 \leftrightarrow c_2, \quad a_{i1} \leftrightarrow a_{i2}$$

to the data in $O(n)$ time. (The transformed problem is solved and the optimal solution to the original problem obtained by interchanging its x_1 and x_2 coordinates.) The problem is now transformed into a form similar to that of § 2.1. The transformation can be carried out in one pass through the data, though for clarity it will be presented here in three parts. Each takes $O(n)$ time.

First make the transformation

$$X_1 = x_1 - x_1', \qquad X_2 = x_2 - x_2'$$

in order to map x_1', x_2' to the origin. This results in the problem:

$$\begin{aligned} &\text{maximize} && c_0 + c_1X_1 + c_2X_2, \\ &\text{subject to} && a_{i1}X_1 + a_{i2}X_2 \leqq B_i \qquad (i = 1, 2, \cdots, n) \end{aligned}$$

where $c_0 = c_1x_1' + c_2x_2', B_i = b_i - a_{i1}x_1' - a_{i2}x_2'$.

The constant c_0 can be added in once the problem is solved, and hence, it will be ignored. Now the problem is transformed to bring the gradient direction of the objective function to the ordinate axis:

$$X = X_1, \qquad Y = c_1X_1 + c_2X_2.$$

This is nonsingular since $c_2 \neq 0$. The transformed problem is:

maximize $\quad Y,$

subject to $\quad A_{i1}X + A_{i2}Y \leqq B_i \qquad (i = 1, 2, \cdots, n),$

where $A_{i1} = a_{i1} - a_{i2}c_1/c_2$, $A_{i2} = a_{i2}/c_2$.

Now, since $X = Y = 0$ satisfies the constraints of this problem, we must have $B_i \geqq 0$ for all i. Also, since this known solution has $Y = 0$, we can assume $Y \geqq 0$ in the optimal solution. There is, therefore, no loss in adding this as a constraint to the problem. Now, the third transformation is projective:

$$x = \frac{X}{Y}, \qquad y = \frac{1}{Y},$$

which produces:

minimize $\quad y,$

subject to $\quad A_{i1}x + A_{i2} \leqq B_i y,$

since maximizing Y is equivalent to minimizing its reciprocal for $Y \geqq 0$. The inverse of the three transformations applied can be readily determined. It is

$$x_1 = x_1' + \left(\frac{x}{y}\right), \qquad x_2 = x_2' + \frac{1 - c_1 x}{c_2 y}.$$

This will be needed to reinstate the solution to the original problem. The value of $(c_1x_1 + c_2x_2)$ can then be calculated.

In the above transformed problem, it is necessary to partition the constraints into those with $B_i = 0$ and $B_i > 0$. Let these be the sets I_0, I_1 respectively. The constraints in I_0 can be reduced to: $u_1 \leqq x \leqq u_2$, where

$$u_1 = \max\left\{-\frac{A_{i2}}{A_{i1}}: A_{i1} > 0, B_i = 0\right\}$$

and

$$u_2 = \min\left\{-\frac{A_{i2}}{A_{i1}}: A_{i1} < 0, B_i = 0\right\}.$$

The constraints in I_1 can be rewritten:

$$y \geqq \alpha_i x + \beta_i \qquad (i \in I_1),$$

where

$$\alpha_i = \frac{A_{i1}}{B_i}, \qquad \beta_i = \frac{A_{i2}}{B_i}.$$

Therefore, the problem can finally be stated:

minimize $\quad y = F(x)$

where $\quad F(x) = \max_{i \in I_1} \{\alpha_i x + \beta_i\}$ and $u_1 \leqq x \leqq u_2$.

This problem is a slight generalization of that of § 2.1 in that the interval $U = [u_1, u_2]$, from which x must be chosen, may be finite or semi-infinite. However, the pairing algorithm is easily modified to deal with this problem, as follows.

First, if $u_1 > u_2$, then $\min y = +\infty$ and it follows that the solution to the original problem is $x_1 = x'_1$, $x_2 = x'_2$. This can be checked in Step 0. Next, if $u_1 = u_2$, then the solution is $x = u_1$, and $y = F(u_1)$ must be evaluated in $O(n)$ time. Again, this can be done in Step 0. This leaves the case $u_1 < u_2$. The following changes can be made in Step 0. Evaluate $F(u_1)$ and its right gradient ρ, as described in Step 4 of the pairing algorithm. If $\rho > 0$, stop, the optimal solution is $x = u_1$, $y = F(u_1)$. Otherwise, determine $F(u_2)$ and its left gradient λ. If $\lambda < 0$, stop, the optimal solution is $x = u_2$, $y = F(u_2)$. Otherwise the optimum x lies in the interior of U. Modify Step 1 as follows. If, in the pairing process, $x_i \geqq u_2$, then delete the right half of the pair and ignore this x_i in the determination of x^* in Step 2. Similarly, if $x_i \leqq u_1$, then delete the left half of the pair and ignore this x_i. In both cases, the deleted line is dominated on U. Note that with these modifications, $u_1 < x^* < u_2$ at each iteration. It is clear that the transformation process can be carried out in linear time, and that the modifications to the pairing algorithm do not alter its linear-time behaviour. Thus the algorithm can be carried out in time $O(n)$.

2.3. Finding a feasible point. It remains to consider how a point satisfying the constraints

$$a_{i1}x_1 + a_{i2}x_2 \leqq b_i \qquad (i = 1, 2, \cdots, n)$$

can be determined, or the nonexistence of any such point proven. First observe that if any constraint has $a_{i1} = a_{i2} = 0$ then, if $b_i < 0$, there is no solution to the constraints. Otherwise, the constraint is redundant and can be deleted. Such constraints can obviously be processed in linear time and thus it will be assumed that no such constraints are present. Now the constraints will be partitioned into three sets, I_0, I_1, I_2, corresponding respectively to $a_{i2} = 0$, $a_{i2} < 0$ and $a_{i2} > 0$. The constraints can now be rewritten:

$$x_2 \geqq \alpha_i x_1 + \beta_i \qquad (i \in I_1),$$

$$-x_2 \geqq \alpha_i x_1 + \beta_i \qquad (i \in I_2),$$

$$u_1 \leqq x_1 \leqq u_2,$$

where

$$\alpha_i = a_{i1}/|a_{i2}|, \qquad \beta_i = -b_i/|a_{i2}|,$$

$$u_1 = \max \{b_i/a_{i1}: a_{i1} < 0, i \in I_0\},$$

$$u_2 = \min \{b_i/a_{i1}: a_{i1} > 0, i \in I_0\}.$$

Writing $x_1 = x$, $F_j(x) = \max_{i \in I_j} \{\alpha_i x + \beta_i\}$ $(j = 1, 2)$ and $U = [u_1, u_2]$ the constraints can thus be expressed in the form

$$F_1(x) \leqq x_2 \leqq -F_2(x) \quad \text{and} \quad x \in U.$$

There is a point satisfying the constraints if there exists $x \in U$ such that $-F_2(x) \geqq F_1(x)$ or equivalently $\delta(x) = F_1(x) + F_2(x) \leqq 0$. (A point satisfying the original constraints is then given by $x_1 = x$ and $x_2 = \frac{1}{2}(F_1(x) - F_2(x))$.) Since F_1 and F_2 are convex functions (of the form considered in § 2.1), it follows that δ is also convex. The problem of finding a feasible point can therefore be reduced to the problem:

determine $\quad \delta(x_0) = \min \{\delta(x): x \in U\}$.

If $\delta(x_0) \leqq 0$, we can find a point satisfying the constraints, otherwise there is no such point. In fact, noting that if $u_1 > u_2$ there is no solution, and if $u_1 = u_2$, the constraint

set is included in the line $x_1 = u_1$ (and thus the problem is essentially one-variable), it follows that the constraint set has an interior point if and only if $\delta(x_0) < 0$ and $u_1 < u_2$. The above rule will then give an interior point. The pairing algorithm, with the amendments of § 2.2, can be further modified to solve this more general problem of minimizing a sum of two convex functions of the appropriate form. The necessary changes are as follows. We have two lists, one for F_1 and one for F_2, of total length $O(n)$. The calculation of $\delta(x)$ and its left and right gradients as required in Steps 0 and 4 is carried out by determining $F_j(x)$, λ_j, ρ_j $(j = 1, 2)$ as described in the original algorithm. Then $\delta(x) = F_1(x) + F_2(x)$, $\lambda = \lambda_1 + \lambda_2$, $\rho = \rho_1 + \rho_2$. The tests of Steps 0 and 4 are carried out using this λ and ρ. Otherwise all operations are for the two lists separately, with one exception. In forming the median in Step 2, the x_i values from both lists are pooled together. The median x^* is determined from this pooled set. The modified algorithm clearly still runs in $O(n)$ time. This completes the solution of the two-variable LP problem.

3. Three-variable LP. This section considers the three-variable LP problem:

$$\begin{aligned} &\text{maximize} && c_1x_1 + c_2x_2 + c_3x_3, \\ &\text{subject to} && a_{i1}x_1 + a_{i2}x_2 + a_{i3}x_3 \leqq b_i \qquad (i = 1, 2, \cdots, n), \end{aligned} \tag{1}$$

where the c_j, a_{ij}, b_i $(j = 1, 2, 3, i = 1, 2, \cdots, n)$ are arbitrary real numbers presented as data. The previous best known bound for this problem is again $O(n \log n)$, using the algorithm of Preparata and Muller [9] to determine all the $O(n)$ vertices of the constraint set, which is a convex polyhedron in R^3. Here an appropriate generalization of the method of § 2 to this three-variable problem is considered, and an $O(n)$ algorithm is developed. The section is divided in a similar way to § 2. Subsection 3.1 considers the relevant minimax problem. A component of the algorithm for this is the line-search, which is considered in § 3.2. Then § 3.3 first examines the LP problem when any point of the feasible region is known, and subsequently considers how such a point can be determined.

3.1. The minimax problem. Here R will denote the real line, R^2 the plane and R^3 space of three dimensions. Also, if u_i $(i = 1, 2, \cdots, p)$ are any real numbers, then med (u_i) will denote their median, i.e., the $\lceil \frac{1}{2}p \rceil$th in ranked order. As already remarked, it is well known [2] that med (u_i) can be determined in $\Theta(p)$ time. It is assumed throughout that such a linear-time algorithm is employed for median calculations.

Let $s = (x, y)$ be a typical point of R^2. Then a flat will mean any affine function of the form $(f_1x + f_2y + f_3)$ and a restraint any half-plane inequality of the form $h_1x + h_2y + h_3 \leqq 0$.

Now consider a function F defined as the maximum of n flats:

$$F(x, y) = \begin{cases} \max\limits_{1 \leqq i \leqq n} \{f_{i1}x + f_{i2}y + f_{i3}\} & \text{if } (x, y) \in D, \\ \infty & \text{otherwise,} \end{cases}$$

where D is the intersection of m restraints:

$$h_{j1}x + h_{j2}y + h_{j3} \leqq 0 \qquad (1 \leqq j \leqq m).$$

Now F is a polyhedral convex function with effective domain D [10].

Consider the problem of determining

$$F(x_0, y_0) = \min \{F(x, y) \colon (x, y) \in R^2\}.$$

We will construct an $O(m+n)$ time and space algorithm for determining $s_0=(x_0, y_0)$. The basic ideas behind the algorithm are as follows. Since F is convex, only the flats which are maximal, and the restraints which are binding, in an arbitrarily small (open) neighbourhood of s_0 are relevant in determining the location of s_0. Thus any flat or restraint not satisfying this condition can be discarded from the definition of F without disturbing s_0. This condition is checked directly for restraints in the algorithm, but for the flats the idea of dominance is employed. Consider any two flats $(f_1x+f_2y+f_3)$ and $(f'_1x+f'_2y+f'_3)$. If they are parallel (i.e., $f_1=f'_1, f_2=f'_2$) then the value of one is always greater than (or equal to) the other on R^2. (The one which attains max (f_3, f'_3) is the larger.) We say the larger dominates the smaller on R^2 and clearly any flat of F which is dominated by another on R^2 is irrelevant. If the flats are not parallel then they take the same value on the critical line

$$(f_1-f'_1)x+(f_2-f'_2)y+(f_3-f'_3)=0$$

in R^2. On one side of this line (i.e., in one of the two half-planes which it defines) one of the flats will be greater in value, and on the other side the other flat will be the larger. Again we say that one of the flats dominates the other on each half-plane. Thus, if s_0 is known to lie in one or other of these half-planes, then only the flat which is not dominated in that half-plane needs to be retained in order to locate s_0. These ideas are used to successively remove flats and restraints from the definition of F. In order to do this, *line-searches* of F are employed. Thus, given an arbitrary straight line L in R^2, we must determine whether $s_0\in L$, or if not, on which side of L it lies. In § 3.2 it is shown that any line-search of F, as defined above, can be carried out in $O(m+n)$ time. Here we presume this result. It should be observed that if F is unbounded below on any line, then $\min F=-\infty$ and s_0 is the relevant point at infinity on the line.

The algorithm to determine s_0 will now be described. It will be assumed that F and D are represented by linear lists, giving the coefficients of each flat or restraint. Various calculated numbers are associated with entries in these lists in an obvious fashion. Note that m and n, the lengths of these lists, are reduced as the algorithm proceeds.

Step 0. Determine any point $s_1\in D$, using the algorithm of § 2. If no such point exists, stop with s_0, chosen arbitrarily, and $F(s_0)=+\infty$. Otherwise, if $n=0$, stop with $s_0=s_1$, and $F(s_0)$ undefined.

Step 1. If $n=1$ then the problem reduces to a two-variable LP problem with a known feasible point s_1. Solve this using the algorithm of § 2 and stop. Otherwise, proceed through the F list, dividing the n flats into $\lfloor\frac{1}{2}n\rfloor$ pairs. For each pair perform the following operations. If the flats are parallel (see above), delete the one which is dominated on R^2. (If they are equal, delete either, but not both.) Otherwise, calculate the critical line for the pair. Thus at the end of this step we have $r\leqq\lfloor\frac{1}{2}n\rfloor$ lines and $d_1=(\lfloor\frac{1}{2}n\rfloor-r)$ flats have been deleted.

Step 2. Add the m lines, $h_{j1}x+h_{j2}y+h_{j3}=0$, corresponding to the restraints, to the r lines determined in Step 1. Thus we have $(m+r)$ lines, which can be expressed in one or other of the forms

$$y=\alpha_i x+\beta_i \ (i\in I_0) \quad \text{or} \quad x=u_i \ (i\in I_1).$$

Now calculate $\alpha^*=\text{med}\{\alpha_i: i\in I_0\}$, and then partition I_0 into three subsets I_2, I_3, I_4 corresponding to $\alpha_i=\alpha^*$, $\alpha_i<\alpha^*$ and $\alpha_i>\alpha^*$ respectively. Let $k_j=|I_j|$ $(j=1, 2, 3, 4)$, so we have

$$m+r=k_1+k_2+k_3+k_4$$

and $k_2+\min(k_3, k_4)\geqq\frac{1}{2}(m+r-k_1)$. (The latter inequality follows from the fact that α^* is the median.)

Now, as far as possible, pair the lines in I_3, I_4 so that each pair contains one line from each. Suppose these pairs are indexed by the set Q, say, where $|Q|=\min(k_3, k_4)$. For each pair $q\in Q$, determine the point of intersection of the lines in the pair. Thus if pair q contains the lines $y=\alpha_i x+\beta_i$ and $y=\alpha_j x+\beta_j$, this is the point (u_q, v_q) where

$$u_q=(\beta_i-\beta_j)/(\alpha_j-\alpha_i),$$

$$v_q=(\beta_i\alpha_j-\beta_j\alpha_i)/(\alpha_j-\alpha_i).$$

Note, by construction, $\alpha_j\neq\alpha_i$. Now add the u_q $(q\in Q)$ to the u_i $(i\in I_1)$ to give a set of $p=k_1+\min(k_3, k_4)$ u_j-values. Now calculate $x^*=\text{med}(u_j)$.

Step 3. Conduct a line-search along the line $x=x^*$. If s_0 is not located, then we will know either $x_0<x^*$ or $x_0>x^*$. By symmetry, we will consider only the case $x_0>x^*$. Now for all $i\in I_1$, with $u_i\leqq x^*$, we can delete either a flat from F or a restraint from D. Suppose d_3 such deletions are made. Then there are at least $(\frac{1}{2}p-d_3)$ of the u_q $(q\in Q)$ such that $u_q\leqq x^*$. Let these u_q be indexed by the set Q', say. Thus there are at least $(\frac{1}{2}p-d_3)$ pairs in Q' which do not intersect in the half-plane $x>x^*$. For each such pair, consider the line $y=\alpha^*(x-u_q)+v_q$ of median slope passing through their point of intersection. Add these to the lines of I_2 to give a set of at least $k_2+(\frac{1}{2}p-d_3)$ parallel lines of slope α^*. Determine the median line in this set. We do this by determining the y-intercept of each line. For lines from I_2 we have y-intercept β_i, and for pairs from Q' we have y-intercept $\gamma_q=v_q-\alpha^*u_q$. Then we calculate $y^*=\text{med}\{\beta_i, \gamma_q: i\in I_2, q\in Q'\}$. The required line is then $y=\alpha^*x+y^*$.

Step 4. Conduct a line-search along the line $y=\alpha^*x+y^*$. If s_0 is not determined, we will know either $y_0<\alpha^*x_0+y^*$ or $y_0>\alpha^*x_0+y^*$. Again, by symmetry, we will deal only with the case $y_0<\alpha^*x_0+y^*$. Now, for all lines $i\in I_2$ with $y_j=\beta_i\geqq y^*$, we can delete either a flat from F or a restraint from D, since we know $y_0<\alpha^*x_0+\beta_i=\alpha_i x_0+\beta_i$. Also, for all pairs in Q' with $y_j=v_q-\alpha^*u_q\geqq y^*$, the line in the pair with $\alpha_i>\alpha^*$ does not meet the region $\{x>x^*, y<\alpha^*x+y^*\}$ which is known to contain s_0. The situation is illustrated in Fig. 2. We have $x_0>u_q$, $\alpha_i>\alpha^*$, $v_q-\alpha^*u_q\geqq y^*$

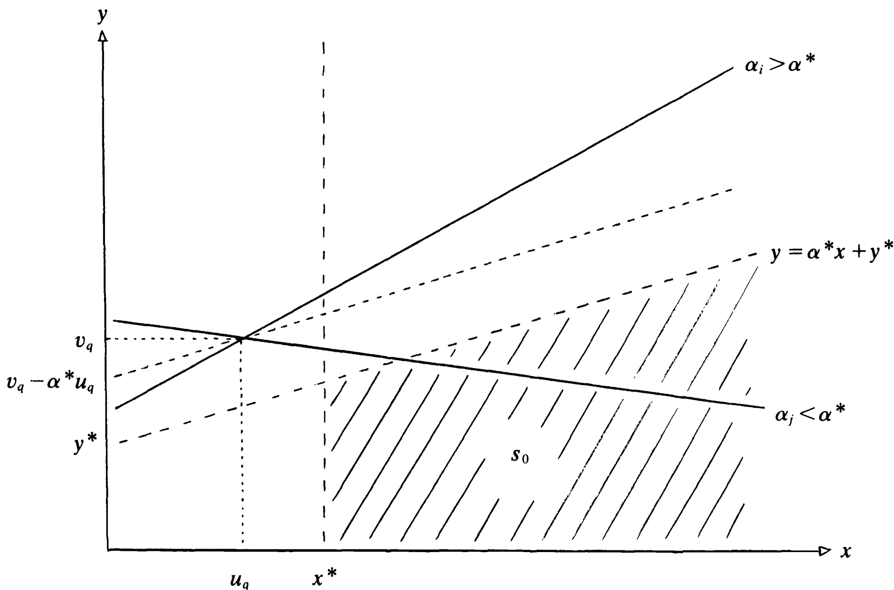


FIG. 2

and $y_0 < \alpha^* x_0 + y^*$. Hence $y_0 < \alpha^* x_0 + y^* \leqq \alpha^* x_0 + v_q - \alpha^* u_q = \alpha^*(x_0 - u_q) + v_q < \alpha_i(x_0 - u_q) + v_q = \alpha_i x_0 + (v_q - \alpha_i u_q) = \alpha_i x_0 + \beta_i$. Thus, for each pair in Q', either a flat from F or a restraint from D can be deleted. Suppose d_4 deletions have been made. Clearly $d_4 \geqq \frac{1}{2}(k_2 + \frac{1}{2}p - d_3)$. Now return to Step 1.

Call each repetition of Steps 1 to 4 an iteration of the method. We will determine how many flats or restraints are deleted in each iteration, assuming there are n and m respectively at the start. The total is clearly $d_1 + d_3 + d_4$, where

$$d_1 = \lfloor \tfrac{1}{2}n \rfloor - r, \; d_4 \geqq \tfrac{1}{2}(k_2 + \tfrac{1}{2}p - d_3).$$

Thus,

$$\begin{aligned} d_3 + d_4 &\geqq \tfrac{1}{2}(k_2 + \tfrac{1}{2}p) \geqq \tfrac{1}{4}(k_2 + p) \\ &= \tfrac{1}{4}(k_2 + k_1 + \min(k_3, k_4)) \\ &\geqq \tfrac{1}{4}(k_2 + \min(k_3, k_4) + \tfrac{1}{2}k_1). \end{aligned}$$

But $k_2 + \min(k_3, k_4) \geqq \frac{1}{2}(m + r - k_1)$ from above. Hence, $d_3 + d_4 \geqq \frac{1}{8}(m + r)$. Thus the total number of deletions

$$\begin{aligned} d_1 + d_3 + d_4 &\geqq \tfrac{1}{8}(m + r) + \lfloor \tfrac{1}{2}n \rfloor - r \\ &\geqq \tfrac{1}{8}(m + \lfloor \tfrac{1}{2}n \rfloor) \\ &\geqq \tfrac{1}{8}(m + n/3) \quad \text{for } n \geqq 2 \\ &\geqq \tfrac{1}{24}(m + n). \end{aligned}$$

Therefore at least $\frac{1}{24}$ of the flats and restraints are deleted in each iteration. Note also that at least one deletion must occur whenever $m > 0$ or $n > 1$. However, all cases with $n \leqq 1$ are trapped in Steps 0 or 1, and thus the algorithm must eventually terminate either in Step 1 or in one of the line-searches of Steps 3 and 4. The work for each line-search is $O(m + n)$ as is proved in § 3.2. All other work in each iteration is also $O(m + n)$ in any reasonable implementation. It follows from an argument similar to that used in § 2.1 that the whole algorithm is therefore $O(m + n)$ since a constant proportion of the data is discarded at each iteration. Note that we have used the weak bound $(m + n)/24$. In fact the lower bound will be nearer $(m/8 + n/16)$.

3.2. Line searching. Here we consider the subsidiary problem of § 3.1, that of determining whether F is minimized on a given line $L = \{s \in R^2 : \alpha_1 x + \alpha_2 y = \beta\}$. If not, we have to decide which side of L contains s_0. The first step is to note that we need only consider the case of minimizing on the line $y = 0$. Otherwise, we make the substitutions:

$$x' = \alpha_2 x - \alpha_1 y, \qquad y' = \alpha_1 x + \alpha_2 y - \beta$$

in each flat and restraint. This substitution can be carried out in $O(m + n)$ time and reduces L to the line $y' = 0$, and its two sides to the upper and lower half-planes. Its inverse is:

$$\begin{aligned} x &= (\alpha_2 x' + \alpha_1(y' + \beta))/(\alpha_1^2 + \alpha_2^2), \\ y &= (-\alpha_1 x' + \alpha_2(y' + \beta))/(\alpha_1^2 + \alpha_2^2). \end{aligned}$$

Henceforth, in this section, we will drop the primes and consider only minimizing F, subject to its restraints, along the line $y=0$. This gives rise to the problem:

$$\text{minimize} \quad G(x)=\max_{1\leqq i\leqq n}\{f_{i1}x+f_{i3}\},$$

$$\text{subject to} \quad h_{j1}x+h_{j3}\leqq 0 \qquad (1\leqq j\leqq m).$$

Each line in G corresponds to a flat of F, and each inequality to a restraint of D. An $O(m+n)$ algorithm for the solution of this problem has been described above in § 2. Here we will consider only the possible results of applying that algorithm. Suppose G takes its minimum value at the point x^0. Then if $G(x^0)=+\infty$, we choose the side of L containing the point s_1 determined in Step 0 of the algorithm of § 3.1. By convexity, this side contains all points satisfying the restraints. If $G(x^0)=-\infty$ (whence $x^0=\pm\infty$), then $F(x^0,0)=-\infty$ and the algorithm of § 3.1 should terminate with the indication that F is unbounded below in the appropriate direction on L. Otherwise, $G(x^0)$ is finite. Now determine all restraints $j\in J$, say, which are binding at $(x^0,0)$, i.e., $j\in J$ if

$$h_{j1}x^0+h_{j3}=0.$$

Similarly, determine all flats $i\in I$ which are maximal at $(x^0,0)$, i.e., $i\in I$ if

$$f_{i1}x^0+f_{i3}=G(x^0)=F(x^0,0).$$

Now consider the restricted problem:

$$\text{minimize} \quad F'(x,y)=\max_{i\in I}\{f_{i1}x+f_{i2}y+f_{i3}\},$$

$$\text{subject to} \quad h_{j1}x+h_{j2}y+h_{j3}\leqq 0 \qquad (j\in J).$$

This problem is identical to the minimization of F over D in any small enough neighborhood of $(x^0,0)$. Thus, by convexity, if F' is minimized at this point, then so is F. Similarly, the side of $y=0$ which contains (all) points $(x,y)\in D$ with $F(x,y)<F(x^0,0)$ is the same side which contains points satisfying the restraints of J with $F'(x,y)<F'(x^0,0)$. Now, using the defining relationships, the restricted problem may be rewritten:

$$\text{minimize} \quad F^*(x,y)=\max_{i\in I}\{f_{i1}(x-x^0)+f_{i2}y\},$$

$$\text{subject to} \quad h_{j1}(x-x^0)+h_{j2}y\leqq 0 \qquad (j\in J),$$

where $F^*(x,y)=F'(x,y)-F'(x^0,0)$. Therefore, we must determine whether F^* is minimized at $(x^0,0)$ or else which of the lower or upper half-planes contains points with $F^*(x,y)<0$. Since F^* is certainly minimized at this point on the line $y=0$, we need only consider points at which $y\neq 0$. Let $X=(x-x^0)/|y|$ and $G^*(X)=F^*(x,y)/|y|$. Also write $\sigma=\operatorname{sgn}(y)$. Then the restricted problem may be further rewritten:

$$\text{minimize} \quad G^*(X)=\max_{i\in I}\{f_{i1}X+f_{i2}\sigma\},$$

$$\text{subject to} \quad h_{j1}X+h_{j2}\sigma\leqq 0 \qquad (j\in J).$$

Now σ can take only the values $+1$, -1. Also $G^*(X)$ can take negative values if and only if $F^*(x,y)$ can do so. Thus, in at most two applications of the algorithm of § 2, we can decide whether F is minimized at $(x^0,0)$. Otherwise there will be exactly one

value of $\sigma = \operatorname{sgn}(y)$ with negative G^*. This determines the half-plane, $y>0$ or $y<0$, in which values of F smaller than $F(x^0, 0)$ may be found. The whole process can clearly be carried out in $O(m+n)$ time. In fact, it is possible to determine the appropriate value of σ in a slightly easier way. Since $F^*(x, 0)$ is minimized at $(x^0, 0)$, it follows that there must exist either:

(a) $p, q \in I$ with $f_{p1}f_{q1} \leqq 0$ (it is possible that $p=q$),
(b) $p \in I, q \in J$ with $f_{p1}h_{q1} \leqq 0$.

Because we require $G^*(X)<0$, we can unite these two cases by noting that we have two inequalities of the form,

(2) $$g_1x+g_2\sigma \leqq 0 \quad \text{with } g_1 \geqq 0, \qquad g_1'X+g_2'\sigma \leqq 0 \quad \text{with } g_1' \leqq 0$$

and strict inequality in at least one of these. If $g_1=0$ then $g_2\sigma<0$, so $\sigma = \operatorname{sgn}(-g_2)$; similarly, if $g_1'=0$. Otherwise, eliminating X we have

$$(g_1'g_2-g_1g_2')\sigma>0, \quad \text{i.e.} \quad \sigma = \operatorname{sgn}(g_1'g_2-g_1g_2').$$

These can be checked quickly. Observe that if $(g_1'g_2-g_1g_2')=0$, so the sign is undefined, then it follows that F is minimized at $(x^0, 0)$. Note also that p, q can be determined by a single pass through the data-structure for G^*. (In fact, the algorithm of § 2 gives these as a by-product without computation.) This completes the description of the line-search. It is clear that, in any reasonable implementation utilizing the algorithm of § 2 where appropriate, the search can be done in $O(m+n)$ time.

A slightly more general line-search is required in § 3.3, in which we are minimizing the sum δ of two minimax functions F_1, F_2 (of the form of F above) over the region D. The algorithm for the one-dimensional form of this problem was also outlined in § 2. The above discussion requires little modification. The restricted problem is again equivalent to at most two one-dimensional problems of the same form. The above quick method for determining σ requires a minor change. We will have sets I_1, I_2, J defined similarly to the above. Using a superscript k to denote the flats belonging to $F_k (k=1,2)$, then J is defined exactly as before and $i \in I_k$ if

$$f_{i1}^k x^0 + f_{13}^k = F_k(x^0, 0),$$

where x^0 is the point at which δ is minimized on $y=0$. Then the cases (a) and (b) preceding inequalities (2) need to be changed to the following. There exist either

(a) $p, q \in I_1$, and $t, u \in I_2$ with $(f_{p1}^1+f_{t1}^2)(f_{q1}^1+f_{u1}^2) \leqq 0$ (again $p=q$ or $t=u$ is a possibility),

or

(b) $p \in I_1, q \in I_2$ and $t \in J$ with $(f_{p1}^1+f_{q1}^2)h_{t1} \leqq 0$.

Now, if in case (a), we define $g_i = f_{pi}^1+f_{ti}^2$ and $g_i' = f_{qi}^1+f_{ui}^2$ $(i=1,2)$ or vice versa depending on signs, then the inequalities (2) and the subsequent analysis carry through unchanged. This follows from the facts that, in analogous notation to the above, we must have

$$\begin{aligned} 0 &\geqq G_1^*(X)+G_2^*(X) \\ &\geqq \max(f_{p1}^1X+f_{p2}^1\sigma, f_{q1}^1X+f_{q2}^1\sigma)+\max(f_{t1}^2X+f_{t2}^2\sigma, f_{u1}^2X+f_{u2}^2\sigma) \\ &\geqq \max((f_{p1}^1+f_{t1}^2)X+(f_{p2}^1+f_{t2}^2)\sigma, (f_{q1}^1+f_{u1}^2)X+(f_{q2}^1+f_{u2}^2)\sigma). \end{aligned}$$

Case (b) is dealt with similarly. Thus, the modifications needed for line-searching in this more general problem are relatively straightforward. The remaining details can be easily worked out by direct analogy with the previous case, and therefore will not be considered further. Again the procedure runs in $O(m+n)$ time, where n is now the total number of flats in F_1 and F_2 combined.

3.3. The LP problem. The solution of the LP (1) can now be obtained by reducing it to that of the previous sections. The reduction process is very similar to that given, in more detail, in § 2.2. First we consider the problem when a point (x'_1, x'_2, x'_3) of the feasible region to (1) is given. If $c_1 = c_2 = c_3 = 0$, then clearly the point is optimal. Thus we will assume $c_3 \neq 0$, possibly after a simple permutation of the variables. (The corresponding data permutation is evidently linear time.) Then we make the substitution $X_j = (x_j - x'_j)(j = 1, 2, 3)$ to bring the problem to the form:

$$\begin{aligned}
&\text{maximize} && c_0 + c_1 X + c_2 X_2 + c_3 X_3 \\
&\text{subject to} && a_{i1} X_1 + a_{i2} X_2 + a_{i3} X_3 \leqq B_i \qquad (i = 1, 2, \cdots, n), \\
&\text{where} && c_0 = c_1 x'_1 + c_2 x'_2 + c_3 x'_3 \text{ and } B_i = b_i - a_{i1} x'_1 + a_{i2} x'_2 + a_{i3} x'_3 \geqq 0.
\end{aligned}$$

The constant c_0 can obviously be ignored. Now we make a further linear transformation $X = X_1$, $Y = X_2$, $Z = c_1 X_1 + c_2 X_2 + c_3 X_3$. (Note that this is nonsingular.) The problem will now be:

$$\begin{aligned}
&\text{maximize} && Z \\
&\text{subject to} && A_{i1} X + A_{i2} Y + A_{i3} Z \leqq B_i, \\
&\text{where} && A_{ij} = a_{ij} - c_j a_{i3}/c_3 \ (j = 1, 2), A_{i3} = a_{i3}/c_3.
\end{aligned}$$

Now, since the feasible point $X = Y = Z = 0$ attains $Z = 0$, there is no loss in assuming $Z > 0$. If this assumption leads to an infeasible problem, the zero solution is obviously optimal. Thus we make the further projective transformation $x = X/Z$, $y = Y/Z$, $z = 1/Z$. The problem now has the form:

$$\begin{aligned}
&\text{minimize} && z, \\
&\text{subject to} && A_{i1} x + A_{i2} y + A_{i3} \leqq B_i z.
\end{aligned}$$

Now, let I_0 be the set of i such that $B_i = 0$, and I_1 the set for which $B_i > 0$. Define $h_{ij} = A_{ij}$ $(i \in I_0)$ and $f_{ij} = A_{ij}/B_i$ $(i \in I_1)$ for $j = 1, 2, 3$. The problem now becomes:

$$\begin{aligned}
&\text{minimize} && \max_{i \in I_1} (f_{i1} x + f_{i2} y + f_{i3}), \\
&\text{subject to} && h_{i1} x + h_{i2} y + h_{i3} \leqq 0 \qquad (i \in I_0),
\end{aligned}$$

which is the problem of § 2. Since the data transformations are all linear-time, and the resulting problem can be solved in linear-time, we have a linear-time algorithm when a feasible point is known.

We now turn to the problem of determining a feasible point, or showing that none exists. We consider only the inequalities of (1):

$$a_{i1} x_1 + a_{i2} x_2 + a_{i3} x_3 \leqq b_i \qquad (i = 1, 2, \cdots, n).$$

First we partition these inequalities on the value of a_{i3}. Let $i \in I^+$ if $a_{i3} > 0$, $i \in I^0$ if $a_{i3} = 0$ and $i \in I^-$ if $a_{i3} < 0$. (The partition takes linear time.) For constraints in I^+, I^-

we divide through by $|a_{i3}|$ and rearrange the inequalities to give

$$x_3 \geqq \alpha_{i1}x_1 + \alpha_{i2}x_2 + \alpha_{i3} \qquad (i \in I^-),$$

$$-x_3 \geqq \alpha_{i1}x_1 + \alpha_{i2}x_2 + \alpha_{i3} \qquad (i \in I^+),$$

$$a_{i1}x_1 + a_{i2}x_2 \leqq b_i \qquad (i \in I^0),$$

where

$$\alpha_{ij} = a_{ij}/|a_{i3}| \qquad (j = 1, 2),$$

$$\alpha_{i3} = -b_i/|a_{i3}|.$$

Now, writing

$$F_1(x_1, x_2) = \max_{i \in I^-} (\alpha_{i1}x_1 + \alpha_{i2}x_2 + \alpha_{i3}), \qquad F_2(x_1, x_2) = \max_{i \in I^+} (\alpha_{i1}x_1 + \alpha_{i2}x_2 + \alpha_{i3}),$$

the problem can be expressed as

$$F_1(x_1, x_2) \leqq x_3 \leqq -F_2(x_1, x_2) \quad \text{with } a_{i1}x_1 + a_{i2}x_2 - b_i \leqq 0 \quad (i \in I^0).$$

A suitable x_3 exists if there are x_1, x_2 satisfying the restraints of I^0 for which $F_1(x_1, x_2) \leqq -F_2(x_1, x_2)$. Letting $\delta(x_1, x_2) = F_1(x_1, x_2) + F_2(x_1, x_2)$, this is precisely when $\delta(x_1, x_2) \leqq 0$. Thus the problem of determining a feasible point, or demonstrating that none exists, is equivalent to minimizing the convex function $\delta = F_1 + F_2$ subject to the restraints of I^0. If the minimum value of δ is positive, then the feasible region is empty, otherwise a point may be readily calculated. This problem is a slight generalization of that of § 3.1. The line-searches in this case were discussed at the end of § 3.2. The algorithm of § 3.1 then requires also the following modifications. Separate lists for F_1 and F_2 must be maintained. Also, the initial pairing in Step 1 must be done on the two lists separately. Otherwise the algorithm requires only trivial change. This completes the description of an $O(n)$ algorithm for three-variable LP.

4. Comments. We have outlined $O(n)$ time and space algorithms for two- and three-variable LPs. This obviously settles, for most purposes, the LP problem in R^d for $d \leqq 3$. Thus the first interesting case of linear programming occurs in R^4. (This is also the simplest case which is still, apparently, open for enumerating all the vertices of the feasible region. See [3].) A bound of $O(n^2)$ for LP in R^4 follows directly from the above. This results from solving (in $O(n)$ time) the n LPs in R^3 which result from taking each inequality as an equality in turn, and choosing the best. However, this bound is far from optimal. It may also be observed that the results given here imply, in a fairly straightforward manner, a linear time solution for the problem of determining a point in the intersection of two polyhedra in R^3 [7]. The author has previously described a somewhat different solution for that problem in [4]. Moreover, since with the above methods we can determine a point in the intersection of half-spaces in R^3 in linear time, the results given here imply the equivalence in linear time of the problems of determining the intersection of half-spaces [9] with that of determining the convex hull [8] in R^3. This settles one of the open problems discussed in [3]. In consequence, the polyhedra-intersection problem seems of less central importance than previously believed. Another question concerns the practicality of the algorithms described here. Certainly the algorithm for the two-variable problem appears simple enough to be of practical, as well as theoretical, interest. However no evaluation of these methods has yet been attempted in comparison with, say, the simplex method. In this respect it may be noted that, while practical LPs generally have large numbers

of variables, efficient algorithms for solving problems with only a few variables could form an effective adjunct to the simplex method itself, by allowing multiple, rather than single, column selection at each iteration. Finally, it may be noted that the methods employed in the algorithms given here generalize to certain other problems. See, for example, [5].

REFERENCES

[1] J. L. BENTLEY AND M. I. SHAMOS, *Divide and conquer for linear expected time*, Inform. Process. Lett., 7 (1978), pp. 87–91.

[2] M. BLUM, R. W. FLOYD, V. R. PRATT, R. L. RIVEST AND R. E. TARJAN, *Time bounds for selection*, J. Comput. System. Sci., 7 (1972), pp. 448–461.

[3] M. E. DYER, *The complexity of vertex enumeration methods*, Math. Oper. Res., to appear.

[4] ———, *An $O(n)$ algorithm for locating a point in the intersection of polyhedra*, Math. Dept., Teesside Polytechnic, 1981.

[5] ———, *An $O(n)$ algorithm for the multiple-choice knapsack linear program*, Maths. Dept., Teesside Polytechnic, Middlesbrough, England, 1981.

[6] L. G. KHACHIYAN, *Polynomial algorithm for linear programming*, Dokl. Akad. Nauk SSSR (1979), pp. 1093–1096. (In Russian.)

[7] D. E. MULLER AND F. P. PREPARATA, *Finding the intersection of two convex polyhedra*, Theoret. Comput. Sci., 7 (1978), pp. 217–236.

[8] F. P. PREPARATA AND S. J. HONG, *Convex hulls of finite sets in two and three dimensions*, Comm. ACM, 20 (1977), pp. 87–93.

[9] F. P. PREPARATA AND D. E. MULLER, *Finding the intersection of n half-spaces in time $O(n \log n)$*, Theoret. Comput. Sci., 8 (1979), pp. 45–55.

[10] R. T. ROCKAFELLAR, *Convex Analysis*, Princeton Univ. Press, Princeton, NJ, 1970.

[11] M. I. SHAMOS AND D. HOEY, *Geometric intersection problems*, Proc. Seventeenth Annual IEEE Symposium on Foundations of Computer Science, October 1976, pp. 208–215.

[12] R. SHOSTAK, *Deciding linear inequalities by computing loop residues*, J. Assoc. Comput. Mach., 28 (1981), pp. 769–779.

[13] A. C. C. YAO, *A lower bound to finding convex hulls*, J. Assoc. Comput. Mach., 28 (1981), pp. 780–787.

SIAM J. COMPUT.
Vol. 13, No. 1, February 1984
© 1984 Society for Industrial and Applied Mathematics
0097-5397/84/1301-0004 $01.25/0

ON SYNCHRONOUS PARALLEL COMPUTATIONS WITH INDEPENDENT PROBABILISTIC CHOICE*

JOHN H. REIF†

Abstract. This paper introduces probabilistic choice to synchronous parallel machine models; in particular parallel RAMs. The power of probabilistic choice in parallel computations is illustrated by parallelizing some known probabilistic sequential algorithms. We characterize the computational complexity of time, space, and processor bounded probabilistic parallel RAMs in terms of the computational complexity of probabilistic sequential RAMs. We show that parallelism uniformly speeds up time bounded probabilistic sequential RAM computations by nearly a quadratic factor. We also show that probabilistic choice can be eliminated from parallel computations by introducing nonuniformity.

Key words. parallel computation, probabilistic computation, randomized algorithm, parallel speedup, parallel RAM

1. Introduction. *Probabilistic choice* is the use of independent randomly chosen moves in an otherwise deterministic computation given a fixed input. The introduction of probabilistic choice in *sequential computations* [see the landmark paper of Rabin (1974)] leads to considerable improvement to the computational complexity of various number theoretic problems [Berlekamp (1979), Rabin (1974), Solovay and Strasser (1977), Adleman, Manders and Miller (1979), Rabin (1980), Zippel (1979)] to combinatorial problems on graphs and matroids [Lovász (1980)] to testing polynomial identities [Schwartz (1980)], and testing program equivalence [Ibarra and Moran (1980)].

Recently, Rabin (1980), Lehman and Rabin (1980), Francez and Rodeh (1980), Reif and Spirakis (1981), (1982a, b) have utilized probabilistic choice in *synchronization algorithms* for asynchronous multiprocesses systems. This paper investigates the use of probabilistic choice in *synchronous parallel machines.* Section 2 provides the relevant definitions.

In § 3 we provide some concrete examples of probabilistic parallel algorithms for combinatorial problems. We give an $O(\log n)^2$ time algorithm for testing if a graph of n vertices has a perfect matching. We give an $O(\log n)$ time, $n^2/\log n$ processor algorithm for testing the product of $n \times n$ matrices. (Note. These algorithms are simply derived by parallelizing known sequential algorithms, and are merely examples of the power of probabilistic choice in parallel computations. Later sections of our paper give deeper and more substantive theoretical results.)

Probabilistic choice has also recently been used in parallel algorithms for routing [Valiant and Brebner (1981)] and sorting. Reischuk (1981) has shown that a probabilistic RAM can sort in time $O(\log n)$ with n processors, and Reif and Valiant (1982) give an $O(\log n)$ time algorithm for sorting in constant valence, fixed connection networks with n processors.

We present in § 4 a pair of simulation results (Theorems 4.1 and 4.2) which relate probabilistic sequential and probabilistic parallel computations on RAMs. By parallel simulation of previously known probabilistic sequential algorithms [Aleliunas et al.

* Received by the editors November 20, 1981, and in revised form November 25, 1982. This work was supported in part by National Science Foundation grant NSF-MCS82-00269 and Office of Naval Research contract N00014-80-C-0674.

† Aiken Computation Laboratory, Division of Applied Science, Harvard University, Cambridge, Massachusetts 02138.

(1979)], our Theorem 4.1 immediately yields as corollaries $O(\log n)$ time probabilistic parallel algorithms for a variety of combinatorial problems such as testing if there exists a path between two vertices of an undirected graph and testing if graph is bipartite. Both these probabilistic parallel algorithms use a polynomial number of processors. Reif (1982a) describes related $O(\log n)$ time, polynomial processor probabilistic parallel algorithms for constructing minimum spanning forests, k-connectivity, k-connected components, and recognizing chordal graphs, comparability graphs, interval graphs, split graphs, permutation graphs, and constant valence planar graphs. Previously the fastest known parallel algorithms for any of these problems required $\Omega(\log^2 n)$ time.

We have an interesting theoretical result (Theorem 5) for speeding up a unit-cost, probabilistic sequential RAM computation of time $T(n)$, by simulation on a probabilistic parallel RAM in unit-cost time $O(T(n)(\log T(n)) \log (T(n)I(n)))^{1/2}$, where $I(n)$ is the maximum integer operated upon the simulated unit-cost probabilistic RAM. Our simulation result also holds for deterministic computations. Previously, Dymond (1980) showed a quadratic speedup of deterministic multitape Turing machines. However, no such speedup has been previously proved for deterministic RAMS.

Theorem 6 of § 6 proves that probabilistic choice can be eliminated from probabilistic parallel RAMs with both errors of acceptance and errors of rejection by introducing nonuniformity, with some increase of time and processor bounds which may be traded off. For example, this implies there exists nonuniform deterministic parallel RAMs which can in unit-cost time $O(\log n)$ test if a graph of n vertices is connected, and in time $O(\log n)^2$ test if a graph of n vertices has a perfect matching. Previously, Adleman (1978) and Bennett and Gill (1981) had shown that probabilistic choice can be eliminated in probabilistic sequential computations with bounded error.

At the end of this paper we provide an extended list of references to literature on probabilistic and parallel algorithms which may aid further research in this area.

2. Definitions of probabilistic machines.

2.1. Abstract machine types. Before describing our probabilistic parallel machines, it is useful to define probabilistic (and also deterministic and nondeterministic) machine types abstractly, without reference to the particular details of operation of the machines.

Let M be a fixed machine. A *configuration* of M is a finite string I over a fixed finite alphabet describing the current state and storage contents of M. Let $\mathcal{I}$ be the set of configurations of M. Let $\mathcal{I}_A \subseteq \mathcal{I}$ be the set of *accepting configurations* of M. Let Σ be the finite input alphabet of M. Given an input string $\omega \in \Sigma^*$, let $I_0(\omega) \in \mathcal{I}$ be the corresponding *initial configuration* of M. Let $\vdash \subseteq \mathcal{I} \times \mathcal{I}$ be the *next move relation* for M; for each $I \in \mathcal{I}$, $\text{NEXT}(I) = \{I' | I \vdash I'\}$ is the set of possible configurations derived from I by a single move of M. (We assume there is no next move from an accepting configuration.) In a *nondeterministic* machine, any $I' \in \text{NEXT}(I)$ may be chosen nondeterministically. In a *probabilistic machine*, each $I' \in \text{NEXT}(I)$ is chosen with equal probability, independently of previous and succeeding choices. In a *deterministic machine M*, $|\text{NEXT}(I)| \leqq 1$ for all $I \in \mathcal{I}$.

Given a fixed input string $\omega \in \Sigma^*$, a *computation sequence* of M is a maximal length sequence of configurations $I_0, I_1, \cdots$ such that $I_0 = I_0(\omega)$ and $I_{i-1} \vdash I_i$ for $i = 1, 2, \cdots$. The computation sequence is *accepting* if it is finite and the last configuration is accepting. In a deterministic or nondeterministic machine, M accepts ω iff there exists an accepting computation sequence from $I_0(\omega)$. In a probabilistic machine, *M accepts ω* iff Prob (COMP (ω) is accepting) $>\frac{1}{2}$, where COMP (ω) is a random

computation sequence from $I_0(\omega)$ (generated by random next moves as defined above). Let the *language accepted* by M be $L(M)=\{\omega\in\Sigma^*|M \text{ accepts } \omega\}$.

2.2. Error restricted probabilistic machines. Let M be a probabilistic machine which accepts language $L(M)$. Let the *acceptance error* $\varepsilon_A(n)$ and the *rejection error* $\varepsilon_R(n)$ be the minimum functions such that for all $n\geqq 0$, $\omega\in\Sigma^n$,

(i) if $\omega\notin L(M)$ then Prob {COMP (ω) is accepting} $\leqq \varepsilon_A(n)$;

(ii) if $\omega\in L(M)$ then Prob {COMP (ω) is rejecting} $\leqq \varepsilon_R(n)$.

Note that by definition $\varepsilon_A(n)\leqq\frac{1}{2}$ and $\varepsilon_R(n)\leqq\frac{1}{2}$.

For deterministic or nondeterministic machines M, M' let $M\approx M'$ if $L(M)=L(M')$. For two probabilistic machines M, M', let $M\approx M'$ if $L(M)=L(M')$ and both M and M' have the same error of acceptance and the same error of rejection.

Let M be a BP-*probabilistic machine* if there exists a constant $\varepsilon<\frac{1}{2}$ such that for all $n\geqq 0$, $\varepsilon\geqq\max(\varepsilon_A(n),\varepsilon_R(n))$. Thus a BP-probabilistic machine has a constant upper bound, which is less than $\frac{1}{2}$, on errors of acceptance and rejection.

Let M be a *R-probabilistic machine* if there exists a constant $\varepsilon<\frac{1}{2}$ such that for all $n>0$, $\varepsilon\geqq\varepsilon_R(n)$, and M never has an accepting computation on any input string $\omega\in\Sigma^*-L(M)$.

2.3. Probabilistic sequential machines. A nondeterministic Turing machine may be made a *probabilistic Turing machine* by allowing next moves to be chosen randomly with equal probability, as described in § 2.1. See Simon (1975) for a discussion of probabilistic Turing machines with unrestricted errors and see Adleman (1978) for some results for R-probabilistic Turing machines. Bennett and Gill (1981) discuss these and various other classes of probabilistic Turing machines.

Our principal sequential machine model is the probabilistic Random Access Machine (RAM), which is defined here similarly to Aho, Hopcroft and Ullman (1974), except that we allow the RAM probabilistic choice. A *probabilistic* RAM consists of

(1) an infinite sequence of memory locations $m_0, m_1, \cdots$ each of which are indexed by and contain a nonnegative integer;

(2) a fixed set of registers R each of which contains a nonnegative integer;

(3) a probabilistic finite state control which allows the following operations:

  (a) for any registers $r_1, r_2\in R$, *store* (or *read*) the contents of r_1 into (or from, respectively) the contents of global memory location m_i, where i is the current contents of register r_2;

  (b) for any registers $r_1, r_2, r_3\in R$, apply an addition, subtraction, multiplication, or division operation on the contents of registers r_1, r_2 and load the result into register r_3.

(Note. We round noninteger rationals to the next lower integer. Also, we substitute 0 for the result of subtraction which is negative.)

A *unit cost* RAM is charged 1 step for each of the above operations; a *log-cost* RAM is charged $\lceil\log(x+2)\rceil$ steps for each of the above operations which are on integers of size x.

We assume a binary input alphabet {0, 1}. Given an input string $\omega\in\{0,1\}^*$, each memory location m_{i-1} initially contains the ith bit of ω for $1\leqq i\leqq|\omega|$, m_n contains 2, and all other memory locations and registers are initially 0. The memory locations $m_0,\cdots,m_n$ are read-only, and cannot be stored into. Also, we assume the finite control has distinguished *initial* and *accepting* states. A configuration is accepting if the machine is in the accepting state. The probabilistic RAM *accepts* input ω if with probability $>\frac{1}{2}$ a random computation sequence is accepting. The probabilistic RAM has *time bound* $T(n)$ (*space bound* $S(n)$, *integer bound* $I(n)$) if on all inputs of length

n and accepting computation sequences, the machine takes $\leqq T(n)$ steps (uses $\leqq S(n)$ space, operates on integers $\leqq I(n)$, respectively). Note that we have defined steps differently for unit-cost and log-cost RAMs. Furthermore, a log-cost RAM (unit-cost RAM, respectively) is charged $\log(x+2)$ (1, respectively) unit of space for each noninput memory location and register utilized in an accepting computation, where x is the largest integer stored in that memory location or register.

2.4. Probabilistic parallel RAMs. Our principal parallel machine model is the Parallel Random Access Machine (P-RAM), similar to that defined in Fortune and Wyllie (1978) and Wyllie (1979). However, we allow these machines probabilistic choice. Initially, given an input string $\omega \in \{0, 1\}^*$, a probabilistic P-RAM consists of a single probabilistic RAM initialized as defined in § 2.3, with an additional operation: *fork* which allows the original RAM to create a new "clone" RAM sharing the same memory, with copies of the original RAM's registers with the same contents, with an identical finite state control, and initialized at some given state. Any new RAMs may also create new RAMs by the fork operations. All these RAMs operate synchronously with the original RAM. Furthermore, their probabilistic choices are assumed to be independent. RAMs are allowed to simultaneously read the same memory location. However, if two distinct RAMs simultaneously store into the same memory contents, then the entire computation of the P-RAM fails and the P-RAM rejects the input. If on a particular computation sequence the original RAM enters its accept state and there have been no such simultaneous memory store conflicts then this computation sequence is considered to be accepting. The probabilistic P-RAM accepts an input string $\omega \in \{0, 1\}^*$ if with probability $>\frac{1}{2}$ a random computation sequence is accepting. (See § 2.2 for definitions of errors of acceptance and rejection.) The probabilistic P-RAM has *time bound* $T(n)$ (*space bound* $S(n)$, *integer bound* $I(n)$, *processor bound* $P(n)$) if on all inputs of length n and accepting computation sequences, the machine taken $\leqq T(n)$ steps, (uses $\leqq S(n)$ space, operates on integers $\leqq I(n)$, uses $\leqq P(n)$ processors, respectively). Note that space and time are charged in units depending on whether the machine is unit-cost or log-cost as defined in § 2.3.

3. Some fast probabilistic parallel algorithms. This section describes some time efficient algorithms for probabilistic P-RAMs which we easily derive by parallelizing known probabilistic sequential algorithms. (For more substantive theoretical results the reader should read later sections; for example, § 4 gives a uniform method for parallelizing any probabilistic sequential algorithm.) All the algorithms described here can be made *R-probabilistic* (with rejection error $<\frac{1}{2}$ and no errors of acceptance) if the probabilistic trials are made twice.

THEOREM 3.1. *There are unit-cost R-probabilistic* P-RAMs *with time bound* $O(\log n)$ *and polynomial processor bound, which given a graph G with n vertices,*

(a) *can test if G has a path between two given vertices, and*

(b) *can test if G is bipartite.*

Proof. Aleliunas *et al.* (1979) give for these problems R-probabilistic sequential algorithms which can be implemented on a unit-cost R-probabilistic RAM in unit space (using integers size $\leqq n^2$ for representing edges) and $O(n^3)$ time. Our probabilistic parallel algorithms are derived immediately by applying Theorem 4.1. □

Note that the fastest known deterministic P-RAM algorithm for testing undirected connectivity requires $\Omega(\log n)^2$ time [Hirschberg, Chandra and Sarwate (1978)].

THEOREM 3.2. *A unit-cost R-probabilistic* P-RAM *with time bound* $O(\log n)^2$ *and processor bound* $O(n^{3.31})$ *can test if a graph of n vertices has a perfect matching.*

Proof. Let $G=(V,E)$ be an undirected graph with vertices $V=\{1,\cdots,n\}$. Lovász (1980) gives a probabilistic sequential algorithm which chooses an $N=n^{O(1)}$ and constructs a symmetric $n\times n$ matrix B where for $1\leqq i,j\leqq n$

(a) B_{ij} is a random element of $\{1,\cdots,N\}$ if $i<j$ and $(i,j)\in E$,

(b) $B_{ij}=-B_{ji}$ if $i>j$ and $(i,j)\in E$,

(c) $B_{ij}=0$ otherwise.

G has a perfect matching if the determinant of any such B is not 0. If G has no perfect matching, then for n sufficiently large the determinant of B is 0 with probability $\geqq\frac{1}{2}$. The parallel matrix inversion algorithm of Csanky (1976) as improved by Preparata and Sarwate (1978) can be used to compute the determinant in time $O(\log n)^2$ and $O(n^{3.31})$ processors on a P-RAM.

THEOREM 3.3. *A unit-cost R-probabilistic* P-RAM *with time bound $O(\log n)$ and processor bound $O(n^2/\log n)$ given $n\times n$ integer matrices A, B, C can test $A\cdot B\neq C$.*

Proof. Choose a random column vector $x\in\{-1,1\}^n$ and test $A(Bx)\neq Cx$. This test can be done by a probabilistic P-RAM within time $O(\log n)$ and processor bound $O(n^2/\log n)$ by forming $n/\log n$ binary trees of processors, each of size $2n$ and depth $O(\log n)$, and pipelining the required dot products. Freivalds (1979) shows that if $A\cdot B\neq C$ then Prob $\{A(Bx)=Cx\}<\frac{1}{2}$.

Note that the naive pipelining algorithm for testing $A\cdot B\neq C$ in time $O(\log n)$ on a deterministic P-RAM requires $\Omega(n^3/\log n)$ processors, and requires time $\Omega(n\log n)$ given only $n^2/\log n$ processors.

4. Simulation results between probabilistic RAMs and probabilistic P-RAMs. Fortune and Wyllie (1978) and Wyllie (1979) characterize the computational complexity of their deterministic P-RAMs in terms of the complexity of deterministic complexity classes. It is the aim of this section to do the same for our probabilistic P-RAMs. Our simulation methods are similar, except for the use of probabilistic choice to insure that the probability of errors of acceptance and rejection are preserved.

4.1. Simulation of a probabilistic RAM by a probabilistic P-RAM.

THEOREM 4.1. *Let M be a probabilistic* RAM *with constructible time bound $T(n)\geqq n$, space bound $S(n)\geqq\log n$, and integer bound $I(n)$. Then there is a probabilistic* P-RAM *M' such that $M\approx M'$ (see* § 2.2 *for definition of the equivalence relation $\approx$ and note that if M is deterministic, then M' is also deterministic*); *if M is unit-cost then M' has unit-cost time bound $O(S(n)\log I(n)+\log T(n))$, and processor bound $O(I(n)^{S(n)}T(n))$; if M is log-cost then M' has log-cost time bound $O(S(n)+\log T(n))^2$ and processor bound $O(4^{S(n)}T(n))$.*

(Note. Theorem 4.1 gives a speed-up for unit-cost RAMs only if $S(n)\log I(n)<T(n)$; Theorem 5.1 provides a uniform quadratic speed-up even if $S(n)=T(n)$.)

Proof. Fix some input string $\omega\in\Sigma^n$ and let $I_0(\omega)$ be the initial configuration of M. Let $\mathcal{I}$ be the set of configurations of M with space $S(n)$. Let $p=|\mathcal{I}|(T(n)+1)$. Let each $I\in\mathcal{I}$ and each t, $0\leqq t\leqq T(n)$, be encoded as a distinct integer $i=\langle I,t\rangle$, where $1\leqq i\leqq p$. We can assume that the encoding and its decoding are computed in a constant number of steps of a unit-cost RAM or in $O(\log p)$ steps of a log-cost P-RAM.

Our simulating probabilistic P-RAM M' will begin by a series of fork operations yielding RAMs $M_1,\cdots,M_p$. This takes unit-cost time $O(\log p)$ and at most log-cost time $O(\log p^2)$. Each RAM M_i, $1\leqq i\leqq p$, has a local register r_i and an associated global memory location NEXT$_i$ which is initialized as follows: suppose $i=\langle I,t\rangle$ then if I has any immediate successor I', let M_i randomly choose some such I' and store $\langle I',t+1\rangle$ into NEXT$_i$ and otherwise if I has no successors then let M_i store into NEXT$_i$. After

this initialization, each M_i, for $1 \leqq i \leqq p$, synchronously:

(1) reads the contents of NEXT_j into register r_i where j is the contents of NEXT_i, and

(2) then stores NEXT_i with the contents of r_i.

This is repeated $\lceil \log p \rceil$ times. We can assume $\langle I_0(\omega), 0\rangle = 1$ and M_1 is the original RAM of M'. We let M_1 enter the accepting state (so M' accepts) if NEXT_1 ever contains integer $\langle I, t\rangle$ where I is an accepting configuration of M.

If M' enters the accepting state on a particular computation, then there must be a sequence of memory locations $\text{NEXT}_{\langle I_0, 0\rangle}, \cdots, \text{NEXT}_{\langle I_{t-1}, t-1\rangle}$ which are initialized to $\langle I_1, 1\rangle, \cdots, \langle I_t, t\rangle$ where $I_0(\omega) = I_0, I_1, \cdots, I_t$ is an accepting computation sequence of M, and $t \leqq T(n)$. Thus the memory essentially forms a path from $\text{NEXT}_{\langle I_0, 0\rangle}$ to $\text{NEXT}_{\langle I_t, t\rangle}$. On each iteration the path length decreases by a factor of $\frac{1}{2}$. Thus after $\lceil \log p \rceil$ iterations, $\text{NEXT}_{\langle I_0, 0\rangle}$ contains $\langle I_t, t\rangle$.

Suppose $I_0, I_1, \cdots$ is an execution sequence of M, derived from a particular sequence of probabilistic choices ρ. Suppose also that the RAMs of M' make a sequence of probabilistic choices ρ' such that $M_{\langle I_t, t\rangle}$ initially loads $\text{NEXT}_{\langle I_t, t\rangle}$ with $\langle I_{t+1}, t+1\rangle$ for $t = 0, 1, \cdots, T(n)-1$. Then M errors on acceptance (rejection, respectively) of ω when making probabilistic choices ρ iff M' errors on acceptance (rejection, respectively) of ω when making probabilistic choices ρ'. Since ρ and ρ' are chosen randomly, it follows that $M \approx M'$. If M is unit-cost, then $|\mathcal{I}| \leqq I(n)^{S(n)}$; so M' has unit-cost time and unit-cost space bound $O(\log p) = O(S(n) \log I(n) + \log T(n))$ and the processor bound is $p = O(I(n)^{S(n)} T(n))$. If M is log-cost, then $|\mathcal{I}| \leqq 2^{2 \cdot S(n)} = 4^{S(n)}$; so M' has log-cost time bound $O(\log p)^2 = O(S(n) + \log T(n))^2$ and processor bound is $p = O(4^{S(n)} T(n))$.

4.2. Simulation of a probabilistic P-RAM by a probabilistic RAM.

THEOREM 4.2. *Let M be a probabilistic* P-RAM *with time bound $T(n)$, space bound $S(n)$, and processor bound $P(n)$. Then there is a probabilistic* RAM *M' with space bound $O(S(n)+P(n))$ such that $M \approx M'$. Furthermore, if M is unit-cost then M' has unit-cost time bound $O(T(n)P(n))$, and if M is log-cost then M' has log-cost time bound $O(T(n)P(n) \log P(n))$.*

Proof. The simulating probabilistic RAM will have only 5 registers; the first register of M' will store an integer p giving the total number of RAMs currently being executed, and the second register of M' will store an integer designating the RAM currently being simulated; the other 3 registers of M' will be used for arithmetic operations and indirect addressing of memory locations. Suppose each RAM of M has r registers. The registers of the simulated RAMs of M will be stored in a special block of memory locations, which is increased by $r+1$ on every fork operation. The simulation of M' by M is straightforward; on each move of M, M' must simulate a move by each of the currently active RAMs of M. This requires $O(P(n))$ steps if M' is unit-cost, and $O(P(n) \log P(n))$ steps if M' is log-cost. By storing two copies of the memory of M, it is easy to detect simultaneous store conflicts. M' is allowed to enter its accepting state just when the original RAM of M enters its accepting state and there are no simultaneous store conflicts. Since the probabilistic choices taken by the individual probabilistic RAMs are assumed to be independent, and the simulating probabilistic RAM M' takes independent probabilistic choices, the probability of errors of acceptance and rejection of M and M' are identical. Thus $M \approx M'$.

5. Parallel speed-up of probabilistic RAMS.

THEOREM 5.1. *Let M be a probabilistic* RAM *with constructible unit-cost time bound $T(n) \geqq n$ and integer bound $I(n)$. There is a probabilistic* P-RAM *M' such that $M \approx M'$ and M' has unit-cost time bound $O(T(n)(\log T(n)) \log (T(n)I(n)))^{1/2}$.*

Proof. Let $\omega \in \{0, 1\}^*$ be an input string of length n. There is a constant $c \geqq 1$ such that M has at most c choices for next moves at each step. Thus the choices can be represented by a sequence $\rho = \rho_0, \cdots, \rho_{T(n)-1}$ where $\rho_t \in \{1, \cdots, c\}$. The parallel simulation of M by M' begins by making independent random choice of $\rho_0, \cdots, \rho_{T(n)-1}$ in $O(\log T(n))$ parallel time, and storing these choices in distinct memory locations.

The fundamental idea (previously used in Hopcroft, Paul and Valiant (1975) and Dymond (1980) for speed-up of deterministic Turing machines) is to partition the $T(n)$ steps into consecutive intervals of length L, $1 \leqq L \leqq T(n)$ to be determined below.

Let q be the number of states in the finite control of M. Suppose in the following that M is unit-cost. Then M can read from and store into at most $3L$ registers and memory locations within a time interval Δ of length L. Furthermore, we can encode by a positive integer not more than $s = q(T(n)I(n))^{3L}$ the current state and the contents and addresses of the registers and memory locations read from (or stored into) during Δ.

Let $H = \lceil T(n)/L \rceil - 2$. For each $t = 0, L, 2L, \cdots, HL$ the simulating M' constructs in global memory a table PREDICT_t. Given a positive integer $i \leqq s$ encoding possible state of M and contents and addresses of all registers and memory locations to be read during time interval $\Delta_t = \{t, t+1, \cdots, t+L-1\}$, $\text{PREDICT}_t(i)$ is a positive integer not more than s encoding the contents and addresses of all registers and memory locations to L stored into during Δ_t using the predetermined choice sequence $\rho_t, \rho_{t+1}, \cdots, \rho_{t+L-1}$. However, let $\text{PREDICT}_t(i) = 0$ if this choice sequence requires reading a register or memory location whose contents are not defined by i, or if the contents of a register or memory location are provided by i but are not read from. These tables can be constructed in parallel by M' in unit-cost time $O(L + \log s)$.

$T(n)$ distinguished global memory locations of M' are used to store the contents of the memory of M. Also, a special register is used to store the state of the finite control of M. These are initialized as in the initial configuration of M. The simulation of M by M' will then proceed sequentially in H phases, each corresponding to a time interval Δ_t, for $t = 0, L, 2L, \cdots, HL$.

Suppose at the start of the phase corresponding to interval Δ_t, M' is currently storing (as described above) the configuration I_t of M, where $I_0, I_1, \cdots, I_t$ is the sequence of configurations of M induced from $I_0 = I_0(\omega)$ by the choice sequence $\rho_0, \rho_1, \cdots, \rho_{t-1}$ chosen by M' at the start of the simulation. Then there is a unique sequence of configurations $I_t, I_{t+1}, \cdots, I_{t+L}$ induced by the predetermined choice sequence $\rho_t, \rho_{t+1}, \cdots, \rho_{t+L-1}$. Hence there is a unique i_t, $1 \leqq i \leqq r$, such that $\text{PREDICT}_t(i_t) \neq 0$ and i_t encodes contents of registers and memory locations consistent with I_t. $\text{PREDICT}_t(i_t)$ is encoded and is used to update the memory of M' to store the configuration I_{t+L}. After the phase associated with time interval Δ_{HL}, M' simulates M step by step sequentially for $t = (H+1)L, (H+1), \cdots, T(n)$. Let the original RAM of M' enter the accepting state if the simulated M does. Since the choice sequence $\rho_0, \cdots, \rho_{T(n)-1}$ is chosen randomly by M', it induces a random computation sequence of M from $I_0(\omega)$, so $M \approx M'$.

The unit-cost time for initialization and computation of the PREDICT tables is $O(L + \log s) = O(L \log (T(n)I(n)))$. The unit-cost time for each phase is $O(\log \log s) = O(\log (L \log (T(n)I(n))))$ since encoding and decoding of elements of the PREDICT tables is done in parallel. There are $< T(n)/L$ phases. Thus the total unit-cost simulation time is

$$O(L \log (T(n)I(n))) + (T(n)/L)O(\log (L \log (T(n)I(n)))) + L$$
$$= O(T(n)(\log T(n)) \log (T(n)I(n)))^{1/2},$$

for

$$L = (T(n)(\log T(n)))/\log (T(n)I(n))^{1/2}. \qquad \square$$

6. Elimination of probabilistic choice in parallel computations. Let M be a (uniform) probabilistic P-RAM with time bound $T(n)$ and processor bound $P(n)$. Let $Z(n)$ be the maximum number of probabilistic choices made by all the RAMs of M on any input of length n. (Note that $Z(n) \leqq T(n)P(n)$.) Let $\varepsilon_A(n)$, $\varepsilon_R(n)$ be the acceptance and rejection error functions for M, and let $\varepsilon(n) = \max(\varepsilon_A(n), \varepsilon_R(n))$. Also, let $\lambda(n) = (1+2n)/\log(1/(4\varepsilon(n)(1-\varepsilon(n))))$. We assume $\varepsilon(n) < \frac{1}{2}$ so $\lambda(n)$ is finite.

The following theorem states that we can eliminate the probabilistic choice in M by introducing *nonuniformity with advice bound* $A(n)$: i.e., we allow the nonuniform P-RAM to have in the initial configuration for each input length $n \geqq 0$, a distinguished sequence of $A(n)$ memory locations each initialized to either 0 or 1 and fixed for all inputs of length n.

THEOREM 6.1. *For any $\tau(n)$, $1 \leqq \tau(n) \leqq \lambda(n)$, there is a deterministic nonuniform* P-RAM $\hat{M}$ *which accepts $L(M)$ with time bound $O(T(n)\tau(n) + \log(\lambda(n)/\tau(n)))$, processor bound $O(P(n)\lambda(n)/\tau(n))$, and advice bound $O(\lambda(n)Z(n))$.*

Note. Thus to eliminate probabilistic choice we have a trade-off between an increase in time bounds and an increase in processor bounds. However, if $\varepsilon(n)$ decreases exponentially, then neither the time bound nor the processor bound are asymptotically increased.

Theorem 6 will be proved as follows: first we show that we can eliminate probabilistic choice from M if $\varepsilon(n)$ is sufficiently small; then we show how to make $\varepsilon(n)$ sufficiently small.

We can assume a constant $c \geqq 1$ such that M has $\leqq c^{P(n)}$ choices of moves next from any configuration. Fix some input length $n \geqq 0$. A parallel *choice sequence* ρ is of the form $\rho_0, \rho_1, \cdots, \rho_{T(n)-1}$ where $\rho_i \in \{1, \cdots, c^{P(n)}\}$ for $i = 0, 1, \cdots, T(n)-1$. Let $R_{T(n)}$ be all choice sequences of length $T(n)$. Given an input $\omega \in \{0, 1\}^n$, a choice sequence in $S_{T(n)}$ induces a computation sequence of M. Let $R_{T(n)}(\omega) = \{\rho \in R_{T(n)} | (\omega \in L(M)$ and M has an accepting computation sequence on input ω and choice sequence $\rho)$ or $(\omega \notin L(M)$ and M has a nonaccepting computation sequence on input ω and choice sequence $\rho)\}$.

LEMMA 6.1. *Suppose $\varepsilon(n) < 2^{-n}$. Then there is a deterministic nonuniform* P-RAM $\hat{M}$ *which accepts $L(M)$ with time bound $O(T(n))$, processor bound $P(n)$ and advice bound $O(Z(n))$.*

Proof. It suffices to show (*):

(*) if $\varepsilon(n) < 2^{-n}$ then there exists some choice sequence $\rho^* \in R_{T(n)}$ such that for all $\omega \in \{0, 1\}^n$, $\rho^* \in R_{T(n)}(\omega)$.

Our proof is by contradiction. For each $\rho \in R_{T(n)}$ let $f(\rho) = |\{\omega \in \{0, 1\}^n | \rho \in R_{T(n)}(\omega)\}|$ and let $r = |R_{T(n)}|$. Note that for each $\omega \in \{0, 1\}^n$, $|R_{T(n)}(\omega)| \geqq r(1-\varepsilon(n)) > r(1-2^{-n})$. Suppose (*) does not hold, so $2^n > f(\rho)$ for all $\rho \in R_{T(n)}$. Hence

$$r(2^n - 1) \geqq \sum_{\rho \in R_{T(n)}} f(\rho) = \sum_{\omega \in \{0,1\}^n} |R_{T(n)}(\omega)| > r(1-2^{-n})2^n = r(2^n - 1),$$

a contradiction. $\square$

LEMMA 6.2. *For any $\tau(n)$, $1 \leqq \tau(n) \leqq \lambda(n)$, there is a probabilistic* P-RAM M' *which accepts $L(M)$ with acceptance and rejection errors $\varepsilon'_A(n)$, $\varepsilon'_R(n)$ where $\max(\varepsilon'_A(n), \varepsilon'_R(n)) < 2^{-n}$, and time bound $O(T(n)\tau(n) + \log(\lambda(n)/\tau(n)))$ and processor bound $O(P(n)\lambda(n)/\tau(n))$.*

Proof. Let $\omega \in \{0, 1\}^n$ be the input string, for some $n \geqq 0$. Our probabilistic P-RAM M' will simulate M on input ω a total of $\lambda(n)$ times; these simulations will be done by $\lceil \lambda(n)/\tau(n) \rceil$ groups of $P(n)$ probabilistic RAMs, with each group simulating M $\tau(n)$ times. M' is allowed to enter an accepting configuration only if M enters an accepting configuration on at least $\lambda(n)/2$ of the $\lambda(n)$ trials. (This technique of determining the consensus of a series of trials is due to Bennett and Gill (1981).) The count of successful trials can be computed in $\log(\lambda(n)/\tau(n))$ parallel time. The acceptance error of M' is

$$\begin{aligned}\varepsilon'_A(n) &= \sum_{i=\lambda(n)/2}^{\lambda(n)} \binom{\lambda(n)}{i} \varepsilon(n)^i (1-\varepsilon(n))^{\lambda(n)-i} \\ &\leqq (4\varepsilon(n)(1-\varepsilon(n)))^{\lambda(n)/2} \quad \text{by bounds of Chernoff (1952) also given in Feller (1957)} \\ &< 2^{-n} \quad \text{for given } \lambda(n) > 2n/\log(1/(4\varepsilon(n)(1-\varepsilon(n)))).\end{aligned}$$

Also we can similarly show the error of rejection $\varepsilon'_R(n) < 2^{-n}$. Hence $\max(\varepsilon'_A(n), \varepsilon'_R(n)) < 2^{-n}$ as claimed. □

Theorem 6 follows immediately by applying to Lemma 6.1 the probabilistic P-RAM M' derived by Lemma 6.2. By applying Theorem 6 to Theorems 3.1–3.2 we have

COROLLARY 6.1. *There exist unit-cost nonuniform deterministic* P-RAMs *with time bound* $O(\log n)$, *polynomial processor and advice bound, which given a graph* G *with* n *vertices, can test* (a) *whether* G *has a path between two given vertices and can also test* (b) *whether* G *is not bipartite.*

COROLLARY 6.2. *There exists a unit-cost nonuniform deterministic* P-RAM *with time bound* $O(\log n)^2$, *processor and advice bound* $n^{O(1)}$ *which can test if a graph of* n *vertices has a perfect matching.*

7. Conclusion. This paper has primarily considered the power of probabilistic choice for parallel RAMs. Theorems 3.2–3.3 also hold for fixed connection parallel networks with probabilistic processors. Theorems 4.1 and 4.2 can be extended to similar simulation results for other probabilistic parallel machines, such as the hardware modification machines (HMMs) of Cook (1980) augmented with probabilistic choice [see Reif (1981)]. Some similar results were obtained in an independent investigation of Borodin et al. (1980). Also Theorem 6 easily generalizes to other probabilistic parallel machines such as HMMs and circuits with probabilistic choice.

Acknowledgments. The author was informed by Larry Russo of the consensus technique previously used by Bennett and Gill (1980) for decreasing errors of probabilistic choice. Steven Cook and Paul Spirakis gave helpful comments on reading a preliminary draft of this paper.

REFERENCES

L. ADLEMAN, *Two theorems on random polynomial time*, Proc. 19th IEEE Symposium on the Foundations of Computer Science, Ann Arbor, MI, 1978, pp. 75–83.

L. ADLEMAN, K. MANDERS AND G. MILLER, *On taking roots in finite fields*, Proc. 18th IEEE Symposium on the Foundations of Computer Science, 1977, pp. 175–178.

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

R. ALELIUNAS, R. M. KARP, R. H. LIPTON, L. LOVÁSZ AND C. RACKOFF, *Random walks, universal traversal sequences, and complexity of maze problems*, Proc. 20th Annual IEEE Symposium on the Foundations of Computer Science, 1979, pp. 218–223.

A. M. BARZDIN, *On computability by probabilistic machines*, Dokl. Akad. Nauk. SSSR, 189 (1969), pp. 699–702; Soviet Math. Dokl., 10 (1969), pp. 1464–1467.

C. H. BENNETT AND J. GILL, *Relative to a random oracle* A, $P^A \neq NP^A \neq coNP^A$ *with probability* 1, this Journal, 10 (1981), pp. 96–113.

E. R. BERLEKAMP, *Factoring polynomials over large finite fields*, Math. Comp., 24 (1970), pp. 713–735.

A. BORODIN, J. VON ZUR GOTHEN AND J. HOPCROFT, *Fast parallel matrix and gcd computations*, 23rd IEEE Foundations of Computer Science Conference, 1982, pp. 65–71.

A. BORODIN, J. HOPCROFT, M. PETERSON, L. RUZZO AND M. TOMPA, Unpublished notes, July 1980.

S. A. COOK, *Towards a complexity theory of synchronous parallel computation*, presented at Internationales Symposium über Logik und Algorithmik zu Ehren von Professor Horst Specker, Zürich, Switzerland, Feb. 1980.

L. CSANKY, *Fast parallel matrix inversion algorithms*, this Journal, 5 (1976), pp. 618–623.

H. CHERNOFF, *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Ann. Math. Statist., 23 (1952), pp. 493–507.

P. W. DYMOND, *Speedup of multi-tape Turing machines by synchronous parallel machines*, Tech. Rep., Dept. Electrical Engineering and Computer Science, Univ. California, San Diego.

P. W. DYMOND AND S. A. COOK, *Hardware complexity and parallel computation*, IEEE Foundations of Computer Science, Conference, 1980.

W. FELLER, *An Introduction to Probability Theory and its Applications*, John Wiley, New York, 1957.

R. FREIVALDS, *Fast probabilistic algorithms*, 8th MFCS, 1979.

S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, Proc. 10th Annual ACM Symposium on the Theory of Computing, San Diego, CA, 1978, pp. 114–118.

N. FRANCEZ AND RODEN, *A distributed data type implemented by a probabilistic communication scheme*, Proc. 21st Annual IEEE Symposium on the Foundations of Computer Science, Syracuse, New York, Oct. 1980, pp. 373–379.

J. GILL, *Complexity of probabilistic Turing machines*, this Journal, 6 (1977), pp. 675–695.

L. GOLDSCHLAGER, *A unified approach to models of synchronous parallel machines*, Proc. 10th Annual ACM Symposium on the Theory of Computing, San Diego, CA, 1978, pp. 89–94.

D. S. HIRSCHBERG, A. K. CHANDRA AND D. V. SARAWATA, *Computing connected components on parallel computors*, Comm. ACM, 22 (1979), pp. 461–464.

J. E. HOPCROFT AND R. M. KARP, *An* $n^{5/2}$ *algorithm for maximum matchings in bipartite graphs*, this Journal, 2 (1973), pp. 225–231.

J. E. HOPCROFT, W. PAUL AND L. VALIANT, *On time versus space and related problems*, Proc. 16th Annual IEEE Symposium on Foundations of Computer Science, Berkeley, CA, 1975, pp. 57–64.

O. H. IBARRA AND S. MORAN, *Probabilistic algorithms for deciding equivalence of straight-line programs*, Tech. Rep. 80–12, Computer Science Dept., Univ. Minnesota, Minneapolis, March 1980.

D. LEHMAN AND M. RABIN, *On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers' problem*, Proc. 8th ACM Symposium on Principles of Programming Languages, Jan. 1981, to appear.

L. LOVÁSZ, *On determinants, matchings, and random algorithms*, to appear, 1980.

F. P. PREPARATA AND D. V. SARWATE, *An improved parallel process-bound in fast matrix inversion*, Inform. Processing Lett., V7 (1978), pp. 148–150.

M. O. RABIN, *Probabilistic algorithms*, Algorithms and Complexity, New Directions and Recent Results, J. Traub, ed., Academic Press, New York, 1974.

———, *Probabilistic algorithms in finite fields*, this Journal, 9 (1980), pp. 273–280.

———, *N-process synchronization by a* $4 \log_2 N$*-valued shared variable*, Proc. 21st Annual ACM Symposium on the Foundations of Computer Science, Syracuse, NY, 1980, pp. 407–410.

J. H. REIF, *Symmetric complementation*, 14th Annual ACM Symposium on Theory of Computing, San Francisco, May 1982; this Journal, to appear.

J. H. REIF AND P. SPIRAKIS, *Distributed algorithms for synchronizing interprocess communication within real time*, 13th Annual ACM Symposium on the Theory of Computing, Milwaukee, WI, 1981.

———, *Unbounded speed variability in distributed communication systems*, 9th Annual ACM Symposium on Principles of Programming Languages, Albuquerque, NM, Jan. 1982.

———, *Real time resource allocation in a distributed system*, ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Ottawa, Canada, Aug. 1982.

J. H. REIF AND L. G. VALIANT, *A logarithmic time sorting algorithm for linear size networks*, 15th Annual ACM Symposium on Theory of Computing, Boston, MA, 1983, pp. 10–16.

R. REISCHUK, *A fast probabilistic parallel sorting algorithm*, 22nd IEEE Symposium on the Foundations of Computer Science, Nashville, TN, Oct. 1981.

J. T. SCHWARTZ, *Fast probabilistic algorithms for verification of polynomial identities*, J. Assoc. Comput. Mach., 27 (1980), pp. 701–717.

J. SIMON, *On some central problems in computational complexity*, TR75-224, Dept. Computer Science, Cornell Univ., Ithaca, NY, 1975.

R. SOLOVAY AND V. STRASSEN, *A fast Monte-Carlo test for primality*, this Journal, 5 (1977), pp. 84–85.

L. G. VALIANT AND G. J. BREBNER, *Universal schemes for parallel computation*, Proc. 13th Annual ACM Symposium on the Theory of Computing, Milwaukee, WI, 1981, pp. 263–277.

J. C. WYLLIE, *The complexity of parallel computations*, Ph.D. thesis and TR-79-387, Dept. Computer Science, Cornell University, Ithaca, NY, 1979.

R. ZIPPEL, *Probabilistic algorithms for sparse polynomials*, EUROSAM Proc. 1979.

SIAM J. COMPUT.
Vol. 13, No. 1, February 1984
© 1984 Society for Industrial and Applied Mathematics
0097-5397/84/1301-0005 $01.25/0

CLIQUE COVERING OF GRAPHS IV. ALGORITHMS*

NORMAN J. PULLMAN†

Abstract. We present linear time algorithms for computing the minimum number of complete subgraphs needed to cover or partition the edges of any simple graph G with maximal degree less than 5.

Key words. clique, covering, partition, algorithm, complete subgraph

1. Introduction and summary. For our purposes, graphs have no self-adjacent vertices or multiple edges. The graph K_n on n vertices in which every pair of distinct vertices is joined by an edge is said to be *complete*. The literature lacks agreement on a noun to describe a complete subgraph of a given graph. Bondy and Murty [2] for example, describe the vertex set of such a graph as a "clique" while Harary [5] uses that word to describe maximal complete subgraphs. We have steered a middle course with Orlin [7] and describe any complete subgraph K of a graph G as a *clique* of G. If K has k vertices we call it a clique of *order* k or a k*-clique*. If $k=1$ then K is just a vertex, if $k=2$ we refer to K as an *edge* and if $k=3$ we refer to K as a *triangle*. A *clique-covering of* G is a family $\mathscr{C}$ of cliques of G such that every edge of G is in at least one member of $\mathscr{C}$. If the members of $\mathscr{C}$ are pairwise edge-disjoint then $\mathscr{C}$ is said to be a *clique-partition of* G. The cardinality of $\mathscr{C}$ is denoted by $|\mathscr{C}|$. A clique-covering (respectively, -partition) $\mathscr{C}$ is said to be *minimal* if $|\mathscr{C}'| \geqq |\mathscr{C}|$ for all clique-coverings (-partitions) $\mathscr{C}'$. The *clique-covering number* of G, $\mathrm{cc}(G)$ is the cardinality of a minimal clique-covering; the *clique-partition number of* G, $\mathrm{cp}(G)$ is the cardinality of a minimal clique-partition. Since every clique-partition is a clique-covering, we have immediately that $\mathrm{cc}(G) \leqq \mathrm{cp}(G)$; and equality holds when (but not only when) G is triangle-free, in which case $\mathrm{cc}(G)=\mathrm{cp}(G)$ is the number of edges of G. If G has no edges then $\mathrm{cc}(G)=\mathrm{cp}(G)=0$.

Various authors have considered the problem of estimating and calculating these numbers: for example, Erdös, Goodman and Pósa [4], Harary [5, pp. 17–20], Lovász [6], Orlin [7], and (in matrix-theoretic terms) Ryser [11] and [12]. Recently, further results were obtained by the present writer with deCaen [8], [10] and Donald [9].

In this paper, algorithms are presented for determining minimal clique-coverings and -partitions and clique-covering and -partition numbers of graphs with maximal degree at most 4.

A brute-force approach to the calculation of these entities for graphs on n vertices would involve an exponential function of n operations (exponential time) even if the maximal degree is at most 4. The algorithms we propose accomplish their work in $O(n)$ operations (linear time).

Since the general problem (arbitrary maximal degree) of computing $\mathrm{cc}(G)$ is known to be NP-complete and that of computing $\mathrm{cp}(G)$ is probably NP-complete[1] (see [7]), we feel that linear time algorithms for a moderately large subclass of graphs may be of interest.

The five algorithms are presented in § 4. Mathematical preliminaries are in § 2 and § 3. The longer proofs are in § 6.

---

* Received by the editors January 14, 1981, and in revised form November 29, 1982. This work was supported in part by the Natural Science and Engineering Research Council of Canada under grant A4041.
† Mathematics and Statistics Department, Queen's University, Kingston, Ontario, Canada K7L 3N6.
[1] *Note added in proof.* The NP-completeness of the clique partition problem was proven in I. Holyer's paper: *The* NP-*completeness of some edge partition problems*, this Journal, 10 (1981), pp. 713–717.

Here is a summary of the concepts behind the algorithms and an outline of how they operate.

A subgraph H of G *separates the cliques* of G if every clique of G has either all or none of its edges in H. The graph G is *clique-separable* if a subgraph H, whose edge-set is a nonempty proper subset of the edge-set of G, separates the cliques of G. The maximal clique-inseparable subgraphs of G are called *clique-blocks*. They partition the edges of G. Moreover, if $B_1, B_2, \cdots$ are its clique-blocks, then $\mathrm{cp}(G) = \sum_i \mathrm{cp}(B_i)$ and $\mathrm{cc}(G) = \sum_i \mathrm{cc}(B_i)$.

Suppose G has n vertices, none of which has degree exceeding 4. If G is clique-inseparable, and contains some edges, then G is one of sixteen possible graphs if $n \leqq 6$ and one of three possible graphs if $n > 6$. This fact (Theorem 1) is the basis of our algorithms.

Algorithm 1 works as follows. Given G,

(a) a clique-block B containing a triangle of G is determined,

(b) identified (as one of the nineteen possibilities) by easily ascertained earmarks;

(c) the number of its edges $\varepsilon(B)$, $\mathrm{cc}(B)$ and $\mathrm{cp}(B)$ are then known. The quantities $\varepsilon(B) - \mathrm{cc}(B)$ and $\varepsilon(B) - \mathrm{cp}(B)$ are then subtracted from subtotals initially equal to $\varepsilon(G)$, the number of edges in G.

(d) B is deleted from the graph and, if any triangles remain, the cycle (a) (b) (c) (d) is repeated. When none remain, the subtotals are then, $\mathrm{cc}(G)$ and $\mathrm{cp}(G)$ respectively.

The peculiar method of accumulating subtotals for $\mathrm{cc}(G)$ and $\mathrm{cp}(G)$ given in (c) was chosen because the algorithm stops when there are no more triangles, and this method automatically includes the one-edge clique blocks in the totals.

To carry out the steps (a), a graph associated with G whose vertices are triangles in G is computed. One of its connected components is determined at each iteration of (a). This can all be done in a total of $O(n)$ operations.

Algorithms 2 and 2′ for finding $\mathrm{cc}(G)$ and $\mathrm{cp}(G)$ respectively, work in a similar way, but they have a routine built into them that calculates the blocks directly. This may make them more convenient to use than Algorithm 1. Here is how they work.

(a) an edge of G is chosen and the clique-block B containing it is found,

(b) B is identified (as in Algorithm 1),

(c) the numbers $\mathrm{cc}(B)$ and $\mathrm{cp}(B)$ are then known and added to running subtotals which were zero initially, and

(d) the edges of B are removed from G. If any edges remain the cycle is repeated. Eventually G is emptied of its edges and the subtotals give the required numbers.

Algorithms 3 and 1′ only calculate $\mathrm{cp}(G)$. They operate on a somewhat different principle. A subgraph D of G is *deletable* if $\mathrm{cp}(G) = \mathrm{cp}(\bar{D}) + \mathrm{cp}(D)$, where $\bar{D}$ is obtained by deleting the edges (but not the vertices) of D from G. For example all 4-cliques are deletable. Algorithm 3 tries to avoid the computation of a whole clique-block containing an initial edge e. Instead, it first finds the number of triangles sharing edges with those triangles containing e. Next, by subjecting the numbers to a short list of tests, it determines a deletable edge, triangle or set of triangles. It then removes the edges of these deletable objects from the graph, while adding the appropriate number (e.g., 1 if the object is an edge or triangle) to a running subtotal. A new pass is then begun if any edges remain in the graph. When no edges remain, the subtotal is $\mathrm{cp}(G)$. Algorithm 1′ is similar but much simpler. Unlike the other four algorithms, it can only be used when G has no 4-cliques.

These algorithms each remove a positive number ε of edges at each pass. Each pass takes a total of at most $c\varepsilon$ operations, where c is a constant independent of G. This is mainly due to the fact that the maximum of the degrees of the n vertices is at most 4. Since G has at most $2n$ edges the algorithms run in at most $2cn$ operations, i.e., in linear time. If G is presented as a vertex-incidence table then the input involves at most $5n$ symbols, so $O(n)$ operations are required from input to output.

2. Preliminaries. Following Bondy and Murty [2], as we do for most of the notation, we use $\bar{H}(G)$ to denote the subgraph of G obtained by deleting the edges, but not the vertices, of H from G. This subgraph $\bar{H}(G)$ is called the *complement of H in G*. We write $\bar{H}$ instead of $\bar{H}(G)$ when it will cause no confusion.

We say that a subgraph *separates the cliques of G*, if, for every clique K of G, either every edge of K lies in H or every edge of K lies in $\bar{H}$.

The proofs omitted from the following lemmas can be found in [8].

LEMMA 2.1 [8, Lemma 2.1.1]. *The following statements are equivalent*:

(a) *H separates the cliques of G.*

(b) *$\bar{H}$ separates the cliques of G.*

(c) *If K is a triangle in G then all its edges lie in H or all its edges lie in $\bar{H}$.*

LEMMA 2.2 [8, Theorem 2.1]. *If H separates the cliques of G then* $\operatorname{cp}(G)=\operatorname{cp}(H)+\operatorname{cp}(\bar{H})$ *and* $\operatorname{cc}(G)=\operatorname{cc}(H)+\operatorname{cc}(\bar{H})$.

LEMMA 2.3. *If $\mathcal{H}$ is a family of subgraphs each separating the cliques of G then the union of the subgraphs in $\mathcal{H}$ separates the cliques of G.*

Proof. Let K be a clique in G and L be the union of all members of $\mathcal{H}$. If $K\not\subset L$, then $K\not\subseteq H$ for all $H\in\mathcal{H}$; therefore $K\subseteq\bar{H}$ for all $H\in\mathcal{H}$, consequently $K\subseteq\bigcap\{\bar{H}: H\in\mathcal{H}\}=\bar{L}$. Therefore L separates the cliques of G.

Suppose H is a subgraph of G. Define the *neighborhood* of H, $N(H)$, to consist of H and every vertex and edge of G adjacent to vertices of H. Letting $\Delta(G)$ denote the maximum of the degrees of the vertices of G, we have:

LEMMA 2.4 [8, Lemma 2.1.2]. *If $\Delta(G)=k$ and K is a k-clique, then $N(K)$ separates the cliques of G.*

If v is a vertex of G adjacent to, but not a vertex of, a subgraph H, we say v is *externally adjacent* to H. If vx is an edge of G, v is externally adjacent to H and x is a vertex of H, then we say that vx is *externally adjacent to H (at x)*. Let $\nu'(H)$ and $\varepsilon'(H)$ denote the number of vertices and edges of G externally adjacent to H.

LEMMA 2.5. *If $\Delta(G)=k$ and K is a k-clique of G then $\nu'(K)\leqq k$, $\varepsilon'(K)\leqq k$ and, unless $N(K)$ is a $(k+1)$-clique,* $\operatorname{cc}(N(K))=1+\nu'(K)$ *and* $\operatorname{cp}(N(K))=1+\varepsilon'(K)$.

Proof. See § 6.

If H is a subgraph containing some, but not all of the edges and vertices of G, then we say that H is a *proper* subgraph. If a proper nonempty subgraph separates the cliques of G, then we say that G is *clique-separable*. Otherwise G is *clique-inseparable*. Note that empty graphs (those with no edges) are clique-inseparable. If a subgraph B of G separates the cliques of G, but no proper nonempty subgraph of B does so, then B is called a *clique-block* of G. Note that B is then clique-inseparable in itself. (If L were a proper subgraph of B separating the cliques of B then L would also separate the cliques of G because $\bar{L}(B)\subseteq\bar{L}(G)$.) Therefore a subgraph B is a clique-block of G if and only if B is a clique-inseparable graph and B is not a subgraph of any other clique-inseparable graph contained in G. The definitions directly imply the following lemma.

LEMMA 2.6. (a) *Edges contained in no triangles of* G, *and* (b) *triangles sharing edges with no other triangles of* G, *are clique-blocks in* G.

LEMMA 2.7. *The family of all clique-blocks in* G *partitions the edge-set of* G.

Proof. See § 6.

A subgraph D of G is *deletable* if $\mathrm{cp}(G) = \mathrm{cp}(\bar{D}) + \mathrm{cp}(D)$. For example, by Lemma 2.2, subgraphs that separate the cliques of G are deletable. Applying Lemma 2.4 (and Lemma 2.5 for part (e)) we have:

LEMMA 2.8. *The following are deletable subgraphs of* G:

(a) *isolated vertices,*

(d) *neighborhoods of* $\Delta(G)$*-cliques* (*in particular,* $(\Delta(G)+1)$*-cliques*),

(c) *triangles sharing no edges with other triangles of* G,

(d) *neighborhoods of* $\Delta(G)$*-cliques* (*in particular,* $(\Delta(G)+1)$*-cliques*),

(e) $\Delta(G)$*-cliques.*

Moreover, (f) *if* H' *is a deletable subgraph of* H *and* H *is a deletable subgraph of* G, *then* H' *is a deletable subgraph of* G.

3. Triangle graphs and their properties. If G has some triangles let T be the graph whose vertices are the triangles of G, with two distinct vertices of T deemed adjacent in T if, as triangles in G, they share a common edge. We refer to T as the *triangle graph* of G and denote it by $T(G)$. If G is triangle-free, then we define $T(G) = \varnothing$.

LEMMA 3.1. *If every edge of* G *lies in some triangle of* G, *then the triangle graph of* G *is connected if and only if* G *is clique-inseparable.*

Proof. See § 6.

COROLLARY. *The connected components of* $T(G)$ *are the images under* T *of those clique-blocks of* G *which are not edges.*

LEMMA 3.2. *Suppose* G *contains no* 4*-cliques and* $\mathcal{T}(\mathcal{C})$ *denotes the set of triangles in a clique-partition* $\mathcal{C}$ *of* G. *If* $\mathcal{T}$ *is a maximum independent set of vertices of* $T(G)$, *then there exists a minimal clique-partition* $\mathcal{C}$ *of* G *such that* $\mathcal{T} = \mathcal{T}(\mathcal{C})$. *Conversely, if* $\mathcal{C}$ *is a minimal clique-partition of* G, *then* $\mathcal{T}(\mathcal{C})$ *is a maximum independent set of vertices of* $T(G)$.

Proof. See § 6.

In the notation of [2], $\varepsilon(H)$ denotes the number of edges of H and $\alpha(H)$ denotes the size of a maximum independent set of vertices of H.

COROLLARY 1. *If* G *is* 4*-clique-free, then*

$$\mathrm{cp}(G) = \varepsilon(G) - 2\alpha(T(G)).$$

COROLLARY 2. *If* P_j, P_j' *and* C_j *are defined as in Fig.* 1, *then* (a) $\mathrm{cp}(P_j) = \mathrm{cp}(P_j') = j + (\frac{1}{2})(1+(-1)^j)$ *and* (b) $\mathrm{cp}(C_j) = j + (\frac{1}{2})(1-(-1)^j)$.

Proof. (a) The graph $T(P_j)$ is a simple path of length $j-1$,

$$T(P_j') = T(P_j), \quad \varepsilon(P_j') = \varepsilon(P_j) = 2j+1, \quad \alpha(T(P_j)) = \frac{j + (\frac{1}{2})(1-(-1)^j)}{2}.$$

(b) The graph $T(C_j)$ is a simple cycle of length j,

$$\varepsilon(C_j) = 2j, \qquad \alpha(T(C_j)) = \frac{j - (\frac{1}{2})(1-(-1)^j)}{2}.$$

The graphs of Fig. 1 and the graphs G_i of Fig. 2 are all clique-inseparable by Lemma 3.1 as their triangle graphs are simple paths (in the case of P_i, P_j'), simple cycles (in the case of C_j) and other connected graphs T_j. The following theorem

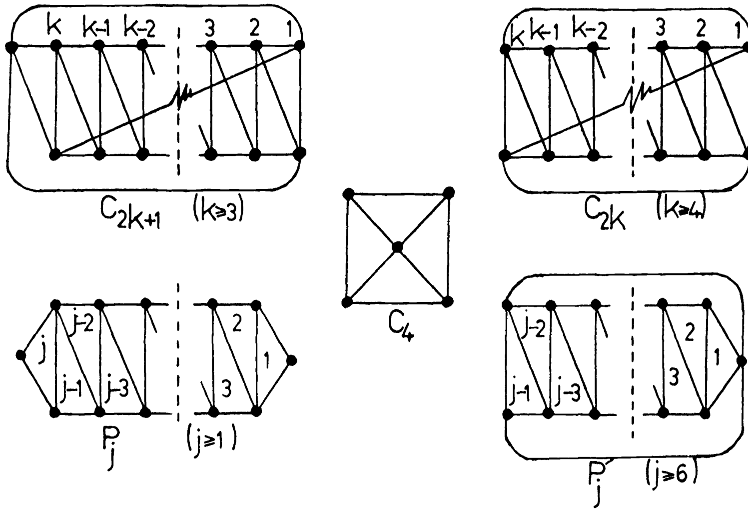


FIG. 1

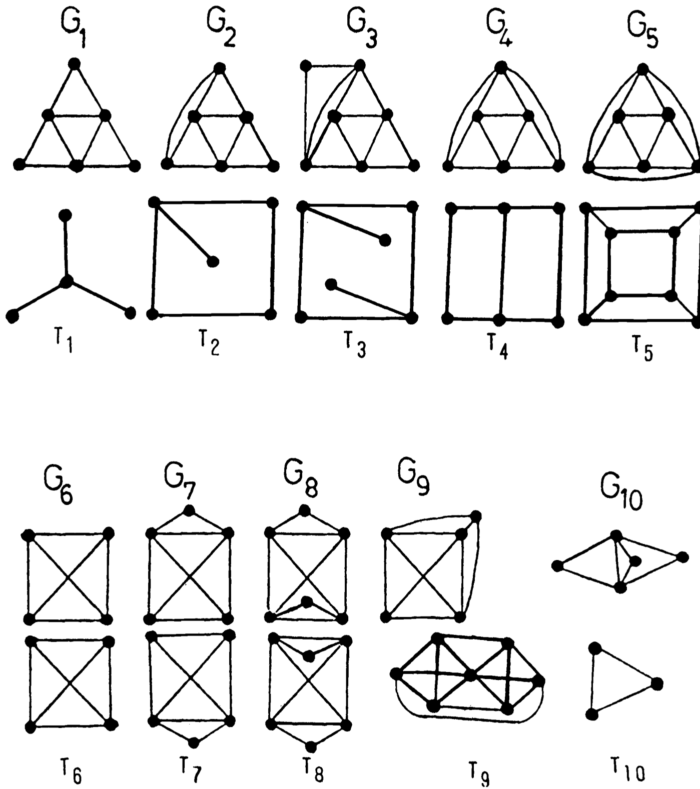


FIG. 2

establishes that apart from K_2, K_5 and $\bar{K}_n$ (the edgeless graph on n vertices), they are the only clique-inseparable graphs of maximum degree at most 4.

THEOREM 1. *If G is clique-inseparable and $\Delta(G) \leqq 4$, then G is $\bar{K}_n$, K_2, K_5, one of the ten graphs G_i, one of the graphs P_j, P_j' or C_j.*

Proof. See § 6.

COROLLARY 1. *If $\Delta(G) \leqq 4$, then the only connected triangle graphs $T(G)$ are simple paths of all lengths $l \geqq 0$, simple cycles of lengths $l \geqq 3$, except $l = 5$ and $l = 6$, the graphs $T(G_j)$ for $1 \leqq j \leqq 9$, and $T(K_5)$.*

Table 1 lists all the clique-inseparable, nonempty graphs whose triangle graphs have eight or fewer vertices. It also lists the anatomical features by which our algorithms can identify them, and it lists their clique-covering and -partition numbers. These numbers were obtained by Lemma 3.2 and its second corollary. Following [2] we let $\nu(L)$ denote the number of vertices in L and $\delta(L)$ denote the minimum of the degrees of its vertices.

TABLE 1

G	P_1	P_2	P_3	G_{10}	P_4	C_4	G_1	G_6	P_5	G_2	G_7	P_6
$\nu(T(G))$	1	2	3	3	4	4	4	4	5	5	5	6
$\varepsilon(T(G))$	0	1	2	3	3	4	3	6	4	5	8	5
$\delta(T(G))$	0	1	1	2	1	2	1	3	1	1	2	1
$\Delta(T(G))$	0	1	2	2	2	2	3	3	2	3	4	2
cc (G)	1	2	3	3	4	4	3	1	5	4	2	6
cp (G)	1	3	3	5	5	4	3	1	5	4	3	7
$\varepsilon(G)$	3	5	7	7	9	8	9	6	11	10	8	13

G	P'_6	G_3	G_4	G_8	P_7	P'_7	C_7	G_9	P_8	P'_8	C_8	G_5
$\nu(T(G))$	6	6	6	6	7	7	7	7	8	8	8	8
$\varepsilon(T(G))$	5	6	7	10	6	6	7	15	7	7	7	12
$\delta(T(G))$	1	1	2	2	1	1	2	4	1	1	2	3
$\Delta(T(G))$	2	3	3	4	2	2	2	6	2	2	2	3
cc (G)	6	4	5	3	7	7	7	2	8	8	8	4
cp (G)	7	4	5	5	7	7	8	4	9	9	8	4
$\varepsilon(G)$	13	12	11	10	15	15	14	9	17	17	16	12

Algorithm 3 computes cp (G) by locating, identifying and removing a sequence of deletable subgraphs, while adding their clique-partition numbers. The location and identification procedure is based on the following lemma.

LEMMA 3.3. *Suppose $\Delta(G) \leqq 4$, t is a triangle in G, $N(t)$ is the family of triangles in G sharing edges with t, and $H(t)$ is the subgraph of G generated by $N(t)$. Let $d(t)$ be the degree of t as a vertex of $T(G)$.*

(a) *If $d(t) \leqq 1$ then t is deletable.*

(b) *If $d(t) = 2$ and $d(t') = 3$ for some t' in $N(t)$, then t is deletable.*

(c) *If $d(t) = 2$ and $d(t') = 2$ for all t' in $N(t)$, then the clique-block of G containing $N(t)$ is G_{10}, G_2, P_j, P'_j or C_j for some j.*

(d) *If $d(t) = 3$, then $H(t)$ is deletable. Suppose $N(t) = \{t, t_1, t_2, t_3\}$. If t_1 shares an edge of t_2, then* cp $(H(t)) = 1$. *Otherwise* cp $(H(t)) = 3$.

(e) *If $d(t) = 4$, then $H(t)$ is deletable and* cp $(H(t)) = 3$.

(f) *If $d(t) = 6$, let $t' \in N(t) - t$. If $d(t') = 6$ then $H(t) \cup H(t')$ is deletable and its clique-partition number is* 1. *Otherwise $H(t)$ is deletable and* cp $(H(t)) = 4$.

Proof. See § 6.

COROLLARY. *If $\Delta(G) \leqq 4$ and G contains no 4-cliques, then every triangle in G, of minimal degree in the triangle graph of G, is deletable.*

4. The algorithms. Suppose, as is customary, that the graph G on n vertices is presented to us as a vertex-incidence table Γ. That is, Γ has n rows and its ith row is headed by the symbol for the ith vertex of G, followed by symbols for the vertices adjacent to it.

We have found it more convenient to use instead, an edge-incidence table of G, that is, a table Γ' having five columns. Its ith row consists of: the symbol for the ith edge of G (in the third column), preceded by the symbols for its ends in columns 1 and 2, the symbols for the edges adjacent at one of its ends in the fourth column, and the symbols for the edges adjacent at the other end in the fifth column. If G was given by a vertex-incidence table Γ initially, then Γ' can be obtained from Γ in $O(n)$ operations if $\Delta(G) \leqq 4$. We describe such a procedure in § 5 for the reader's convenience.

4.1. Algorithm 1. Initially, a subtotal for cp (G) and cc (G) is begun as $\varepsilon(G)$, the number of edges in G. The triangle graph's vertex-incidence table is computed (a procedure for doing this in $O(n)$ operations is described in § 5) and we set $Y = T(G)$. In each pass of the algorithm:

(a) a connected component X of Y is computed,

(b) the block B for which $T(B) = X$ is identified (by means of Table 1 and the second corollary to Lemma 3.2) sufficiently to establish cc (B), cp (B) and $\varepsilon(B)$,

(c) cc (B), cp (B) are added to the subtotals and $\varepsilon(B)$ is subtracted from them both;

(d) Y is replaced by $Y - X$, the graph obtained by deleting all vertices and edges of X from Y.

This sequence of steps is repeated until Y has no vertices left. Thus a total of ω passes is performed, where ω is the number of connected components of $T(G)$, ($\omega = 0$ if $T(G) = \varnothing$). At the ith pass, a clique-block B_i is determined which contains a triangle of G. At the ωth pass the subtotal for cc (G) is $\varepsilon(G) + \sum_{i=1}^{\omega} (\text{cc}(B_i) - \varepsilon(B_i))$ but $\varepsilon(G) - \sum_{i=1}^{\omega} \varepsilon(B_i)$ is the number of clique-blocks which are 2-cliques of G, and hence contribute 1 each to cc (G). Therefore the subtotal on cc (G) showing at the last pass is cc (G). Similarly for cp (G). There is an algorithm (see Aho, Hopcroft and Ullman [1, pp. 176–179]) which computes one by one, all the connected components of a graph on k vertices and m edges in $O(\max(m, k))$ operations. Since the number of vertices of $T(G)$ is at most $\frac{4}{3}n$, and $T(G)$ has at most $3n$ edges (because $\Delta(T(G)) \leqq 6$) it follows that the connected components of $T(G)$ can be computed in linear time. At every pass, we interrupt the Aho, Hopcroft and Ullman algorithm by a constant number of operations after it computes one connected component of a certain graph. It then resumes work on a smaller graph. It follows that Algorithm 1 works in linear time.

ALGORITHM 1 (computes cc (G) and cp (G) if $\Delta(G) \leqq 4$).

Step 1. Let $\text{CC} = \text{CP} = \varepsilon(G)$ and $T = T(G)$.

Step 2. If $T = \varnothing$, then cp $(G) = \text{CP}$, cc $(G) = \text{CC}$, stop. Otherwise, let Y be a connected component of T.

Step 3. Let ν be the number of vertices of Y.
If $1 \leqq \nu \leqq 8$ use Table 2.
If $\nu = 10$ and $\varepsilon = 30$, subtract 29 from each of CC and CP. For all other ν,
if $\varepsilon(Y)$ is odd, subtract $(\nu + 1)$ from CC and $\nu + (\frac{1}{2})(1 - (-1)^{\nu})$ from CP,
if $\varepsilon(Y)$ is even, subtract ν from CC and $\nu - (\frac{1}{2})(1 - (-1)^{\nu})$ from CP.

Step 4. Replace T by $T - Y$ (deleting the edges and vertices of Y). Go to Step 2.

TABLE 2

Condition on Y				Action to be taken	
ν	ε	Δ	δ	subtract from CC	subtract from CP
1				2	2
2				3	2
3				4	[see lines below]
3	2				4
3	3				2
4	3	2		5	4
4	3	3		6	6
4	4			4	4
4	6			5	5
5	<8			6	6
5	8			6	5
6	5			7	6
6	6			8	8
6	7			6	6
6	10			7	5
7	6			8	8
7	7			7	6
7	15			7	5
8			1	9	8
8			>1	8	8

4.2. Algorithms 2, 2′ and 3. Algorithms 2, 2′ and 3 do not require the calculation of $T(G)$ or the use of a special algorithm to compute the connected components of $T(G)$ (clique-blocks of G) as algorithm 1 does.

Algorithms 2 and 2′ have built-in routines which determine, identify and delete all the clique-blocks of G.

Algorithm 2 works this way. After choosing an edge e, it finds the clique-block B containing e, determines cc(B) on the basis of $\varepsilon(B)$ and $\nu(T(B))$, and deletes the edge-set of B from G before beginning the cycle again.

ALGORITHM 2 (computes cc(G) when $\Delta(G) \leqq 4$).

Step 1. Let $C = 0$ and $W = 0$.

Step 2. If G has no edges, then cc$(G) = C$. Stop.

Step 3. Let e be any edge of G.
Let $S(e)$ be the set of triangles sharing the edge e.
Let $\tau(e) = |S(e)|$.
Let F be the set of edges, other than e, belonging to members of $S(e)$.
Add $\tau(e)$ to W.
Label e with $\tau(e)$.

Step 4. If $F = \varnothing$, go to Step 5.
(i) For each edge f of F, add $\tau(f)$ to W, label f with $\tau(f)$.
(ii) Replace F by the set of unlabelled edges belonging to $\bigcup\{S(f) : f \in F\}$.
(iii) Go to Step 4.

Step 5. Let L be the set of labelled edges. [These now are the edges of B, the clique-block containing e.]
Let $\varepsilon = |L|$ and $\nu = W/3$. [ν is the number of vertices in $T(B)$].
If $\varepsilon = 1$ or 6, then add 1 to C.
If $\varepsilon = 7$, then add 3 to C.
If $\varepsilon = 8$ and $\nu = 5$, then add 2 to C.
If $\varepsilon = 8$ and $\nu \neq 5$, then add 4 to C.
If $\varepsilon = 9$ and $\nu = 4$; if $\tau(f) = 2$ for all f in some member of $S(e)$, then add 3 to C, otherwise add 4 to C.
If $\varepsilon = 9$ and $\nu = 7$, then add 2 to C.
If $\varepsilon = 10$ and $\nu = 5$, then add 4 to C.
If $\varepsilon = 10$ and $\nu = 6$, then add 3 to C.
If $\varepsilon = 10$ and $\nu > 6$, then add 1 to C.
If $\varepsilon = 11$, then add 5 to C.
If $\varepsilon = 12$, then add 4 to C.
Otherwise, add ν to C.
Step 6. Remove L from the edge-set of G and go to Step 2.

Tables 3a, b derived from Table 1 and Corollary 2 to Lemma 3.2, explain the instructions in Step 5 of Algorithms 2 and 2′.

TABLE 3a

B	K_2	K_3	P_2	G_6	P_3	G_{10}	C_4	G_7	P_4	G_1	G_9	G_2	G_8	K_5	P_5	G_4	G_3	G_5
$\varepsilon(B)$	1	3	5	6	7	7	8	8	9	9	9	10	10	10	11	11	12	12
$\nu(T(B))$	0	1	2	4	3	3	4	5	4	4	7	5	6	10	5	6	6	8
cc (B)	1	1	2	1	3	3	4	2	4	3	2	4	3	1	5	5	4	4
cp (B)	1	1	3	1	3	5	4	3	5	3	4	4	5	1	5	5	4	4

TABLE 3b

B	P_ν, P'_ν		C_ν	
	odd $\nu \geqq 7$	even $\nu \geqq 6$	odd $\nu \geqq 7$	even $\nu \geqq 8$
$\varepsilon(B)$	$2\nu+1$	$2\nu+1$	2ν	2ν
$\nu(T(B))$	ν	ν	ν	ν
cc (B)	ν	ν	ν	ν
cp (B)	ν	$\nu+1$	$\nu+1$	ν

Algorithm 2′ is a modification of Algorithm 2 that computes cp (G).

ALGORITHM 2′ (finds cp (G) when $\Delta(G) \leqq 4$).
Steps 1, 2, 3, 4, 6 are the same as in Algorithm 2, except "cc (G)" is replaced by "cp (G)".
Step 5. Let L be the set of labelled edges, $\varepsilon = |L|$ and $\nu = W/3$.
If $\varepsilon = 1$, 3 or 6, then add 1 to C.
If $\varepsilon = 5$, then add 3 to C.
If $\varepsilon = 7$ and $\tau(f) = 3$ for some f in L, then add 5 to C.
If $\varepsilon = 7$ and $\tau(f) \neq 3$ for all f in L, then add 3 to C.
If $\varepsilon = 8$ and $\nu = 5$, then add 3 to C.

If $\varepsilon = 8$ and $\nu \neq 5$, then add 4 to C.
If $\varepsilon = 9$ and $\nu = 4$, if $\tau(f) = 2$ for all f in some member of $S(e)$, then add 3 to C, otherwise add 5 to C.
If $\varepsilon = 9$ and $\nu \neq 4$, then add 4 to C.
If $\varepsilon = 10$ and $\nu = 5$, then add 4 to C.
If $\varepsilon = 10$ and $\nu = 6$, then add 5 to C.
If $\varepsilon = 10$ and $\nu > 6$, then add 1 to C.
If $\varepsilon = 11$, then add 5 to C.
If $\varepsilon > 12$, add $\nu + 1$ to C if $\varepsilon + \nu$ is odd, add ν to C if $\varepsilon + \nu$ is even.

We now present Algorithm 3. In each pass of this algorithm, we detect and remove a deletable object from G (a vertex or nonempty set of edges), while adding its clique-partition number to a running subtotal. When the edge-set of G is exhausted, the subtotal is cp (G).

We cite the lemmas, which justify the deletion of the object and the statement of its clique-partition number, in brackets following the corresponding step or substep in the algorithm.

ALGORITHM 3 (computes cp (G) when $\Delta(G) \leqq 4$).
Step 1. Let $C = 0$.
Step 2. If G has no edges, $C = $ cp (G). Stop.
Step 3. Let e be any edge of G.
Let $S(e)$ be the set of triangles in G sharing the edge e. If S is empty, replace G by $G - e$, add 1 to C and go to Step 2 [Lemma 2.8(b)].
Step 4. Choose any triangle t in S. Say its edges are e, f, g.
Let $N(t)$ be the family of triangles sharing edges of t, that is, $N(t) = S(e) \cup S(f) \cup S(g)$.
Let $d(t) = |N(t)| - 1$, the number of triangles, other than t, sharing edges with t. [So $d(t)$ is the degree of t as a vertex of $T(G)$.]
If $d(t) \leqq 1$, add 1 to C, replace G by $\bar{t}(G)$ and go to Step 2 [Lemma 3.3(a)].
If $d(t) = 2$, then $N(t) = \{t, t_1, t_2\}$.
If $d(t_i) = 1$ or 3 for $i = 1$ or 2, add 1 to C, replace G by $\bar{t}_i(G)$ if $d(t_i) = 1$, by $\bar{t}(G)$ otherwise and go to Step 2 [Lemma 3.3(a) or (b)].
If $d(t_1) = 4$, add 3 to C, replace G by $\overline{N(t_1)}(G)$ and go to Step 2 [Lemma 3.3(e)].
If $d(t_1) = d(t_2) = 2$, go to Step 5.
If $d(t) = 3$, then $N(t) = \{t, t_1, t_2, t_3\}$.
If $d(t_i) > 3$ for some i, add 3 to C, replace G by $\overline{N(t_i)}(G)$ and go to Step 2 [Lemma 3.3(e)].
If $d(t_i) = 3$ for all i, add 1 to C if $t_2 \in N(t_1)$; add 3 to C otherwise. Replace G by $\overline{N(t)}(G)$ and go to Step 2 [Lemma 3.3(d)].
If $d(t_i) < 3$ for some i, add 1 to C, replace G by $\bar{t}_i(G)$ and go to Step 2 [Lemma 3.3(a) or (b)].
If $d(t) = 4$, add 3 to C, replace G by $\overline{N(t)}(G)$ and go to Step 2 [Lemma 3.3(e)].
If $d(t) = 6$, then $N(t) = \{t, t_1, \cdots, t_6\}$.
If $d(t_1) < 6$, add 4 to C, replace G by $\overline{N(t)}(G)$ and go to Step 2.
If $d(t_1) \geqq 6$, add 1 to C, replace G by $\overline{N(t) \cup N(t_1)}(G)$ and go to Step 2 [Lemma 3.3(f)].
Step 5. $N(t) = \{t, t_1, t_2\}$. [By Lemma 3.3(c), t is a subgraph of: G_2, G_{10}, P_j (or P_j') or C_j; now we distinguish between these cases.] Let $u_1 = t$ and $u_2 = t_1$.

For all $i \geqq 2$, if $|N(u_i)| > 2$ and $u_i \neq t_2$, let u_{i+1} be an element of $N(u_i) - \{u_i, u_{i-1}\}$. If $|N(u_{i+1})| = 4$, then let $F = N(u_1) \cup N(u_3)$, replace G by $\bar{F}(G)$, add 4 to C and go to Step 2. [In this case we detected that t was a subgraph of G_2 which is the subgraph of G generated by F.]
Otherwise, let $k = i$ and $F = \bigcup_{i=1}^{k} u_i$.
If $|N(u_k)| = 2$, go to Step 6. Otherwise [t is a subgraph of G_{10} or C_k], replace G by $\bar{F}(G)$. Add 5 to C if $k = 3$, add $k + (\frac{1}{2})(1-(-1)^k)$ to C if $k \neq 3$, go to Step 2 [Corollary 2 to Lemma 3.2].

Step 6. Let $s_1 = t_2$ and $s_0 = t$. [Now we know that t belongs to P_j or P_j' and that u_k is one end of the path $T(P_j)$ (or $T(P_j')$). We seek the other end.]
For all $i \geqq 1$, if $|N(s_i)| > 2$, let s_{i+1} be an element of $N(s_i) - \{s_i, s_{i-1}\}$.
Otherwise, let $l = i$. Add $k + l + 1$ to C, let $H = F \cup \bigcup_{i=0}^{l} s_i$, replace G by $\bar{H}(G)$ and go to Step 2. [Corollary 2 to Lemma 3.2].

At every pass, Algorithms 2, 2′ and 3 delete γ edges at the cost of τ operations. A step-by-step examination shows that $\tau \leqq c\gamma$ for some constant c independent of the number n of vertices of G. Therefore they complete their tasks in at most $2cn$ operations, as G has at most $(n/2)\Delta(G)$ edges. Here are three points to keep in mind while carrying out the examination.

1. The calculation of $S(e)$ takes at most 9 comparisons, so the calculation of $N(t)$ takes at most 27 comparisons.

2. In Steps 5 and 6, Algorithm 1 calculates $N(t)$ for μ values of t and deletes at least 2μ edges. The total cost per execution of those steps is at most 27μ plus at most some constant multiple of μ other operations.

3. Deleting an edge requires fewer than 50 operations as it needs the deletion of one symbol from at most 6 rows of Γ', followed by the deletion of at most 6 symbols from one row. This requires a total of at most 12 deletions and 36 comparisons.

4.3. Modifications and extensions—Algorithm 1′. It should be pointed out that Algorithms 1 and 2′ can be easily modified to produce minimal clique-partitions, as can Algorithms 1 and 2 to produce minimal clique-coverings of G when $\Delta(G) \leqq 4$.

On the other hand we do not know how to extend the algorithms to deal with the computation of clique-coverings, partitions, cc (G) or cp (G) when $\Delta(G) > 4$. This is due in part, to our inability to find an analogue of Theorem 1 for such graphs.[2]

Another difficulty is, that while cliques of maximum order in G are always part of some minimal clique-covering (and some minimal clique-partition) when $\Delta(G) \leqq 4$, the same is not true when $\Delta(G) > 4$. For example, the six triangles with vertices at A, B, C, D, E, F in the graph M of Fig. 3 form a minimal clique-covering (and -partition), but the 4-clique $UVWX$ belongs to no minimal clique-covering (or -partition) even though it is a clique of maximum order in M.

Even if G is known to have no cliques of order exceeding three, our methods appear to us to have no obvious extension which copes with graphs of maximum degree exceeding four. For example, when $\Delta(G) \leqq 4$ and G is 4-clique-free then the corollary to Lemma 3.3 suggests the following simple procedure (which works in linear time):

(a) find a vertex t of minimal degree in $T(G)$,
(b) tally t,

[2] *Note added in proof.* D. N. Hoover has shown that when $\Delta(G) \geqq 5$, the problem of computing cp (G) in NP-complete. (Private communication.)

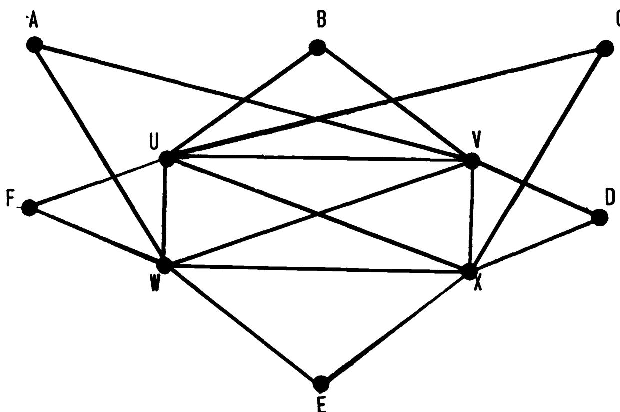


FIG. 3

(c) replace G by $\bar{t}(G)$

and repeat (a), (b), (c) until the graph is triangle-free, then cp (G) is the sum of the tallies and the number of edges left in the final graph.

We can write a version of this procedure in a more formal manner.

ALGORITHM 1′ (computes cp (G) when $\Delta(G) \leqq 4$ and G has no 4-cliques).

Step 1. Let $T = T(G)$ and $C = \varepsilon(G)$.

Step 2. If $T = \varnothing$, then cp $(G) = C$. Stop.

Step 3. Let t be a vertex of T of minimal degree. [t is deletable, by the Corollary to Lemma 3.3.]

Subtract 2 from C. [t contributes 1 to cp (G) and 3 to $\varepsilon(G)$.]

Replace T by $T - N(t)$. [The graph $T - N(t)$, obtained by deleting the vertices of the neighborhood in T of t (along with every edge incident with them) from T, is $T(\bar{t}(G))$.] Go to Step 2.

Unfortunately, the conclusion of the corollary to Lemma 3.3 may be false if we relax the hypothesis "$\Delta(G) \leqq 4$". For example, the graph D of Fig. 4 (supplied by D. de Caen) has maximum degree 5 and no 4-cliques. The triangle marked "t" is not deletable (even though its degree in $T(D)$ is minimal) because it is not part of a maximum independent set of vertices in $T(D)$ as required by Lemma 3.2.

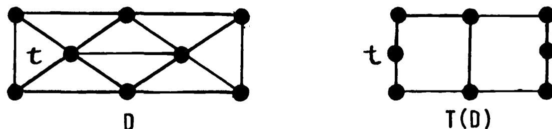


FIG. 4

5. Appendix 1. Algorithms for the edge incidence table of G and the vertex incidence table of $T(G)$.

5.1. Construction of the edge-incidence table. We assume that G is given by a vertex-incidence table Γ. The following procedure produces an edge-incidence table Γ' from Γ and works in $O(n)$ operations when G has n vertices and $\Delta(G) \leqq 4$.

Suppose the symbols for the vertices of G are the integers from 1 to n, and the ith row of Γ, Γ_i begins with i, followed by the vertices j adjacent to i, if any.

Step 1. Let $C = 0$ and $i = 1$. Let Λ be a table, initially empty, which will be 3 symbols wide and $\varepsilon(G)$ rows long. Let ϕ be a table, n rows long, whose

ith row ϕ_i, will be at most $1+\Delta(G)$ symbols wide. Initially $\phi_i = i$ for $1 \leqq i \leqq n$.

Step 2. Suppose $\Gamma_i = [i, v_i^1, v_i^2, \cdots, v_i^d]$ and $d = 0$ if $\Gamma_i = [i]$.

Step 3. If $d = 0$, replace i by $i+1$ and go to Step 2.
Otherwise for $j = 1, 2, \cdots, d$: if $v_i^j > i$ replace C by $C+1$, let $\Lambda_C = [i, v_i^j, C]$ and replace ϕ_i by ϕ_i, C.

Step 4. If $i < n$, replace i by $i+1$ and go to Step 2.

Step 5. [Now we have numbered the edges of G from 1 to ε and Λ has become a table whose Cth row lists v, w, C where v, w are the ends of the Cth edge. In that sense, Λ is an edge-encoding table. Also, $\phi_i = [i, C_1^i, C_2^i, \cdots]$ where $\{C_1^i, C_2^i, \cdots\}$ is the set of all deg (i) edges incident with the vertex i.] We form Γ' as follows.
For $C = 1, 2, \cdots, \varepsilon$:
if $\Lambda_C = [v, w, C]$ then $\phi_v = [v, C_1^v, C_2^v, \cdots, C_{\deg(v)}^v]$
and $\phi_w = [w, C_1^w, C_2^w, \cdots, C_{\deg(w)}^w]$.
Let $\Gamma'_C = [\Lambda_C | \phi_v | \phi_w]$.

Example. The graph G given in Fig. 5 has tables Γ, Λ, ϕ and Γ' given in Tables 4a, b, c, d.

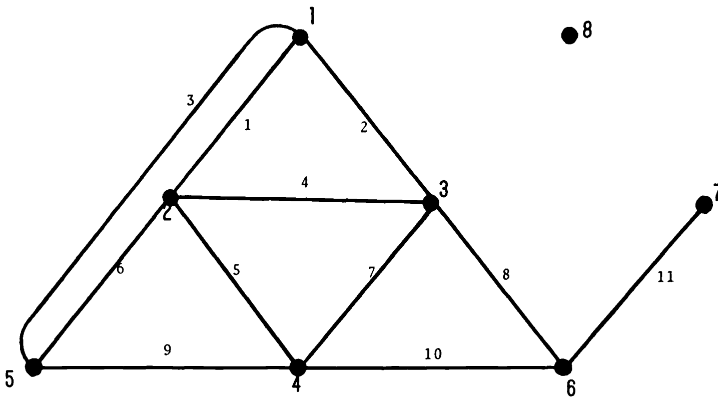


FIG. 5

TABLE 4a

Vertex incidence table Γ				
vertex	adjacent vertices			
1	2	3	5	
2	1	3	4	5
3	1	2	4	6
4	2	3	5	6
5	1	2	4	
6	3	4	7	
7	6			
8				

TABLE 4b

Edge-encoding table Λ

end 1	end 2	edges
1	2	1
1	3	2
1	5	3
2	3	4
2	4	5
2	5	6
3	4	7
3	6	8
4	5	9
4	6	10
6	7	11

TABLE 4c

Vertex-edge-incidence table ϕ

vertex	incident edges			
1	1	2	3	
2	1	4	5	6
3	2	4	7	8
4	5	7	9	10
5	3	6	9	
6	8	10	11	
7	11			
8				

TABLE 4d

Edge-incidence table Γ'

end 1	end 2	edge	edges at end 1			edges at end 2		
1	2	1	2	3		4	5	6
1	3	2	1	3		4	7	8
1	5	3	1	2		6	9	
2	3	4	1	5	6	2	7	8
2	4	5	1	4	6	7	9	10
2	5	6	1	4	5	3	9	
3	4	7	2	4	8	5	9	10
3	6	8	2	4	7	10	11	
4	5	9	5	7	10	3	6	
4	6	10	5	7	9	8	11	
6	7	11	8	10				

5.2. Construction of the vertex-incidence table L'' for the triangle graph $T(G)$.
Step 1. Let $i=0$, $C=0$, and let $L_j^2=\varnothing$ for $1 \leqq j \leqq \varepsilon(G)$.
Step 2. Replace i by $i+1$. If $i>\varepsilon$, go to Step 6.

Step 3. For the ith edge e_i, use Γ' to determine the τ triangles containing e_i. List them as
$N(e_i) = [e_i f^1 g^1, e_i f^2 g^2, \cdots, e_i f^\tau g^\tau]$ with $N(e_i) = \varnothing$ if $\tau = 0$.
Let $j = 0$.

Step 4. If $j = \tau$, go to step 2. Otherwise replace j by $j+1$.

Step 5. If $f^j > e_i$ and $g^j > e_i$ let $C = C+1$ and $L_C^* = e_i f^j g^j$, go to step 4. Otherwise, go to Step 4.

Step 6. For $1 \leqq k \leqq C$: replace L_j^2 by L_j^2, k for every j in L_k^*.

Step 7. If $L_k^* = [e\ f\ g]$, let $L_k'' = [L_e^2 - e, L_f^2 - e, L_g^2 - e]$.

The following traces the construction of L'' for the graph G of Fig. 3. It uses the table Γ' found in § 5.1.

Step 3. $N(1) = [1\ 2\ 4 | 1\ 3\ 6]$ as $\text{ends}(2) \cap \text{ends}(4) = 3$, $\text{ends}(2) \cap \text{ends}(j) = \varnothing$ $(j = 5, 6)$, $\text{ends}(3) \cap \text{ends}(6) = 5$, and $\text{ends}(3) \cap \text{ends}(j) = \varnothing$ $(j = 4, 5)$, from Table Γ'.

Step 5. $L_1^* = [1\ 2\ 4]$
$L_2^* = [1\ 3\ 6]$

Step 3. $N(2) = [2\ 1\ 4]$

Step 3. $N(3) = [3\ 1\ 6]$

Step 3. $N(4) = [4\ 1\ 2 | 4\ 5\ 7]$

Step 5. $L_3^* = [4\ 5\ 7]$

Step 3. $N(5) = [5\ 6\ 9 | 5\ 4\ 7]$

Step 5. $L_4^* = [5\ 6\ 9]$

Step 3. $N(6) = [6\ 1\ 3 | 6\ 5\ 9]$

Step 3. $N(7) = [7\ 4\ 5 | 7\ 8\ 10]$

Step 5. $L_5^* = [7\ 8\ 10]$

TABLE 5a
L_i^* lists edges in ith triangle

L^*(Steps 1–5)		
1	2	4
1	3	6
4	5	7
5	6	9
7	8	10

Step 6. $(k=1)$ $L_1^2 = [1]$, $L_2^2 = [1]$, $L_4^2 = [1]$
$(k=2)$ $L_1^2 = [1\ 2]$, $L_3^2 = [2]$, $L_6^2 = [2]$
$(k=3)$ $L_4^2 = [1\ 3]$, $L_5^2 = [3]$, $L_7^2 = [3]$
$(k=4)$ $L_5^2 = [3\ 4]$, $L_6^2 = [2\ 4]$, $L_9^2 = [4]$
$(k=5)$ $L_7^2 = [3\ 5]$, $L_8^2 = [5]$, $L_{10}^2 = [5]$

TABLE 5b
L_i^2 lists triangles on ith edge

L^2(Step 6)	
1	2
1	
2	
1	3
3	4
2	4
3	5
5	
4	
5	

Step 7. $L_1^* = [1\ 2\ 4]$
$L_1'' = [L_1^2 - 1, L_2^2 - 1, L_4^2 - 1] = [2\ 3]$
$L_2^* = [1\ 3\ 6]$
$L_2'' = [L_1^2 - 2, L_3^2 - 2, L_6^2 - 2] = [1\ 4]$

$L_3^* = [4\ 5\ 7]$
$L_3'' = [L_4^2 - 3, L_5^2 - 3, L_7^2 - 3] = [1\ 4\ 5]$
$L_4^* = [5\ 6\ 9]$
$L_4'' = [L_5^2 - 4, L_6^2 - 4, L_9^2 - 4] = [3\ 2]$
$L_5^* = [7\ 8\ 10]$
$L_5'' = [L_7^2 - 5, L_8^2 - 5, L_{10}^2 - 5] = [3]$

TABLE 5c
Vertex-incidence table for $T(G)$

	L''		
1	2	3	
2	1	4	
3	1	4	5
4	3	2	
5	3		

6. Appendix 2. Proofs.

Proof of Lemma 2.5. There is at most one vertex and one edge externally adjacent to K, per vertex of K, because K is a $\Delta(G)$-clique. Therefore $\nu'(K) \leqq k$ and $\varepsilon'(K) \leqq k$. For each vertex v externally adjacent to K, let $C(v)$ be the clique whose vertices consist of v and those vertices of K adjacent to v. If v' is another vertex externally adjacent to K, then $C(v')$ has no vertices in common with $C(v)$, as there is at most one edge externally adjacent to K per vertex of K. Therefore $C(v')$ and $C(v)$ are in different cliques of any covering $\mathscr{C}$ of $N(K)$. Moreover, K and $C(v)$ are in different members of $\mathscr{C}$ because $N(K)$ is not a $(k+1)$-clique. Therefore $|\mathscr{C}| \geqq 1 + \nu'(K)$, but K and the family of all $C(v)$ with v externally adjacent to K form a clique-covering of $N(K)$ having $1 + \nu'(K)$ members. Therefore $\mathrm{cc}\,(N(K)) = 1 + \nu'(K)$.

It was shown in [8, Lemma 3.3], that $\mathrm{cp}\,(N(K)) = 1 + \varepsilon'(K)$ when $\varepsilon'(K) = k$. Suppose $\varepsilon'(K) < k$, then K and its $\varepsilon'(K)$ externally adjacent edges form a clique-partition of $N(K)$. Therefore $\mathrm{cp}\,(N(K)) \leqq 1 + \varepsilon'(K)$. By adjoining $k - \varepsilon'(K)$ edges, one to each of the $k - \varepsilon'(K)$ vertices of K of degree $k-1$, and augmenting a minimal clique-partition $\mathscr{C}$ of $N(K)$ by these edges, we can form a clique-partition $\mathscr{C}'$ of a new neighborhood $N'(K)$ of K, with k edges externally adjacent to K. Therefore, $|\mathscr{C}'| = |\mathscr{C}| + k - \varepsilon'(K)$. If $|\mathscr{C}| < 1 + \varepsilon'(K)$, then we would have $|\mathscr{C}'| < 1 + k$ contradicting [8, Lemma 3.3].

Proof of Lemma 2.7. Let B and H be clique-blocks in G. Suppose the edge-sets $E(B)$ and $E(H)$ of B and H are not disjoint. Let t be any triangle in G. If t contains an edge of $B \cap H$ then $t \subseteq B$ and $t \subseteq H$, because B and H separate the cliques of G. Therefore $t \subseteq B \cap H$ if (and only if) t contains an edge of $B \cap H$. If t does not contain an edge of $B \cap H$, then all three edges of t are in B and none are in H, or all are in H and none are in B, or none are in $B \cup H$ (as B and H separate cliques). In any case, $t \subseteq \overline{B \cap H}$. Thus any triangle in G is either wholly in $B \cap H$ or wholly in $\overline{B \cap H}$. Therefore, by Lemma 2.1(c), $B \cap H$ separates the cliques of G, and hence $B = H$.

Proof of Lemma 3.1. Suppose that the triangle graph T of G has more than one connected component and T_1 is one of them. There is some subgraph H of G for which $T(H) = T_1$. Every triangle of G lies in H or $\bar{H}$, because every vertex of T lies in T_1 or some other connected component of T. Therefore, by Lemma 2.1(c), H

separates the cliques of G. The subgraph H must be proper, because $T_1 \neq T$. Therefore G is clique-separable. Conversely, if G is clique-separable, then it has more than one clique-block. Let H_1 be one of them and H_2 another. If an edge of $T(G)$ had one end in $T(H_1)$ and the other in $T(H_2)$, then a triangle in H_1 would share an edge with a triangle in H_2. This contradicts Lemma 2.7 which implies that the edge-sets of H_1 and H_2 are disjoint. Therefore the triangle graph of G is disconnected.

Proof of Lemma 3.2. Let $\mathscr{C}_0$ be a fixed minimal clique-partition of G and $\mathscr{C}$ be an arbitrary clique-partition. We have $|\mathscr{T}(\mathscr{C}_0)| \geqq |\mathscr{T}(\mathscr{C})|$ with equality if and only if $\mathscr{C}$ is minimal because G is 4-clique-free. Moreover, $\mathscr{T}(\mathscr{C})$ is independent because $\mathscr{C}$ partitions the edges of G. If $\mathscr{T}_0$ is a maximum independent set of vertices of $T(G)$, then in particular, $|\mathscr{T}_0| \geqq |\mathscr{T}(\mathscr{C}_0)|$. Augment $\mathscr{T}_0$ by those edges of G which are in none of the triangles in $\mathscr{T}_0$, to form a clique-partition $\mathscr{C}_1$ of G with $\mathscr{T}(\mathscr{C}_1) = \mathscr{T}_0$. Thus $|\mathscr{T}(\mathscr{C}_1)| = |\mathscr{T}(\mathscr{C}_0)|$ and hence $\mathscr{C}_1$ is minimal and, as $|\mathscr{T}(\mathscr{C}_1)| = |\mathscr{T}_0|$, $\mathscr{T}(\mathscr{C}_1)$ is a maximum independent set. If $\mathscr{C}$ is a minimal clique-partition of G, then $|\mathscr{T}(\mathscr{C})| = |\mathscr{T}(\mathscr{C}_0)|$, and hence $\mathscr{T}(\mathscr{C})$ is maximum.

Proof of Theorem 1. If G is triangle-free, then G has 0 or 1 edges by Lemma 2.6(a). Therefore G is empty (i.e., edgeless) or $G = K_2$.

Therefore, we may suppose that $T(G) \neq \emptyset$. According to Lemma 3.1, $T(G)$ is connected.

Suppose first that G contains a 4-clique, H. By Lemma 2.4, $N(H)$ separates the cliques of the clique-inseparable graph G. Therefore $G = N(H)$. According to Lemma 2.5, H has ε' externally adjacent edges (at most one per vertex of H). Inseparability implies that $\varepsilon' \neq 1$. Therefore the number of edges of G, $\varepsilon(G) = 6, 8, 9$ or 10. If $\varepsilon(G) = 6$, 8 or 9, then $G = G_6, G_7$ or G_9 respectively. If $\varepsilon(G) = 10$ then $G = G_8$ or K_5 accordingly as G has 6 or 5 vertices respectively.

We may now assume that G is 4-clique-free. Suppose that $T(G)$ contains a triangle. Call its vertices x, y, and z. These are triangles in G. Consider the intersection of their edge-sets in G. If it were empty, then as each of their pairwise edge intersections is nonempty, their union would form a 4-clique. Therefore the triangles x, y, z share an edge and hence G contains G_{10}. Since G is 4-clique-free and clique-inseparable it follows that $G = G_{10}$.

Now we may assume that $T(G)$ is triangle-free, as well as 4-clique-free. Therefore $\Delta(T(G)) \leqq 3$, because if some triangle in G shared its edges with four other triangles, then one of its edges would be shared by two of the other triangles; G would contain G_{10} and hence $T(G)$ would contain $T(G_{10}) = K_3$.

We distinguish four cases corresponding to the possible values of $\Delta(T(G))$.

Case 1. $\Delta(T(G)) = 0$. In this case, $T(G) = K_1$. Therefore G contains exactly one triangle. By Lemma 2.6(b), it separates the cliques of G. The clique-inseparable graph G must then be that triangle. Therefore $G = P_1$.

Case 2. $\Delta(T(G)) = 1$. In this case $T(G) = K_2$, as $T(G)$ is connected. Therefore G contains exactly two triangles and these share one edge. Therefore G contains P_2, but G is clique-inseparable and contains no triangles other than the two comprising P_2. Consequently $G = P_2$.

Case 3. $\Delta(T(G)) = 2$. Either $T(G)$ is a simple path or simple cycle in this case, because $T(G)$ is connected. It can be shown, by a careful examination of cases when $j \leqq 5$ and by induction for $j > 5$ that if $T(G)$ is a simple path of length $j-1$, then $G = P_j$ if $1 \leqq j \leqq 5$ and $G = P_j$ or P'_j if $j > 5$. Similarly, by a careful examination of cases when $j \leqq 6$ and by induction when $j > 6$, it can be shown that if $T(G)$ is a simple cycle of length j, then $G = G_{10}$ if $j = 3$ and $G = C_j$ for $j = 4$ and $j > 6$. There is no G for which $T(G)$ is a 5-cycle or a 6-cycle.

Case 4. $\Delta(T(G))=3$. In this case, $T(G)$ contains the subgraph T_1 and G contains the subgraph G_1 of Fig. 2. We distinguish four subcases corresponding to the number of edges determined by the vertices A, B, C of degree 2 of G_1.

Subcase 4.0. None of AB, BC, CA are edges of G. In this case, suppose a triangle XYZ of G had an edge XY in G_1. One of X, Y has degree 4 in G_1, therefore Z is in G_1 (because $\Delta(G) \leqq 4$). Therefore the arbitrary triangle XYZ of G is in G_1 and hence $G=G_1$ by the clique-inseparability of G. We also have $T(G)=T_1$.

Subcase 4.1. Exactly one pair of the vertices $\{A, B, C\}$ of degree 2 of G_1 are adjacent in G. Suppose we label this pair AB. In this case G_2 is a subgraph of G. If G_2 separates G's cliques then $G=G_2$ and $T(G)=T_2$. Otherwise some triangle XYZ in G has an edge XY in G_2 and an edge XZ not in G_2. Both X and Y must be in $\{A, B, C\}$ as $\Delta(G) \leqq 4$. Therefore $XY=AB$ and hence G_3 is a subgraph of G. This subgraph separates the cliques of G. To show that, let UVW be any triangle in G with edge UV in G_3. At least one of the vertices U, V has degree 4 in G_3. Therefore each edge of UVW is in G_3 and hence G_3 separates the cliques of G. By inseparability, $G_3=G$ so $T(G)=T_3$.

Subcase 4.2. Exactly two pairs of the vertices $\{A, B, C\}$ of G_1 are adjacent in G. Suppose these are AB and BC. In this case G_4 is a subgraph of G. If UVW is any triangle in G with an edge UV in G_4 then at least one end of UV has degree 4 in G_4 and hence $UVW \subseteq G_4$. Therefore G_4 separates the cliques of G, but G is clique-inseparable so $G=G_4$ and $T(G)=T_4$.

Subcase 4.3. The vertices of degree 2 in G_1 are in a triangle in G. In this case, G_5 is a 4-regular subgraph and hence separates the cliques of the clique-inseparable graph G. Therefore $G=G_5$ and $T(G)=T_5$. This concludes the proof of Theorem 1.

Proof of Lemma 3.3. (a) If $d(t)=0$, then t is a clique-block by Lemma 2.6(b). If $d(t)=1$, let L denote the clique-block of G containing t. By Figs. 1 and 2, $L=P_j$ or P_j' for some $j>1$, or $L=G_i$ for some $1 \leqq i \leqq 3$. In each case, t is part of a maximal independent set of vertices of $T(L)$. Therefore, by Lemmas 2.8(f) and 3.2, t is deletable from G.

(b) According to Fig. 2 $H(t)=G_i$ for $i=2, 3$ or 4. In each case, t is part of a maximal independent subset of $T(H(t))$. Therefore, by Lemmas 3.2 and 2.8(f), t is deletable.

(c) The connected component of t in $T(G)$ is T_2 or a path or cycle by Figs. 1 and 2.

(d) According to Fig. 2, $H(t) \subseteq G_i$ for some $1 \leqq i \leqq 7$. If $H(t) \subseteq G_i$, then $\{t_1, t_2, t_3\}$ is independent in $T(G)$ if $1 \leqq i \leqq 5$, and $N(t)$ is a 4-clique in $T(G)$ if $i=6$ or 7. If t_1 is adjacent to t_2 in $T(G)$, then $H(t)=G_6$ and $\operatorname{cp}(G_6)=1$. Otherwise $\{t_1, t_2, t_3\}$ is part of a clique-partition of G_i ($i \leqq 5$) by Lemma 3.2, therefore $t_1 \cup t_2 \cup t_3 = H(t)$ is deletable from G_i and hence from G by Lemma 2.8(f). Moreover $H(t)=G_1$, so $\operatorname{cp}(H(t))=3$ by Table 1.

(e) According to Fig. 2 $H(t)=G_7$ because neither T_8 nor T_9 contain vertices of degree 4 adjacent to all other vertices of degree 4. According to Table 1, $\operatorname{cp}(G_7)=3$.

(f) First note that every triangle in K_5 shares an edge with six other triangles. Therefore the triangle graph of K_5 is 6-regular. Theorem 1 and the definitions of P_j, P_j' and C_j imply that K_5 is the only clique-inseparable graph whose triangle graph is 6-regular. By Figs. 1 and 2, G_9 is the only clique-inseparable graph whose triangle-graph has exactly one vertex of degree 6. Both graphs are deletable by Lemma 2.8.

Acknowledgments. Thanks to Selim Akl, Dominique de Caen and David A. Gregory for many helpful conversations, and to the referees for their diligence.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Don Mills, Ontario, Canada, 1974.

[2] J. A. BONDY AND U. S. R. MURTY, *Graph Theory with Applications*, Macmillan, London, 1977.

[3] A. DONALD, *Edge and arc partitions of arbitrary graphs and digraphs*, M.Sc. thesis, Queen's University, Kingston, Ontario, Canada, 1979.

[4] P. ERDÖS, A. W. GOODMAN AND L. PÓSA, *The representation of a graph by set intersections*, Can. J. Math., 18 (1966), pp. 106–112.

[5] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1968.

[6] L. LOVÁSZ, *On covering of graphs*, Theory of Graphs, Proc. Colloquium held at Tihany, Hungary, September 1966, P. Erdös and G. Katona, eds., Academic Press, New York, 1968, pp. 231–236.

[7] J. ORLIN, *Contentment in graph theory: covering graphs with cliques*, K. Nederlandse Ak. van Wetenschappen Proc. Ser. A, 80 (1977), pp. 406–424.

[8] N. J. PULLMAN AND D. DE CAEN, *Clique coverings of graphs* I: *clique-partitions of regular graphs* Utilitas Math., 19 (1981), pp. 177–206.

[9] N. J. PULLMAN AND A. W. DONALD, *Clique coverings of graphs* II: *complements of complete graphs*, Utilitas Math., 19 (1981), pp. 207–214.

[10] N. J. PULLMAN AND D. DE CAEN, *Clique coverings of graphs* III: *clique-coverings of regular graphs*, Congressus Numerantium, 29 (1980), pp. 795–808.

[11] H. RYSER, *Intersection properties of finite sets*, J. Combin. Theory A, 14 (1973), pp. 79–92.

[12] ———, *Intersection properties of finite sets*, Colloquio Internazionale sulle Teorie Combinatorie, Roma 3–5 settembre 1973, Tomo II, Accademia Nazionale dei Lincei Rome, 1976, pp. 328–334.

SIAM J. COMPUT.
Vol. 13, No. 1, February 1984

© 1984 Society for Industrial and Applied Mathematics
0097-5397/84/1301-0006 $01.25/0

FORMAL SYSTEMS FOR TUPLE AND EQUALITY GENERATING DEPENDENCIES*

C. BEERI† AND M. Y. VARDI†‡

Abstract. We develop several formal systems for tuple and equality generating dependencies. There are three kinds of systems, based upon substitution, tuple elimination and transitivity. We specialize our systems to several subclasses: total dependencies, template dependencies and binary dependencies. We also show that finding a formal system for embedded multivalued dependencies is equivalent to solving the implication problem for that class.

Key words. database, dependency, implication, formal system

1. Introduction. One of the important issues in the design of relational database schemes is the specification of the constraints that the data must satisfy to model correctly the part of the world under consideration.

Of particular interest are the constraints called *data dependencies*. The first dependencies to be studied were the *functional dependencies* [Codd], which were followed by the *multivalued dependencies* [Del], [Fag1], [Zan]. Later, several types of dependencies were investigated in the literature. Recently, several researchers have independently proposed a new type of dependencies, *tuple* and *equality generating dependencies*, which generalizes all previously studied types [BV2], [Fag2], [YP].[1] Intuitively, the meaning of such a dependency is that if some tuples, satisfying certain equalities, exist in the database, then some other tuples must also exist in the database, or some values in the given tuples must be equal. We assume that the database is many-sorted, i.e., different attributes have different underlying domains.

The *implication problem* for dependencies is to decide whether a given dependency is logically implied by a given set of dependencies. This problem is recursively unsolvable in general [BV2], [CLM], and is solvable but computationally intractable if all tuple generating dependencies are total [BV2], [BV4], [CLM]. A proof procedure for implication of dependencies, called "chase", was studied in [BV3] generalizing [ABU], [MMS] (similar procedures were studied in [Pa], [SU1], [YP]).

The chase enables us to semi-decide implication. In contrast, a *formal system* for dependencies enables us to *derive* new dependencies from the given ones. The interest in formal systems has many aspects. First, a formal system provides us with tools to operate on dependencies, e.g., for equivalence preserving transformations [Ma]. Secondly, a formal system may lead us to discover efficient decision procedures to the implication problem. For example, the formal system for functional dependencies of [Arm] has led to a linear time decision procedure for this class [BB], and the formal system for deriving multivalued dependencies from functional and join dependencies has led to a quadratic time decision procedure for this case [Va2]. Finally, a formal system helps us to gain insight into the dependencies. This insight can lead to useful application, e.g., synthesis of database schemes from functional dependencies [Be]. Formal systems for dependencies have attracted a lot of attention in the last few years, see for example [Arm], [BFH], [BV1], [PJ], [Sc], [SU1], [SU2], [Va1], [YP].

* Received by the editors May 7, 1981, and in revised form November 5, 1982.

† Department of Computer Science, The Hebrew University of Jerusalem, Jerusalem 91904, Israel.

‡ The research of this author was partially supported by grant 1849/79 of the U.S.A.-Israel Binational Science Foundation. Current address: IBM Research Laboratory, San Jose, California 95193.

[1] We use here the terminology of [BV2], which is different from those of [Fag2], [YP].

It has been demonstrated that dependencies are actually equivalent to sentences in first-order logic [Nic], and in fact the chase is a variant of a well-known theorem-proving procedure: refutation by resolution and paramodulation [BV3]. Thus, it might be argued, there is no need to develop formal systems for dependencies, since any formal system for first-order logic will do. However, dependencies are just a fragment of first-order logic, a fragment that seems to be suitable to expressing integrity constraints of databases, and we would like to have a formal system which would enable us to infer only dependencies and not general first-order sentences, unlike a formal system for first-order logic. Finding formal system for fragments of first-order logic is an active research area in mathematical logic, e.g., a formal system for equations [Bir] and a formal system for equational implications [Sel].

The basic operations in our formalism are replacing a relation by its image under some mapping and replacing the image by the source relation. The first operation is called *instantiation* in traditional theorem-proving terminology [CL]. While these operations allow succinct description of complex derivations, they can be replaced by the much simpler operations of duplicating a tuple (the equivalent of an atomic formula) and renaming variables. We do not pursue this point of view in this paper, but the interested reader can easily translate all our formal systems into that equivalent formalism.

In a formal proof we have *facts* and *rules*. The facts represents specific knowledge relevant to a particular case. The rules express general knowledge about a particular subject area and are used as production rules to generate new fact from old ones. The task of the system is to prove a goal fact from the given ones. Basically, there are two kinds of systems. In *forward* systems, the rules operates on the given fact until a termination condition involving the goal fact is achieved. This is also called a bottom-up search. In *backward* systems, the rules operate on the goal fact until a termination condition involving the given facts is achieved. This is also called top-down search. As an example consider refutation of Horn sets [HW]. Positive unit refutation is an example of a forward system, and input refutation is an example of a backward system. In this paper we have three systems. One of them is a forward system, and the other two are backward systems. We consider one of the backward systems to be highly unnatural for reasons to be discussed later.

Though our main interest is in the full class of tuple and equality generating dependencies, several subclasses are also of interest, e.g., total tuple generating dependencies, template dependencies and embedded multivalued dependencies. We address the problem of specializing our formal systems to such subclasses. That is, when the given dependencies and the goal dependency are all of some subclass, we want all dependencies in the derivation to be of that subclass. Not all subclasses of interest are known to have such a specialized formal system.

It is known that equality can be eliminated from first-order logic by adding the equality axioms: reflexivity, symmetry, transitivity, and substitutivity. This can also be applied to one-sorted dependencies [BV2]. Since the identity relation is not many-sorted, e.g., it is reflexive, we can not eliminate equality from our dependencies which are many-sorted. Nevertheless, in [BV3] it is shown that the role of equality can be "minimized" in deciding implication. Hence, our approach is to develop first formal system for tuple generating dependencies, and then, using a theorem of [BV3], to extend these systems to equality generating dependencies.

The outline of the paper is as follows. In § 2 we define the relational model, tableaux and dependencies, and we describe the chase procedure to test implication of dependencies. In § 3 we develop formal systems for total tuple and equality

generating dependencies, viewing a total tuple generating dependency as a tableau operator. In § 4 we generalize these systems to tuple generating dependencies. In § 5 we address the problem of specializing our systems to several subclasses, and we show that finding a formal system for embedded multivalued dependencies is equivalent to solving the implication problem for this subclass. We also show how to "decompose" a total tuple generating dependency to a set of "weaker" dependencies. We conclude with several remarks in § 6.

2. Basic definitions.

2.1. Attributes, tuples and relations. *Attributes* are symbols taken from a given finite set U called the *universe*. All sets of attributes are subset of the universe. We use the letters $A, B, C, \cdots$ to denote attributes and $X, Y, \cdots$ to denote sets of attributes. We do not distinguish between the attribute A and the set $\{A\}$. The union of X and Y is denoted by XY, and the complement of X in the universe is denoted by $\bar{X}$.

With each attribute A is associated an infinite set called its *domain*, denoted DOM (A). The domains of distinct attributes are assumed to be disjoint. The domain of a set of attributes X is DOM $(X) = \bigcup_{A \in X}$ DOM (A). An *X-value* is a mapping $w: X \to \text{DOM}(X)$, such that $w(A) \in \text{DOM}(A)$ for all $A \in X$. An *X-relation* is a non-empty set (not necessarily finite) of X-values. If $X = U$ then we may omit it for simplicity. A *tuple* is a U-value. We use $a, b, c, \cdots$ to denote elements of the domains, $s, t, u, \cdots$ to denote tuples, and $I, J, \cdots$ to denote relations.

For a tuple w and a set $Y \subseteq U$ we denote the restriction of w to Y by $w[Y]$. We do not distinguish between $w[A]$, which is an A-value, and $w(A)$, which is an element of DOM (A). Let I be an X-relation, and let $Y \subseteq X$. Then the *projection* of I on Y, denoted $I[Y]$, is a Y-relation $I[Y] = \{w[Y]: w \in I\}$. The set of all attribute values in an X-relation I is VAL $(I) = \bigcup_{A \in X} I[A]$. For an X-value w, VAL (w) stands for VAL $(\{w\})$.

2.2. Tableaux. A *valuation* is a partial mapping $h: \text{DOM}(U) \to \text{DOM}(U)$ such that if $h(a)$ is defined then $h(a) \in \text{DOM}(A)$ for all $A \in U$ and $a \in \text{DOM}(A)$. The valuation h can be extended to tuples and relations as follows. Let w be a tuple; then $h(w)$ is $h \circ w$ (i.e., h composed with w). Let I be a relation; then $h(I) = \{h(w): w \in I\}$. We say that α is a valuation on a tuple w (a relation I) if α is defined exactly on VAL (w) (VAL (I)). Let α be a valuation on a relation I, and let J be a relation. An *extension* of h to J is a valuation on $I \cup J$ that agrees with h on VAL (I).

A *tableau* [ASU] is a pair $T = \langle w, I \rangle$, where w is a tuple and I is a finite relation, such that VAL $(w) \subseteq$ VAL (I). T defines an operation on relation as follows:

$$T(J) = \{h(w): h \text{ is a valuation s.t. } h(I) \subseteq J\};$$

i.e., $T(J)$ is the set of images of w under all valuations that map every tuple of I to some tuple of J. Observe that $J \subseteq T(J)$.

Example 1. Let $U = \{A, B\}$, DOM $(A) = \{a0, a1, \cdots\}$, and DOM $(B) = \{b0, b1, \cdots\}$. Let I be the relation $\{w1, w2, w3\}$:

	A	B
$w1$:	$a0$	$b1$
$w2$:	$a1$	$b1$
$w3$:	$a1$	$b0$

.

Let w be the tuple

A	B
$a0$	$b0$

.

Now $T = \langle w, I \rangle$ is a tableau. Let J be the relation:

A	B
$a0$	$b0$
$a1$	$b0$
$a1$	$b1$
$a2$	$b1$
$a2$	$b3$

.

$T(J)$ is the relation:

A	B
$a0$	$b0$
$a1$	$b0$
$a1$	$b1$
$a2$	$b1$
$a2$	$b3$
$a0$	$b1$
$a1$	$b3$
$a2$	$b0$

.

Clearly, the values in a tableau serve as formal variables, and therefore can be consistently renamed.

LEMMA 2.1 [ASU]. *Let $\langle w, I \rangle$ be a tableau, and let h be a one-to-one valuation on I. Then, for every relation J, we have $\langle w, I \rangle(J) = \langle h(w), h(I) \rangle(J)$.*

Let w be any tuple, and consider the tableau $\langle w, \{w\} \rangle$. Clearly $\langle w, \{w\} \rangle(I) = I$ for any relation I; i.e. $\langle w, \{w\} \rangle$ defines the identity operation. We will denote this tableau as $\mathbf{1}$.

We now show that the set of tableau operations is actually a monoid by demonstrating that it is closed under composition.[2] Let I be a relation and let u and v be tuples; then $I(u/v)$ is the result of substituting v for u in I. Formally, we define a one-to-one valuation h on $I \cup \{u\}$ as follows: $h(u[A]) = v[A]$ for all $A \in U$, and h is the identity on all other values. Now we define $I(u/v)$ as $h(I)$. If $\langle u, I \rangle$ is a tableau and $\mathrm{VAL}(v) \cap \mathrm{VAL}(I) = \emptyset$, then by Lemma 2.1 for any relation K, $\langle u, I \rangle(K) = \langle v, I(u/v) \rangle(K)$. Let I be a finite nonempty relation, $I = \{w_1, \cdots, w_m\}$. I can be viewed as a mapping from tableaux to relations as follows. Let $\langle u, J \rangle$ be a tableau. We first form m distinct copies of $\langle u, J \rangle$: $\langle u_1, J_1 \rangle, \cdots, \langle u_m, J_m \rangle$, by renaming of values so that no value from $\{u_i\} \cup J_i$ occurs either in I or in $\{u_j\} \cup J_j$, if $i \neq j$. Now $I(u, J)$ is $\bigcup_{i=1}^m J_i(u_i/w_i)$.

LEMMA 2.2. *For any relation K:*

(1) *If h is a valuation such that $h(I(u, J)) \subseteq K$, then $h(I) \subseteq \langle u, J \rangle(K)$.*

(2) *If h is a valuation on I such that $h(I) \subseteq \langle u, J \rangle(K)$, then there is an extension h' of h to $I(u, J)$ such that $h'(I(u, J)) \subseteq K$.*

Proof.

(1) Assume that $h(I(u, J)) \subseteq K$. Then $h(J_i(u_i/w_i)) \subseteq K$, for $1 \leqq i \leqq m$. Thus, $h(w_i) \in \langle w_i, J_i(u_i/w_i) \rangle(K) = \langle u, J \rangle(K)$. I.e., $h(I) \subseteq \langle u, J \rangle(K)$.

[2] This result was independently shown in [FMUY] by a different technique.

(2) Assume that h is a valuation on I such that $h(I) \subseteq \langle u, J\rangle(K)$. Thus, there are valuations $h_1, \cdots, h_m$ such that $h_i(w_i) = h(w_i)$ and $h_i(J_i(u_i/w_i)) \subseteq K$. But $\langle u_i, J_i\rangle$ and $\langle u_j, J_j\rangle$ share no value if $i \neq j$, so there is a valuation h' on $I(u, J)$ which agrees with h_i on $J_i(u_i/w_i)$. It follows that h' is an extension of h and $h'(I(u, J)) \subseteq K$. □

LEMMA 2.3. *Let $\langle w, I\rangle$ and $\langle u, J\rangle$ be tableaux. Then, for every relation K, $\langle w, I\rangle\ (\langle u, J\rangle(K)) = \langle w, I(u, J)\rangle(K)$.*

Proof. Let $t \in \langle w, I(u, J)\rangle(K)$. That is, there is a valuation h on $I(u, J)$ such that $h(I(u, J)) \subseteq K$ and $h(w) = t$. By Lemma 2.2, $h(I) \subseteq \langle u, J\rangle(K)$, so $t \in \langle w, I\rangle(\langle u, J\rangle(K))$.

Let $t \in \langle w, I\rangle(\langle u, J\rangle(K))$. That is, there is a valuation h on I such that $h(I) \subseteq \langle u, J\rangle(K)$ and $h(w) = t$. By Lemma 2.2 there is an extension h' of h to $I(u, J)$ such that $h'(I(u, J)) \subseteq K$, so $t \in \langle w, J(u, J)\rangle(K)$. □

We denote $\langle w, I(u, J)\rangle$ by $\langle w, I\rangle \circ \langle u, J\rangle$.

Example 1 (continued). To construct $T \circ T = T^2$ we first form three distinct copies of T:

	A	B		A	B		A	B
u_1:	$a2$	$b2$	u_2:	$a4$	$b4$	u_3:	$a6$	$b6$
	$a2$	$b3$		$a4$	$b5$		$a6$	$b7$
J_1:	$a3$	$b3$	J_2:	$a5$	$b5$	J_3:	$a7$	$b7$
	$a3$	$b2$		$a5$	$b4$		$a7$	$b6$

.

Now we form $J_i(u_i/w_i)$:

$J_1(u_1/w_1)$		$J_2(u_2/w_2)$		$J_3(u_3/w_3)$	
A	B	A	B	A	B
$a0$	$b3$	$a1$	$b5$	$a1$	$b7$
$a3$	$b3$	$a5$	$b5$	$a7$	$b7$
$a3$	$b1$	$a5$	$b1$	$a7$	$b0$

.

Now we get that $T^2 = \langle w, J\rangle$, where J is:

A	B
$a0$	$b3$
$a3$	$b3$
$a3$	$b1$
$a1$	$b5$
$a5$	$b5$
$a5$	$b1$
$a1$	$b7$
$a7$	$b7$
$a7$	$b0$

.

Let T_1, T_2 be tableaux. We say that T_1 is *covered* by T_2, denoted $T_1 \leqq T_2$, if $T_1(I) \subseteq T_2(I)$, for every relation I.

LEMMA 2.4. [ASU] *Let $\langle u, I\rangle$ and $\langle v, J\rangle$ be tableaux. The following conditions are equivalent*:

(1) $\langle u, I\rangle \leqq \langle v, J\rangle$.

(2) $u \in \langle v, J\rangle(I)$.

(3) *There is a valuation h on J such that $h(J) \subseteq I$ and $h(v) = u$.*

Proof.

(1) $\Rightarrow$ (2). Assume that $\langle u, I\rangle \leqq \langle v, J\rangle$. Then $\langle u, I\rangle(I) \subseteq \langle v, J\rangle(I)$. But clearly $u \in \langle u, I\rangle(I)$, so $u \in \langle v, J\rangle(I)$.

(2) $\Rightarrow$ (3). Assume that $u \in \langle v, J\rangle(I)$. By definition there is a valuation h on J such that $h(J) \subseteq I$ and $h(v) = u$.

(3) $\Rightarrow$ (1). Assume that $h(J) \subseteq I$ and $h(v) = u$. Let $t \in \langle u, I\rangle(K)$. I.e., there is a valuation g on I such that $g(I) \subseteq K$ and $g(u) = t$. But then $g \circ h$ is a valuation on J such that $g \circ h(J) \subseteq K$ and $g \circ h(v) = g(u) = t$, so $t \in \langle v, J\rangle(K)$. □

Put otherwise, $\langle u, I\rangle \leqq \langle v, J\rangle$ iff there are a valuation h on J and a relation I', such that $u = h(v)$ and $I = h(J) \cup I'$. Thus, covering of tableaux can be easily axiomatized, answering a question posed by [ASU]. Searching for the appropriate valuation h can be quite difficult, since testing covering of tableaux is NP-complete [ASU], [BV4].

Example 1 (continued). To show that $T \leqq T^2$, we compute $T^2(I) = T(T(I))$ and get:

A	B
$a0$	$b1$
$a1$	$b1$
$a1$	$b0$
$a0$	$b0$

Now $w \in T^2(I)$, so $T \leqq T^2$. However $T(J)$ is:

A	B
$a0$	$b3$
$a3$	$b3$
$a3$	$b1$
$a1$	$b5$
$a5$	$b5$
$a5$	$b1$
$a1$	$b7$
$a7$	$b7$
$a7$	$b0$
$a0$	$b1$
$a1$	$b1$
$a1$	$b0$

Now we have that $w \notin T(J)$, so $T \leqq T^2$. We leave it to the reader to check that the powers of T form a strictly increasing (with respect to containment) sequence of tableaux.

2.3. Dependencies. For any given application only a subset of all possible relations is of interest. This subset is defined by constraints which are to be satisfied in the relations of interest. A class of constraints that was extensively studied is the class of dependencies.

A *tuple generating dependency* (tgd) says that if some tuples, satisfying certain equalities exist in the relation, then some other tuples (possibly with some unknown values), must also exist in the relation. Formally, a tgd is a pair of finite relations $\langle I', I\rangle$. It is satisfied by a relation J if for every valuation h on I such that $h(I) \subseteq J$ there is an extension h' of h to I' so that $h'(I') \subseteq J$.

Multivalued dependencies (mvd) [Fag1], [Zan] are tgd's of a special form. An mvd is a tgd $\langle\{w\}, I\rangle$, where $|I| \leqq 2$ and $\mathrm{VAL}(w) \subseteq \mathrm{VAL}(I)$. It is usually written $X \twoheadrightarrow Y$, where $X = \{A: |I[A]| = 1\}$ and $Y = \{A: w[A] = u[A]\}$ for some tuple $u \in I$.

An mvd is an example of a *total tuple generating dependency* (ttgd). A tgd $\langle I', I\rangle$ is *total* if VAL $(I') \subseteq$ VAL (I). In this case we can assume a simpler form for the tgd.

LEMMA 2.5. [BV3] *Let $\langle I', I\rangle$ be a ttgd, $I' = \{w_1, \cdots, w_m\}$, and let J be a relation; then J satisfies $\langle I', I\rangle$ if and only if it satisfies $\langle\{w_i\}, I\rangle$ for all $1 \leqq i \leqq m$.*

Thus, we can assume without loss of generality that every ttgd is of the form $\langle\{w\}, I\rangle$, and, since J satisfies the ttgd $\langle\{w\}, I\rangle$ iff $\langle w, I\rangle(J) = J$, we will not distinguish between $\langle\{w\}, I\rangle$ and $\langle w, I\rangle$, and treat it both as a tableau operation and a ttgd. The exact meaning will be clear from the context.

A class of dependencies that lies between the class of ttgd's and the class of mvd's is the class of *join dependencies*. We will not deal with join dependencies in this paper. Formal systems for join dependencies are studied in [BV1], [Sc], [Va1].

An *equality generating dependency* (egd) says that, if some tuples satisfying certain equalities exist in the relation, then some values in these tuples must be equal. Formally, an egd is a pair $\langle(a_1, a_2), I\rangle$, where a_1 and a_2 are A-values for some attribute A, and I is a finite relation such that $a_1, a_2 \in I[A]$. We also call such an egd an A-egd. A relation J satisfies $\langle(a_1, a_2), I\rangle$ if for every valuation h such that $h(I) \subseteq J$ we have $h(a_1) = h(a_2)$. Note that if $a_1 = a_2$ then $\langle(a_1, a_2), I\rangle$ is satisfied by every relation.

Functional dependencies (fd) [Codd] are egd's of a special form. An fd is an egd $\langle(a_1, a_2), I\rangle$, where $|I| = 2$ and $\{a_1, a_2\} = I[A]$. It is usually written $X \to A$, where $X = \{B : |I[B]| = 1\}$.

Example 2. Let $U = \{A, B, C, D\}$, DOM $(A) = \{a0, a1, \cdots\}$, DOM $(B) = \{b0, b1, \cdots\}$ etc. Let I and J be the relations

I:

A	B	C	D
$a0$	$b0$	$c1$	$d0$
$a0$	$b1$	$c0$	$d1$

J:

A	B	C	D
$a0$	$b0$	$b0$	$d0$
$a1$	$b0$	$c0$	$d1$

.

Let u and v be the tuples:

	A	B	C	D
u:	$a0$	$b0$	$c0$	$d0$
v:	$a1$	$b0$	$c0$	$d0$

.

Let d_1 be the ttgd $\langle u, I\rangle$. d_1 is equivalent to the mvd $A \twoheadrightarrow ABD$. Let d_2 be the egd $\langle(a0, a1), J\rangle$. d_2 is equivalent to the fd $BC \to A$. $\langle\{u, v\}, I\rangle$ is a tgd.

A dependency is *trivial* if it is satisfied by every relation.

LEMMA 2.6. [BV3]

(1) *The egd $\langle(a_1, a_2), I\rangle$ is trivial if and only if $a_1 = a_2$.*

(2) *The ttgd $\langle w, I\rangle$ is trivial if and only if $w \in I$.*

(3) *The tgd $\langle I', I\rangle$ is trivial if and only if there is a valuation h on $I \cup I'$ which is the identity on I such that $h(I') \subseteq I$.*

2.4. Implication of dependencies. For a set of dependencies D we denote by SAT (D) the set of relations that satisfy all dependencies in D. D *implies* a dependency d, denoted $D \vDash d$, if SAT $(D) \subseteq$ SAT (d). That is, if d is satisfied for every relation which satisfies all dependencies in D. The *implication problem* is to decide for a given set of dependencies D and a dependency d whether $D \vDash d$. In general the implication problem is recursively unsolvable [BV2], [CLM]. If, however, D consists of egd's and ttgd's then the problem is solvable. A proof procedure[3] for the implication problem—

[3] We distinguish between a *decision procedure* which always halts, and a *proof procedure* which may run forever if the answer to the decision problem is negative.

the chase—was developed in [BV3] generalizing [ABU], [MMS]. (Similar procedures were studied by [Pa], [SU1], [YP].) In the case that D consists of egd's and ttgd's this procedure is a decision procedure.

In the sequel we use D to denote finite sets of dependencies, and we use d and e to denote single dependencies.

Intuitively, to test whether D implies $\langle I', I\rangle$ (or $\langle (a_1, a_2), I\rangle$), we "chase" I by D into SAT (D) and then check if I' is in I (or if a_1 and a_2 are identical in I). Consider first the case that all dependencies are ttgd's. A *chase* of I by D is a sequence of relations $I_0, I_1, \cdots$ such that $I = I_0$ and I_{j+1} is obtained from I_j by an application of a *chase rule*. To each ttgd in D there corresponds a TT-rule:

TT-rule (for a ttgd $\langle w, J\rangle$ in D): I_{j+1} is $\langle w, J\rangle(I_j)$.

Example 2 (continued). Consider the ttgd $\langle u, I\rangle$. The effect of the TT-rule for this ttgd on a relation J is as follows: if t_1 and t_2 are tuples in J such that $t_1[A] = t_2[A]$, then add to J the tuple t defined by $t[ABD] = t_1[ABD]$ and $t[C] = t_2[C]$.

We assume that for all $j \geqq 0$, $I_{j+1} \neq I_j$. Thus, the chase is a strictly increasing sequence, and we have:

LEMMA 2.7. [BV3] *All chases of I by D are finite and have the same last relation, which is in SAT (D).* □

This unique last relation is denoted $\text{chase}_D(I)$. It can be used to decide implication.

THEOREM 2.1. [BV3] *Let D be a set of ttgd's, and let* $\langle w, I\rangle$ *be a ttgd. Then* $D \vDash \langle w, I\rangle$ *if and only if* $w \in \text{chase}_D(I)$. □

Example 1 (continued). Let D be $\{\langle w, J\rangle\}$, and let d be $\langle w, I\rangle$. To show that $D \vDash d$, we compute $\text{chase}_D(I)$. I_0 is just I. I_1 is $\langle w, J\rangle(I_0)$:

A	B
$a0$	$b1$
$a1$	$b1$
$a1$	$b0$
$a0$	$b0$

The reader can verify that $\langle w, J\rangle(I_1) = I_1$, so $\text{chase}_D(I) = I_1$ and, since $w \in I_1$, $D \vDash d$. □

Let us now admit also nontotal tgd's. Trying to generalize our TT-rule to tgd's we encounter difficulties, because the new tuples, whose existence in the relation is implied by the existence of some other tuples, are only partly known. The solution is to replace each unknown value by a new distinct value. Let $\langle I', I\rangle$ be a tgd and h a valuation on I. A *distinct extension* h' of h to I' is an extension h' of h to I', where, for all $a \in \text{VAL}(I') - \text{VAL}(I)$, $h'(a)$ is a new distinct value. (Since there is an infinite supply of values, we can always choose this new value so as to avoid all possible name clashes.) Our generalized chase rule is now:

T-rule (for some $\langle J', J\rangle$ in D): let h be a valuation on J such that $h(J) \subseteq I_j$ but for no extensions g of h to J' we have that $g(J') \subseteq I_j$, and let f be a distinct extension of h to J'. I_{j+1} is $I_j \cup f(J')$.

Unlike the TT-rules, the T-rules are nondeterministic, since they depend on the choice of h. Since this rule introduces new values, the chase may be infinite.

Example 2 (continued). Consider the tgd $\langle \{v\}, I\rangle$. The effect of the T-rule for this tgd on a relation J is as follows: for some tuples t_1 and t_2 in J such that $t_1[A] = t_2[A]$ but there is no tuple t in J with $t[BD] = t_1$ and $t[C] = t_2$, add such a tuple t to J, with $t[A]$ being some new value.

THEOREM 2.2 [BV3]. *Let D be a set of tgd's, and let $\langle I', I\rangle$ be a tgd. Then $D \models \langle I', I\rangle$ if and only if there are a chase of I by D: $I_0, I_1, \cdots$, an element I_n of this chase, and a valuation h that is the identity on I such that $h(I') \subseteq I_n$.* □

Let us now admit also egd's. It seems that we need another chase rule for egd's, and indeed in [BV3] such a rule is used. However, it is also shown there how the use of this rule can be avoided.

Let e be an A-egd $\langle(a_1, a_2), I\rangle$. Let w_1 be a tuple such that $w_1[A] = a_1$, and for all $B \in \bar{A}$, we have $w_1[B] \notin I[B]$. Let w_2 be a tuple such that $w_2[A] = a_2$ and $w_2[\bar{A}] = w_1[\bar{A}]$. We associate with e two ttgd's. e_1 is $\langle w_1, I \cup \{w_2\}\rangle$, and e_2 is $\langle w_2, I \cup \{w_1\}\rangle$. Intuitively, e_1 states that, given I, wherever a_2 appears a_1 also appears. More precisely, if a relation contains $h(I)$, then for each tuple in it which contains $h(a_2)$ there exists a tuple identical to it except that $h(a_2)$ is replaced by $h(a_1)$. Similarly, e_2 states that wherever a_1 appears, a_2 also appears. Let D^* be the result of replacing each egd e in D by e_1 and e_2. The idea is that instead of saying that two values are equal, we say that they "look the same from within the relation".

Example 2 (continued). Let e be $\langle(a0, a1), J\rangle$. e stands for the fd $BC \rightarrow A$. w_1, and w_2 are the tuples:

	A	B	C	D
w_1:	$a0$	$b1$	$c1$	$d2$
w_2:	$a1$	$b1$	$c1$	$d2$

e_1 is $\langle w_1, J \cup \{w_2\}\rangle$, and e_2 is $\langle w_2, J \cup \{w_1\}\rangle$.

THEOREM 2.3 [BV3].

(1) *$e \models e_1$ and $e \models e_2$.*

(2) *Let d be a tgd, $D \models d$ if and only if $D^* \models d$.*

(3) *Let e be an A-egd $\langle(a_1, a_2), I\rangle$. Then $D \models e$ if and only if there is a chase of I by D^*: $I_0, I_1, \cdots$, an element I_n of this chase, an A-egd $\langle(a_3, a_4), J\rangle$ in D, and a valuation h on J such that $h(J) \subseteq I_n$, $h(a_3) = a_1$, and $h(a_4) = a_2$.*

(4) *Let e be a nontrivial egd. Then $D \models e$ if and only if $D^* \models e_1$, and there is a nontrivial A-egd in D.*

We will rely upon Theorems 2.1, 2.2, and 2.3 in developing formal systems for dependencies.

3. Formal systems for ttgd's and egd's. A *formal system* for a family of dependencies consists of axioms and inference rules. The axioms are schemas of trivial dependencies, e.g., the reflexivity axiom for fd's [Arm] and mvd's [BFH]. The inference rules specify whether a dependency is inferable from some premises, e.g., the transitivity rule for fd's [Arm] and mvd's [BFH]. If the number of premises in the rule is bounded then the rule is said to be *bounded*. Let Ψ be a class of dependencies, and let F be a formal system. A *derivation* of a dependency d from a set of dependencies D by F in Ψ is a sequence of dependencies from Ψ: $d_0, d_1, \cdots, d_n$, with $d_n = d$, each of which is either an instance of an axiom of F, a member of D, or is inferrable from earlier d_i's by one of the inference rules of F. We say that d is *derivable* from D by F in Ψ, denoted $D \vdash_{F,\Psi} d$, if there is a derivation of d from D by F in Ψ. If F and Ψ are understood from the context, then we simply write $D \vdash d$. F is *sound* for Ψ if for every $D \subseteq \Psi$, $d \in \Psi$ we have that $D \vdash_{F,\Psi} d$ implies $D \models d$, and is *complete for* Ψ if for every $D \subseteq \Psi$, $d \in \Psi$ we have that $D \models d$ implies $D \vdash_{F,\Psi} d$. To show that F is sound suffice it to show that for every d_i in a derivation of d from D in F, $D \models d_i$. That is, if d_i is an instance of an axiom then it is trivial, i.e., the axioms are sound, and if d_i is inferable from $d_{j_1}, \cdots, d_{j_m}$ then $\{d_{j_1}, \cdots, d_{j_m}\} \models d_i$, i.e., the inference rules are sound.

There are two ways of looking at inference rules. An inference rule can be defined as a recursive predicate saying whether a dependency is inferable from the premises, or it can be defined as a recursive function giving the inferred dependency in answer to the given premises. In the remaining parts of this section we exhibit several formal systems for the family of ttgd's and egd's, usually viewing inference rules as predicates. In § 4 we exhibit formal systems for the family of tgd's and egd's, usually viewing inference rules as functions.

3.1. Tableau composition. Our first formal system consists of one axiom and two inference rules. Its completeness is based upon the following lemma.

LEMMA 3.1. *Let D be a set of ttgd's, and let T be a ttgd. Then $D \models T$ if and only if there is a sequence $T_1, \cdots, T_n$, $n \geqq 0$, of tableaux from D such that $T_n \circ \cdots \circ T_1 \geqq T$. (For $n=0$ the composition is defined as* **1**.)

Proof. $D \models \langle w, I\rangle$ iff (by Theorem 2.1) $w \in \text{chase}_D(I)$ iff there is a sequence $T_1, \cdots, T_n$, $n \geqq 0$, of tableaux from D such that $w \in \text{chase}_D(I) = T_n(\cdots(T_1(I))\cdots) = T_n \circ \cdots \circ T_1(I)$ iff (by Lemma 2.4) $T_n \circ \cdots \circ T_1 \geqq \langle w, I\rangle$. □

We present now the system TT_1:

TTD0 (triviality): $\vdash \langle w, \{w\}\rangle$.

TTD1 (covering): $\langle u, I\rangle \vdash \langle v, J\rangle$ if $\langle v, J\rangle \leqq \langle u, I\rangle$.

TTD2 (composition): $\langle u, I\rangle, \langle v, J\rangle \vdash \langle u, I\rangle \circ \langle v, J\rangle$.

TTD0 is analogous to the J-axiom of [BV1], [Va1], TTD1 is analogous to their covering rule, and TTD2 is analogous to their projection-substitution rule.

THEOREM 3.1. *The system TT_1 is sound and complete for ttgd's.*

Proof.

Soundness. $\langle w, \{w\}\rangle$ is a trivial ttgd by Lemma 2.6.

Suppose that $\langle v, J\rangle \leqq \langle u, I\rangle$ and $K \in \text{SAT}(\langle u, I\rangle)$. Then $K \subseteq \langle v, J\rangle(K) \subseteq \langle u, I\rangle(K) = K$. So $K \in \text{SAT}(\langle v, J\rangle)$, and TTD1 is sound.

Suppose now that $K \in \text{SAT}(\langle u, I\rangle, \langle v, J\rangle)$. Then

$$K \subseteq (\langle u, I\rangle \circ \langle v, J\rangle)(K) = \langle u, I\rangle(\langle v, J\rangle(K)) = \langle u, I\rangle(K) = K.$$

So $K \in \text{SAT}(\langle u, I\rangle \circ \langle v, J\rangle)$, and TTD2 is sound.

Completeness. Suppose that $D \models \langle w, I\rangle$. By Lemma 3.1, there is a sequence $T_1, \cdots, T_n$, $n \geqq 0$, of tableaux from D such that $T_n \circ \cdots \circ T_1 \geqq \langle w, I\rangle$. By $n-1$ applications of TTD2 (or one application of TTD0) we get $D \vdash T_n \circ \cdots \circ T_1$, and applying TTD1 we get $D \vdash \langle w, I\rangle$. □

3.2. Tableau simplification. Rule TTD2 enables us to derive a "big" ttgd from "smaller" ones. In this section we introduce a rule which enables us to derive from given ttgd's a ttgd of reduced size.

We now present the system TT_2, which has one axiom and one inference rule:

TTD0′ (triviality): $\vdash \langle w, \{w\} \cup I\rangle$.

TTD3 (simplification): $\langle u, I\rangle, \langle v, \langle u, I\rangle(J)\rangle \vdash \langle v, J\rangle$.

Rule TTD0′ is a stronger version of TTD0. By Lemma 2.6, it characterizes all trivial ttgd's. Rule TTD3 has no analogue in the formal systems for join dependencies in [BV1], [Sc], [Va1].

THEOREM 3.2. *The system TT_2 is sound and complete for ttgd's.*

Proof.

Soundness. $\langle w, \{w\} \cup I\rangle$ is a trivial ttgd by Lemma 2.6. To prove that rule TTD3 is sound we have to show that $D \models \langle v, J\rangle$, if $D = \{\langle u, I\rangle, \langle v, \langle u, I\rangle(J)\rangle\}$. We compute a chase of J by D. J_0 is J, J_1 is $\langle u, I\rangle(J_0) = \langle u, I\rangle(J)$, and J_2 is $\langle v, \langle u, I\rangle(J)\rangle(J_1) = \langle v, \langle u, I\rangle(J)\rangle(\langle u, I\rangle(J))$. Thus, $v \in J_2 \subseteq \text{chase}_D(I)$, and $D \models \langle v, J\rangle$.

Completeness. Suppose that $D \models \langle w, I \rangle$. By Theorem 2.1, there is a sequence $T_1, \cdots, T_n$, $n \geqq 0$, of tableaux from D such that $w \in T_n(\cdots(T_1(I)\cdots)$. By TTD0′, $D \vdash \langle w, T_n(\cdots(T_1(I)\cdots)\rangle$, and n applications of TTD3 give $D \vdash \langle w, I \rangle$. □

3.3. Tableau transitivity. Our third system TT_3 consists of rule TTD0′ from the previous section and rule TTD4, which is reminiscent of the transitivity rule for fd's and mvd's but unlike them is not a bounded rule. Rule TTD4 is analogous to rule JD5 for join dependencies of [BV1], [Va1].

TTD4 (transitivity). $\langle w, I \rangle, \langle u_1, J \rangle, \cdots, \langle u_m, J \rangle \vdash \langle u, J \rangle$, if there is a valuation h such that

$$h(I) \subseteq \{u_1, \cdots, u_m\} \text{ and } h(w) = u.$$

The condition in the rule can be reformulated as $u \in \langle w, I \rangle(\{u_1, \cdots, u_m\})$.

To prove completeness we use the following observation.

LEMMA 3.2. *Let D be a set of ttgd's, and let I be a finite relation. Then* $\text{chase}_D(I) = \{w : D \models \langle w, I \rangle\}$.

Proof. By Theorem 2.1, $w \in \text{chase}_D(I)$ iff $D \models \langle w, I \rangle$. □

THEOREM 3.3. *The system TT_3 is sound and complete for ttgd's.*

Proof.

Soundness. To prove that TTD4 is sound we have to show that $D \models \langle u, J \rangle$, if $D = \{\langle w, I \rangle, \langle u_1, J \rangle, \cdots, \langle u_m, J \rangle\}$, and there is a valuation h such that $h(I) \subseteq \{u_1, \cdots, u_m\}$ and $h(w) = u$. We compute a chase of J by D. J_0 is J, J_i is $\langle u_i, J \rangle(J_{i-1})$, for $1 \leqq i \leqq m$, and J_{m+1} is $\langle w, I \rangle(J_m)$. Clearly, $\{u_1, \cdots, u_m\} \subseteq J_m$, so $u \in J_{m+1} \subseteq \text{chase}_D(J)$, and $D \models \langle u, J \rangle$.

Completeness. By Lemma 3.2, suffice it to show that for every $u \in \text{chase}_D(J)$, $D \vdash \langle u, J \rangle$. Let $J_0, \cdots, J_n$ be a chase of J by D. We show by induction on i that for every $u \in J_i$, we have $D \vdash \langle u, J \rangle$. J_0 is J, so if $u \in J$, then $D \vdash \langle u, J \rangle$ by rule TTD0′. Suppose now that the assumption holds for $J_i = \{u_1, \cdots, u_m\}$. Let $u \in J_{i+1}$. That is, there is a ttgd $\langle w, I \rangle \in D$, and a valuation h such that $h(I) \subseteq J_i$ and $h(w) = u$. By the induction hypothesis, $D \vdash \langle u_k, J \rangle$, for $1 \leqq k \leqq m$, so $D \vdash \langle u, J \rangle$ by rule TTD4. □

3.4. ttgd's and egd's. Using Theorem 2.3 we can easily extend the formal system for ttgd's of the preceding sections to deal with egd's as well.

We first present rules that deal only with egd's.

ED0 (triviality). $\vdash \langle (a, a), I \rangle$, if $a \in \text{VAL}(I)$.

With an eye to Lemma 2.4 we define covering for egd's. An egd $\langle (a_1, a_2), I \rangle$ *covers* an egd $\langle (a_3, a_4), J \rangle$, denoted $\langle (a_3, a_4), J \rangle \leqq \langle (a_1, a_2), I \rangle$, if there is a valuation h such that $h(I) \subseteq J$, $h(a_1) = a_3$, and $h(a_2) = a_4$.

ED1 (covering): $\langle (a_1, a_2), I \rangle \vdash \langle (a_3, a_4), J \rangle$, if $\langle (a_3, a_4), J \rangle \leqq \langle (a_1, a_2), I \rangle$.

LEMMA 3.3. *Rules* ED0 *and* ED1 *are sound.*

Proof. The soundness of ED0 follows from Lemma 2.6, and that of ED1 from Theorem 2.3. □

The next rule enables us to infer ttgd's from egd's. Recall that with each egd e we associate two ttgd's e_1 and e_2.

ETTD0 (translation): $e \vdash e_1$, $e \vdash e_2$.

LEMMA 3.4. *Let F be any sound and complete formal system for ttgd's, and let F' be $F \cup \{\text{ETTD0}\}$. If D is a set of ttgd's and egd's, then $D \vdash_{F'} \langle w, I \rangle$ if and only if $D \models \langle w, I \rangle$.*

Proof.

Only if. We have to show that rule ETTD0 is sound, but this follows immediately from Theorem 2.3.

If. Suppose that $D \models \langle w, I \rangle$. By Theorem 2.3, $D^* \models \langle w, I \rangle$, so by assumption, $D^* \vdash \langle w, I \rangle$. But $D \vdash D^*$ by rule ETTD0. The claim follows. □

All the formal systems in the sequel include ED0 and ETTD0. Thus, in view of Lemmas 2.6 and 3.4, in order to prove completeness it suffices to consider implication of non-trivial egd's, and we can also use the fact that $D \vdash_{\mathrm{ETTD0}} D^*$ without mentioning it explicitly.

We present now three inference rules analogous to TTD2, TTD3, and TTD4 respectively.

ETTD1 (composition). $\langle (a_1, a_2), I \rangle, \langle u, J \rangle \vdash \langle (a_1, a_2), I(u, J) \rangle$.

ETTD2 (simplification). $\langle u, I \rangle, \langle (a_1, a_2), \langle u, I \rangle (J) \rangle \vdash \langle (a_1, a_2), J \rangle$.

ETTD3 (transitivity). $\langle (a_3, a_4), I \rangle, \langle u_1, J \rangle, \cdots, \langle u_m, J \rangle \vdash \langle (a_1, a_2), J \rangle$, if there is a valuation h such that $h(I) \subseteq \{u_1, \cdots, u_m\}$, $h(a_3) = a_1$, and $h(a_4) = a_2$.

Let ETT_1 be the system $TT_1 \cup \{\mathrm{ED0}, \mathrm{ED1}, \mathrm{ETTD0}, \mathrm{ETTD1}\}$, let ETT_2 be the system $TT_2 \cup \{\mathrm{ED0}, \mathrm{ED1}, \mathrm{ETTD0}, \mathrm{ETTD2}\}$, and let ETT_3 be the system $TT_3 \cup \{\mathrm{ED0}, \mathrm{ETTD0}, \mathrm{ETTD3}\}$.

THEOREM 3.4. *The system ETT_1 is sound and complete for ttgd's and egd's.*

Proof.

Soundness. We have to show that rule ETTD1 is sound. Let $K \in \mathrm{SAT}(\langle (a_1, a_2), I \rangle, \langle v, J \rangle)$, and suppose that $h(I(v, J)) \subseteq K$. By Lemma 2.2, $h(I) \subseteq \langle v, J \rangle (K) = K$. It follows that $h(a_1) = h(a_2)$ and $K \in \mathrm{SAT}\langle (a_1, a_2), I(v, J) \rangle$.

Completeness. Suppose that $D \models \langle (a_1, a_2), I \rangle$. By Theorem 2.3, there is a sequence $T_1, \cdots, T_n$, $n \geqq 0$, of tableaux from D^*, an egd $\langle (a_3, a_4), J \rangle$ from D, and a valuation h on J such that $h(J) \subseteq T_n(\cdots(T_1(I))\cdots)$, $h(a_3) = a_1$, and $h(a_4) = a_2$. Let $T_n \circ \cdots \circ T_1$ be $\langle u, K \rangle$. By rule TTD2 or TTD0, $D \vdash \langle u, K \rangle$, so by ETTD2, $D \vdash \langle (a_3, a_4), J(u, K) \rangle$. But, by Lemma 2.2, there is an extension h' of h to $J(u, K)$ such that $h'(J(u, K)) \subseteq I$. It follows that

$$\langle (a_1, a_2), I \rangle \leqq \langle (a_3, a_4), J(u, K) \rangle,$$

and by rule ED1, $D \vdash \langle (a_1, a_2), I \rangle$. □

The proofs of Theorems 3.5 and 3.6 are analogous to the proof of Theorem 3.4, and are left to the reader.

THEOREM 3.5. *The system ETT_2 is sound and complete for ttgd's and egd's.*

THEOREM 3.6. *The system ETT_3 is sound and complete for ttgd's and egd's.*

Up to now we have referred only to clauses (1), (2) and (3) in Theorem 2.3. By referring to clause (4) in that theorem we get rule ETTD4, which is more flexible than ETTD1, ETTD2, and ETTD3 in the sense that it can be combined with any sound and complete system for ttgd's to give a sound and complete system for ttgd's and egd's.

ETTD4: $d, e_1 \vdash e$, if d and e are A-egd's, and d is nontrivial.

THEOREM 3.7. *Let F be a sound and complete system for ttgd's, and let F' be $F \cup \{\mathrm{ED0}, \mathrm{ETTD0}, \mathrm{ETTD4}\}$. Then F' is a sound and complete system for ttgd's and egd's.*

Proof.

Soundness. We have to show that rule ETTD5 is sound, but this is immediate from Theorem 2.3.

Completeness. Suppose that $D \models e$, where e is a nontrivial A-egd. By Theorem 2.3, $D^* \models e_1$, and there is a nontrivial A-egd d in D. By assumption $D^* \vdash e_1$, so by rules ETTD0 and ETTD4, $D \vdash e$. □

3.5. Discussion. We refer now to the forward/backward classification of our formal systems. In deriving $\langle w, I\rangle$ from D, w is the goal fact, and the tuples of I are the given facts. The dependencies in D are the production rules. The system TT_3 is clearly a forward system. Starting with the tuples of I, one generates additional tuples (facts) until the goal tuple is generated. Actually, this is a direct simulation of the chase. In contrast, the system TT_1 is a backward system where the rules operate on a collection of goal tuples K. Initially, K is just $\{w\}$. The operation of a ttgd $\langle u, J\rangle$ from D on the set of goal tuples is to replace a tuple $v \in K$ by $J(u/v)$. The process terminates when there is a valuation h which is the identity on w such that $h(K) \subseteq I$.

The system TT_2 is another type of backward system. It starts with guessing an initial collection of facts that includes also the goal fact. This collection is just $\text{chase}_D(I)$. The operation of a ttgd $\langle u, J\rangle$ from D on this collection of guessed facts is to eliminate facts that are implied by other facts, i.e., replacing $\langle u, J\rangle(K)$ by K. The termination condition is that all the remaining facts are given ones, viz., members of I. It is the initial guess of all facts that makes this system highly unnatural. While in the other systems one starts from the problem (w or I) and then has to guide the production, there seems no natural way of guessing $\text{chase}_D(I)$ "straight from the blue".

These observations extend to the systems ETT_1, ETT_2, and ETT_3.

Let us refer now to the *length* and *size* of the derivations in our formal systems. (By the size of a derivation we mean the number of symbols in the derivation.) If $D \models \langle w, I\rangle$, then for all chases of I by D: $I_0, I_1, \cdots, I_n = \text{chase}_D(I)$ we have $w \in I_n$. In [BV3] we show that both n and the size of the I_j's can be exponential in the size of I. Thus, it is clear that the derivations that were constructed in the various completeness proofs can be of length and size exponential in the size of D and I, since they all simulate the chase, directly or indirectly. Can we construct smaller or shorter derivations?

The answer is probably negative. In [Va3] it is shown that, given a derivation in any of the above systems, one can translate it into a derivation in any of the other systems with at most a polynomial increase in the length and size of the derivation. Furthermore, given a derivation in the system TT_3 whose length is polynomial in the size of D and I, one can construct another derivation whose size is polynomial in the size of D and I. Now, for a sequence $d_0, \cdots, d_n$, one can check in space that is polynomial in the size of the input, whether it is a derivation of d from D by any of the aforementioned formal systems. If we could bound the length of the derivations by a polynomial in the size of D and I, it would follow that the set $\{\langle D, d\rangle : D \models d\}$ is in PSPACE. This is quite unlikely, since it is shown in [CLM] that this set is logspace complete in EXPTIME.

4. Formal systems for tgd's and egd's. The completeness proof for the systems in the previous section used the fact that $\text{chase}_D(I) = T_n(\cdots(T_1(I)\cdots)$. Allowing tgd's in D, that is no longer true. Nevertheless, we will be able to generalize our systems to deal with tgd's, using the essential ideas underlying them.

Let $\langle I', I\rangle$ be a tgd. It can be viewed as an implicational constraint where I serves as the antecedent and I' serves as the consequent, saying roughly, that if I can be "embedded" in a relation J then so can be I'.[4] We partition the set of values in $I \cup I'$ into two sets. The set of *existential* values in $\langle I', I\rangle$ is $\text{EX}(I', I) = \text{VAL}(I') - \text{VAL}(I)$, and the set of *universal* values is the set $\text{VAL}(I)$. (The source for this terminology is the way dependencies are written as first-order sentences. See [Fag2], [Va3].) It is

[4] Indeed, tgd's and egd's are called *embedded implicational dependencies* in [Fag2].

clear that existential and universal values play completely different roles in the "meaning" of $\langle I', I\rangle$. If h is a valuation on I then to extend it to I' we have to define it on $\mathrm{EX}(I', I)$.

4.1. Substitution. Studying rule TTD2, we observe that the basic operation there is that of replacing, when given $\langle w, I\rangle$ and $\langle u, J\rangle$, a tuple v in I by $J(u/v)$, because the existence of $J(u/v)$ entails the existence of v. We generalize that to tgd's.

We present now the system T_1:

TD0 (triviality). $\vdash\langle I', I\rangle$, if there is a valuation h which is the identity on I such that $h(I')\subseteq I$.

TD1 (collapsing). $\langle I', I\rangle\vdash\langle h(I'), h(I)\rangle$, if h is a valuation such that $h|_{\mathrm{EX}(I',I)}$ is a one-to-one mapping into $\mathrm{EX}(h(I'), h(I))$.

TD2 (augmentation). $\langle I', I\rangle\vdash\langle I', I\cup J\rangle$, if $\mathrm{EX}(I', I)\cap \mathrm{VAL}(J)=\varnothing$.

TD3 (projection). $\langle I'\cup J, I\rangle\vdash\langle I', I\rangle$.

TD4 (substitution). $\langle I', I\cup J'\rangle$, $\langle J', J\rangle\vdash\langle I', I\cup J\rangle$, if $\mathrm{VAL}(I)\cap\mathrm{VAL}(J')\subseteq \mathrm{VAL}(J)$ and $\mathrm{EX}(I', I\cup J')\cap\mathrm{VAL}(J)=\varnothing$.

Rule TD3 has no analogous rule for ttgd's. Rule TD0 generalizes rule TTD0, rules TD1 and TD2 generalize rule TTD1, and rule TD4 generalizes rule TTD2.

THEOREM 4.1. *The system T_1 is sound and complete for tgd's.*

Proof.

Soundness. TD0 is sound by Lemma 2.6.

Let $J\in\mathrm{SAT}(\langle I', I\rangle)$, let h be a valuation such that $h|_{\mathrm{EX}(I',I)}$ is a one-to-one mapping into $\mathrm{EX}(h(I'), h(I))$, and let g be a valuation on $h(I)$ such that $g(h(I))\subseteq J$. Since J satisfies $\langle I', I\rangle$, there is an extension f of $g\circ h$ to I' such that $f(I')\subseteq J$. We define an extension g' of g to $h(I')$ as follows. Let $a\in\mathrm{EX}(h(I'), h(I))$. Then there is a unique value $a'\in\mathrm{EX}(I', I)$ such that $a=h(a')$. We let $g'(a)=f(a')$. Now $g'(h(I'))=f(I')\subseteq J$, so TD1 is sound.

Let $K\in\mathrm{SAT}(\langle I', I\rangle)$, and suppose that $\mathrm{EX}(I', I)\cap\mathrm{VAL}(J)=\varnothing$. If h is a valuation on $I\cup J$ such that $h(I\cup J)\subseteq K$, then there is an extension h' of $h|_I$ to I' such that $h'(I')\subseteq K$. Clearly, h' is also an extension of h to I', so TD2 is sound.

Let $K\in\mathrm{SAT}(\langle I'\cup J, I\rangle)$, and let h be a valuation on I such that $h(I)\subseteq K$. There is an extension g of h to $I'\cup J$ such that $h(I'\cup J)\subseteq K$. Clearly, g is also an extension of h to I' such that $g(I')\subseteq K$, so TD3 is sound.

Let $K\in\mathrm{SAT}(\langle I', I\cup J'\rangle, \langle J', J\rangle)$, where $\mathrm{VAL}(I)\cap\mathrm{VAL}(J')\subseteq\mathrm{VAL}(J)$ and $\mathrm{EX}(I', I\cup J')\cap\mathrm{VAL}(J)=\varnothing$. Let h be a valuation on $I\cup J$ such that $h(I\cup J)\subseteq K$. Now $\mathrm{EX}(J', J)\cap\mathrm{VAL}(I)=\varnothing$, and by TD2, $K\in\mathrm{SAT}(J', J\cup I)$, so there is an extension g of h to J' such that $g(J')\subseteq K$, and therefore $g(I\cup J')\subseteq K$. g is undefined on $\mathrm{EX}(I', I\cup J')$, because $\mathrm{EX}(I', I\cup J')\cap\mathrm{VAL}(I\cup J\cup J')=\varnothing$. Since K satisfies $\langle I', I\cup J\rangle$, there is an extension f of g to I' such that $f(I')\subseteq K$. f is also an extension of h, so TD4 is sound.

Completeness. Suppose that $D\models\langle I', I\rangle$. Then there are a chase of I by $D: I_0, I_1, \cdots$, an element I_n of this chase, and a valuation h which is the identity on I such that $h(I')\subseteq I_n$. We can assume without loss of generality that $\mathrm{EX}(I', I)\cap\mathrm{VAL}(I_n)=\varnothing$. We construct, by backward induction on k from n to 0, a relation J_k such that $D\vdash\langle I', J_k\rangle$, $J_k\subseteq I_k$, and $\mathrm{EX}(I', J_k)\cap\mathrm{VAL}(I_k)=\varnothing$.

Basis $(k=n)$. Let $J_n=h(I')$. By TD0, $D\vdash\langle I', h(I')\rangle$, $h(I')\subseteq I_n$, and $\mathrm{EX}(I', h(I'))\cap\mathrm{VAL}(I_n)=\varnothing$.

Induction. Suppose that $D\vdash\langle I', J_{k+1}\rangle$, $J_{k+1}\subseteq I_{k+1}$, and $\mathrm{EX}(I', J_{k+1})\cap\mathrm{VAL}(I_{k+1})=\varnothing$. There are some $\langle K', K\rangle$ in D and a valuation g on $K\cup K'$ such that $g|_{\mathrm{EX}(K',K)}$ is a one-to-one mapping into $\mathrm{EX}(g(K), g(K'))$, $I_{k+1}=I_k\cup g(K')$, $g(K)\subseteq I_k$,

and EX $(g(K'), g(K)) \cap \text{VAL}(I_k) = \varnothing$. J_{k+1} can be viewed as $J' \cup J''$, where $J' \subseteq I_k$ and $J'' \subseteq g(K')$. By TD1 and TD3, $\langle K', K\rangle \vdash \langle J'', g(K)\rangle$. Now

$$\text{VAL}(J') \cap \text{VAL}(J'') \subseteq \text{VAL}(I_k) \cap \text{VAL}(g(K')) \subseteq \text{VAL}(g(K)),$$

and

$$\text{EX}(I', J_{k+1}) \cap \text{VAL}(g(K)) \subseteq \text{EX}(I', J_{k+1}) \cap \text{VAL}(I_{k+1}) = \varnothing,$$

so by TD4, $D \vdash \langle I', J_k\rangle$, where $J_k = J' \cup g(K)$. Clearly, $J_k \subseteq I_k$. Also

$$\text{EX}(I', J_k) \subseteq \text{EX}(I', J_{k+1}) \cup \text{EX}(g(K'), g(K)),$$

and since EX $(g(K'), g(K)) \cap \text{VAL}(I_k) = \varnothing$, we have EX $(I', J_k) \cap \text{VAL}(I_k) = \varnothing$.

In particular, we have $D \vdash \langle I', J_0\rangle$, $J_0 \subseteq I$, and EX $(I', J_0) \cap \text{VAL}(I) = \varnothing$, so by TD2, $D \vdash \langle I', I\rangle$. □

4.2. Tuple elimination. Studying rule TTD3, we observe that the basic operation is that of eliminating, when given $\langle w, I\rangle$ and $\langle u, J\rangle$, a tuple v from I, if the tuples of $J(u/v)$ are in I, because the existence of the tuples of $J(u/v)$ implies the existence of v. We generalize this to tgd's.

We now present the system T_2:

TD0 (triviality)

TD1 (collapsing)

TD2 (augmentation)

TD5 (tuple elimination). $\langle I', I \cup J\rangle, \langle J, I\rangle \vdash \langle I', I\rangle$.

Rule TD5 is implied by rule TD4. For the simplicity of the system we pay with less natural derivations.

THEOREM 4.2. *The system T_2 is sound and complete for tgd's.*

Proof.

Soundness. The rules are implied by the rules of the system T_1, so they are sound.

Completeness. Suppose that $D \models \langle I', I\rangle$. Then there are a chase of I by $D: I_0, I_1, \cdots$, an element I_n of this chase, and a valuation h which is the identity on I such that $h(I') \subseteq I_n$. We can assume without loss of generality that EX $(I', I) \cap \text{VAL}(I_n) = \varnothing$. We leave it to the reader to show, by backward induction on k from n to 0, that $D \vdash \langle I', I_k\rangle$. In particular, it follows that $D \vdash \langle I', I\rangle$. □

4.3. Transitivity. Studying rule TTD4 we see that it is basically a transitivity rule. The essential idea is that if the existence of a set of tuples J implies the existence of a set of tuples J', and the existence of J' implies the existence of a tuple w, then the existence of J implies the existence of w. We generalize this to tgd's. It turns out that in contrast to TTD4 the generalized transitivity rule is a bounded rule.

We present now the system T_3:

TD0′ (triviality). $\vdash \langle I, I\rangle$.

TD1 (collapsing)

TD2 (augmentation)

TD3 (projection)

TD6 (weakening). $\langle h(I'), I\rangle \vdash \langle I', I\rangle$, if h is the identity on I.

TD7 (transitivity). $\langle J, I'\rangle, \langle I', I\rangle \vdash \langle I' \cup J, I\rangle$, if EX $(J, I') \cap \text{VAL}(I) = \varnothing$.

THEOREM 4.3. *The system T_3 is sound and complete for tgd's.*

Proof.

Soundness. Left to the reader.

Completeness. Suppose that $D \models \langle I', I\rangle$. Then there are a chase of I by $D: I_0, I_1, \cdots$, an element I_n of this chase, and a valuation that is the identity on I such that $h(I') \subseteq I_n$. We leave it to the reader to show, by induction on k, that $D \vdash \langle I_k, I\rangle$.

In particular, it follows that $D \vdash \langle I_n, I \rangle$. Since $h(I') \subseteq I_n$ and h is the identity on I, $D \vdash \langle I', I \rangle$ by TD3 and TD6. □

4.4. tgd's and egd's. The extension of the systems T_1, T_2, and T_3 to egd's is very similar to the extension of TT_1, TT_2, and TT_3 in § 3.4.

The rules for egd's need essentially no change. We change however the covering rule for the sake of uniformity.

ED0 (triviality). $\vdash \langle (a, a), I \rangle$, if $a \in \text{VAL}(I)$.

ED2 (collapsing). $\langle (a_1, a_2), I \rangle \vdash \langle (h(a_1), h(a_2)), h(I) \rangle$, for any valuation h.

ED3 (augmentation). $\langle (a_1, a_1), I \rangle \vdash \langle (a_1, a_2), I \cup J \rangle$.

LEMMA 4.1. *Rules* ED2 *and* ED3 *are sound.*

Proof. Both rules follow from rule ED1. □

The translation rule needs no change.

ETD0. (translation). $e \vdash e_1$, $e \vdash e_2$.

LEMMA 4.2. *Let F be any sound and complete formal system for tgd's, and let F' be $F \cup \{\text{ETD0}\}$. If D is a set of tgd's and egd's then $D \vdash_{F'} \langle I', I \rangle$ if and only if $D \models \langle I', I \rangle$.*

Proof. Identical to the proof of Lemma 3.4.

The mixed (tgd-egd) analogues of rules TD4, TD5 and TD6 turn out to be minor variants of the transitivity rule.

ETD1 (transitivity). $\langle (a_1, a_2), I \rangle, \langle I, J \rangle \vdash \langle (a_1, a_2), J \rangle$, if $a_1, a_2 \in \text{VAL}(J)$.

THEOREM 4.4. *Let F be any sound and complete formal system for tgd's, and let F' be $F \cup \{\text{ED0, ED2, ED3, ETD0, ETD1}\}$. Then F' is a sound and complete system for tgd's and egd's.*

Proof.

Soundness. We have to show that rule ETD1 is sound. Let $K \in \text{SAT}(\langle (a_1, a_2), I \rangle, \langle I, J \rangle)$, and let h be a valuation on J such that $h(J) \subseteq K$. There is an extension g of h to I such that $g(I) \subseteq K$, so $g(a_1) = g(a_2)$. But $a_1, a_2 \in \text{VAL}(J)$, so $h(a_1) = g(a_1) = g(a_2) = h(a_2)$.

Completeness. Suppose that $D \models \langle (a_1, a_2), I \rangle$, $a_1 \neq a_2$. By Theorem 2.3 there are a chase of I by D^*, an element I_n of this chase, an egd $\langle (a_3, a_4), J \rangle$ from D, and a valuation h such that $h(J) \subseteq I_n$, $h(a_3) = a_1$, and $h(a_4) = a_2$. Now $D^* \models \langle I_n, I \rangle$, so by assumption $D^* \vdash \langle I_n, I \rangle$. By ED2 and ED3, $\langle (a_3, a_4), J \rangle \vdash \langle (a_1, a_2), I_n \rangle$, so by ETD0 and ETD1, $D \vdash \langle (a_1, a_2), I \rangle$. □

By using clause (4) in Theorem 2.3 we can generalize Theorem 3.7 to tgd's.

ETD2: $d, e_1 \vdash e$, if d and e are A-egd's, and d is nontrivial.

THEOREM 4.5. *Let F be a sound and complete formal system for tgd's, and let F' be $F \cup \{\text{ED0, ETD0, ETD2}\}$. Then F' is a sound and complete system for tgd's and egd's.*

Proof. Identical to the proof of Theorem 3.7. □

4.5. Discussion. While the systems TT_1, TT_2, and TT_3 all look completely different each from the other, the systems T_1, T_2, and T_3 are very similar. Indeed, since T_2 is the simplest system of the three, one may ask why we have bothered to study also T_1 and T_3. The reason is that derivations by T_2 are highly unnatural backward derivations as discussed in § 3.5. In contrast, T_1/T_3 are forward/backward systems which yield natural derivations that correspond to known theorem-proving procedures as is shown in [Va3].

Since the implication problem for tgd's is unsolvable [BV2], [CLM], there can be no recursive bound on the size of derivations of tgd's. Furthermore, it can be shown that a recursive bound on the length of the derivations would lead to a recursive bound on the size of the derivations. Thus, there can be no such bound. Nevertheless,

the size of derivations is a reasonable measure to compare between formal systems. We now show that our formal systems can each simulate the other with only a linear increase in the size of derivations. That is, there is a constant c, such that if there is a derivation of size s of d from D by T_i, for some $1 \leqq i \leqq 3$, then for every $1 \leqq j \leqq 3$, there is a derivation of d from D whose size is at most cs. Thus, we can consider all our formal systems as equally "powerful".

T_1 simulates T_2. We have to simulate rule TD5, but TD5 is just a special case of TD4.

T_2 simulates T_3. We have to simulate rules TD0, TD3, TD6 and TD7.

TD0′ is just a special case of TD0. Suppose now that $\langle I' \cup J, I\rangle \vdash_{\text{TD3}} \langle I', I\rangle$. We have

$$\vdash_{\text{TD0}} \langle I', I \cup I' \cup J\rangle, \text{ and}$$

$$\langle I', I \cup I' \cup J\rangle, \langle I' \cup J, I\rangle \vdash_{\text{TD5}} \langle I', I\rangle.$$

Suppose that $\langle h(I'), I\rangle \vdash_{\text{TD6}} \langle I', I\rangle$. I.e., h is the identity on I. We have

$$\vdash_{\text{TD0}} \langle I', h(I') \cup I\rangle, \text{ and}$$

$$\langle I', h(I') \cup I\rangle, \langle h(I'), I\rangle \vdash_{\text{TD5}} \langle I', I\rangle.$$

Suppose now that $\langle J, I'\rangle, \langle I', I\rangle \vdash_{\text{TD7}} \langle I' \cup J, I\rangle$. I.e., $\text{EX}(J, I') \cap \text{VAL}(I) = \varnothing$. We have

$$\vdash_{\text{TD0}} \langle I' \cup J, I' \cup I \cup J\rangle,$$

$$\langle J, I'\rangle \vdash_{\text{TD2}} \langle J, I' \cup I\rangle \text{ (because } \text{EX}(J, I') \cap \text{VAL}(I) = \varnothing),$$

$$\langle I' \cup J, I' \cup I \cup J\rangle, \langle J, I' \cup I\rangle \vdash_{\text{TD5}} \langle I' \cup J, I' \cup I\rangle, \text{ and}$$

$$\langle I' \cup J, I' \cup I\rangle, \langle I', I\rangle \vdash_{\text{TD5}} \langle I' \cup J, I\rangle.$$

T_3 simulates T_1. We have to simulate rules TD0 and TD4.

Suppose now that $\vdash_{\text{TD0}} \langle I', I\rangle$. That is, there is a valuation h which is the identity on I such that $h(I') \subseteq I$. We have

$$\vdash_{\text{TD0}'} \langle I, I\rangle,$$

$$\langle I, I\rangle \vdash_{\text{TD3}} \langle h(I'), I\rangle, \text{ and}$$

$$\langle h(I'), I\rangle \vdash_{\text{TD6}} \langle I', I\rangle.$$

Suppose that $\langle I', I \cup J'\rangle, \langle J', J\rangle \vdash_{\text{TD4}} \langle I', I \cup J\rangle$, i.e., $\text{VAL}(I) \cap \text{VAL}(J') \subseteq \text{VAL}(J)$ and $\text{EX}(I', I \cup J') \cap \text{VAL}(J) = \varnothing$. We have

$$\vdash_{\text{TD0}'} \langle I \cup J, I \cup J\rangle,$$

$$\langle J', J\rangle \vdash_{\text{TD2}} \langle J', I \cup J\rangle \text{ (because } \text{EX}(J', J) \cap \text{VAL}(I) = \varnothing),$$

$$\langle I \cup J, I \cup J\rangle, \langle J', I \cup J\rangle \vdash_{\text{TD7}} \langle I \cup J \cup J', I \cup J\rangle,$$

$$\langle I', I \cup J'\rangle \vdash_{\text{TD2}} \langle I', I \cup J \cup J'\rangle \text{ (because } \text{EX}(I', I \cup J') \cap \text{VAL}(J) = \varnothing),$$

$$\langle I \cup J \cup J', I \cup J\rangle, \langle I', I \cup J \cup J'\rangle \vdash_{\text{TD7}} \langle I \cup J \cup I' \cup J', I \cup J\rangle, \text{ and}$$

$$\langle I \cup J \cup I' \cup J', I \cup J\rangle \vdash_{\text{TD3}} \langle I', I \cup J\rangle.$$

The reader can verify that there is such a constant c as claimed above. Moreover, our systems can each simulate the other with only a linear increase in the length of derivations.

5. Formal systems for subclasses. An *embedded multivalued dependency* (emvd) [Fag1] is a tgd $\langle \{w\}, \{w_1, w_2\}\rangle$ such that, for all $A \in U$, if $w_1[A] = w_2[A]$ then $w[A] = w_1[A]$. It is usually written $X \twoheadrightarrow Y|Z$, where $X = \{A: w_1[A] = w_2[A]\}$, $Y = \{A: w[A] = w_1[A]\}$, and $Z = \{A: w[A] = w_2[A]\}$. If VAL $(w) \subseteq$ VAL $(\{w_1, w_2\})$ then it is an mvd (defined in § 2.3).

Emvd's have attracted a great deal of interest. It is not known whether the implication problem for this subclass is solvable or not. While many inference rules for emvd's are known ([BV1], [Sc], [TKY1], [TKY2]), no sound and complete formal system for them is known.[5] Thus, attention has shifted to finding minimal classes of dependencies that include the class of emvd's as a proper subclass and for which a sound and complete system can be found. Two such classes are the classes of template and binary dependencies which are studied in the following section.

5.1. Template and binary dependencies. A *template dependency* (td) [SU1] is a tgd of the form $\langle \{w\}, I\rangle$. The class of td's contain the class of emvd's and it also contains the class of *embedded join dependencies* of [MMS] and the class of *projected join dependencies* of [YP]. The implication problem for td's is known to be unsolvable [GL], [Va4]. (In fact, it is shown in these papers that even for projected join dependencies the problem is unsolvable). We show now that both systems T_1 and T_2 are sound and complete for td's.

THEOREM 5.1. *The systems T_1 and T_2 are sound and complete for td's.*

Proof. The systems are obviously sound for td's. To show completeness suppose that $D \models \langle I', I\rangle$, where $I' = \{w\}$. Studying the derivation by T_1 constructed in the proof of Theorem 4.1, we see that every dependency in the derivation is either of the form $\langle I', J_k\rangle$ or $\langle K', K\rangle$ from D. In any case it is a td. Studying the derivation by T_2 constructed in the proof of Theorem 4.2, we see that every dependency in the derivation is either of the form $\langle I', I_k\rangle$ or $\langle K', K\rangle$ from D. In any case it is a td. □

Sadri and Ullman [SU1] have independently developed a formal system for td's, which turns out to be the system T_2 restricted to td's.

A *binary dependency* (bd) is a tgd $\langle I', I\rangle$, where $|I| \leqq 2$. The class of bd's contains the class of emvd's, and it also contains the class of *subset dependencies* of [SW]. It is not known whether the implication problem for this class is solvable or not. None of our systems is complete for bd's, but we can combine rules TD2 and TD7 and get a sound and complete system for bd's.

Let T_4 be the system {TD0′, TD1, TD3, TD6, TD7′}, where TD7′ is:

TD7′ (transitivity). $\langle J', J\rangle, \langle I' \cup J, I\rangle \vdash \langle I' \cup J' \cup J, I\rangle$, if EX $(J', J) \cap$ VAL $(I \cup I') = \varnothing$.

THEOREM 5.2. *The system T_4 is sound and complete for tgd's and for bd's.*

Proof. Left to the reader. □

5.2. Embedded multivalued dependencies. The class of emvd's lies in the intersection of the class of td's and the class of bd's.[6] There is however a very important distinction between td's and bd's on one hand and emvd's on the other. For any fixed universe U there are infinitely many nonequivalent bd's or td's, but only a finite number of nonisomorphic emvd's. ($\langle I', I\rangle$ is isomorphic to $\langle J', J\rangle$ if there is a one-to-one valuation h such that $h(I') = J'$ and $h(I) = J$.) Thus, for a fixed universe U there is a finite number of instances of the implication problem and a finite number of possible

[5] In contrast, for mvd's the implication problem is solvable ([Beer]), and a sound and complete formal system does exist ([BFH]).

[6] This intersection is exactly the class of subset dependencies of [SW].

derivations, so the implication problem is solvable and there is a sound and complete formal system. (That does not mean that we can effectively find the implication testing algorithm and the formal system for any given U). What we would like to have are *uniform* algorithm and system, i.e., an algorithm and a system which are "good" for any U. If one considers only bounded inference rules in the style of [Arm], [BFH] then it is known that there is no uniform sound and complete system for emvd's [PP], [SW]. Nevertheless, it is still possible that we can find a sound and complete system using unbounded rules like rule TTD4 or rule JD5 of [BV1], [Va1]. The following theorem says that finding a uniform implication testing algorithm is equivalent to finding a uniform sound and complete formal system.

THEOREM 5.3. *The implication problem for emvd's is solvable if and only if there is a sound and complete formal system for emvd's.*

Proof.

Only if. Suppose that the implication problem for emvd's is solvable, and consider the formal system consisting of one inference rule:

$$d_1, \cdots, d_k \vdash d, \quad \text{if } \{d_1, \cdots, d_k\} \models d.$$

Clearly, this formal system is sound and complete for emvd's.

If. Suppose that F is a sound and complete formal system for emvd's. Let D and d over a universe U be given. To decide whether $D \models d$ we list every possible sequence of emvd's $d_1, \cdots, d_m$ and check whether it is a derivation of d from D by F. Inasmuch as there is a finite number of nonisomorphic emvd's over U, this process must terminate. Hence, the implication problem for emvd's is solvable. □

Remark. The same argument holds for the subsets dependencies of [SW], for the embedded join dependencies of [MMS], and for projected join dependencies of [YP]. Since the implication problem for projected join dependencies is unsolvable [GL], [Va4], this class can not have a sound and complete formal system. There is, however, a subtle point here. The syntax used here is such that there is no need to specify the universe U explicitly, because it is evident from the syntax. Projected join dependencies, embedded join dependencies, subset dependencies, and embedded multivalued dependencies were all introduced originally in a different syntax, in which the universe is not evident (see for example the syntax described in the beginning of § 5). When we study formal systems for such a syntax, it is crucial to know how the formal system handles that lack of explicitness, because that may affect whether a class of dependencies has or does not have a sound and complete formal system. We refer the reader to [Va4] for a more thorough discussion of this point.

5.3. Decomposition of ttgd's. An *embedded join dependency* (ejd) [MMS] is a td $\langle\{w\}, I\rangle$ such that for every $A \in U$ and two distinct tuples u and v in I, if $u[A] = v[A]$ then $w[A] = u[A]$. If $|I| = 2$ then it is an emvd, if VAL $(w) \subseteq$ VAL (I), then it is a *join dependency* (jd) [ABU], [Riss], and if VAL $(w) \subseteq$ VAL (I) and $|I| = 2$ then it is an mvd. In [BV1], [MM], [Va1] it is shown that every jd is equivalent to a set consisting of one ejd and several mvd's. That is, a jd can be "decomposed" into weaker dependencies. In this section we provide a decomposition theorem for ttgd's, which implies the above result as a special case.

Let $\langle w, I\rangle$ be a ttgd. The decomposition is based upon a distinction between two kinds of values in I: the values which are repeated in I and those that are nonrepeated in I. Let $u \in I$ and $A \in U$. $u[A]$ is repeated in I if there is another tuple $v \in I$ such that $u[A] = v[A]$.

Let REP (I) be the set

$$\{A\colon \text{for some } u \in I,\ u[A] \text{ is repeated in } I\}.$$

For any tuple u and a set $X \subseteq U$, *let* u_X be a tuple such that $u_X[X] = u[X]$ and $u[A]$ is a new distinct value for all $A \in \bar{X}$. Suppose that $I = \{w_1, \cdots, w_m\}$. With each tuple w_i we associate two sets $Y_i = \{A\colon w[A] = w_i[A]\}$ and $X_i = Y_i - \text{REP}(I)$, and a ttgd $\langle u_i, I' \rangle$, where $I' = I \cup \{w_{\text{REP}(I)}\}$, $u_i[X_i] = w_i[X_i]$, and $u_i[\bar{X}_i] = w_{\text{REP}(I)}[\bar{X}_i]$. $X_1, \cdots, X_m$ is a partition of $\overline{\text{REP}(I)}$.

Example 3. Let $U = ABCDEF$, and let $\langle w, I \rangle$ be the ttgd $\langle w, \{w_1, w_2, w_3\} \rangle$:

	A	B	C	D	E	F
w:	$a0$	$b0$	$c0$	$d0$	$e0$	$f0$
w_1:	$a0$	$b1$	$c1$	$d0$	$e1$	$f1$
w_2:	$a1$	$b0$	$c1$	$d1$	$e0$	$f2$
w_3:	$a1$	$b1$	$c0$	$d2$	$e2$	$f0$

Now we have REP $(I) = ABC$, $Y_1 = AD$, $X_1 = D$, $Y_2 = BE$, $X_2 = E$, $Y_3 = CF$, and $X_3 = F$. $w_{\text{REP}(I)}$ is the tuple

A	B	C	D	E	F
$a0$	$b0$	$c0$	$d3$	$e3$	$f3$

u_1, u_2, and u_3 are the tuples

	A	B	C	D	E	F
u_1:	$a0$	$b0$	$c0$	$d0$	$e3$	$f3$
u_2:	$a0$	$b0$	$c0$	$d3$	$e0$	$f3$
u_3:	$a0$	$b0$	$c0$	$d3$	$e3$	$f0$

THEOREM 5.4. *Let* $\langle w, I \rangle$ *be a ttgd,* $I = \{w_1, \cdots, w_m\}$, *and let* $D = \{\langle \{w_{\text{REP}(I)}\}, I \rangle, \langle u_1, I' \rangle, \cdots, \langle u_m, I' \rangle\}$. Then $\langle w, I \rangle \models D$ and $D \models \langle w, I \rangle$.

Proof. We show that $\langle w, I \rangle \vdash D$ and $D \vdash \langle w, I \rangle$.

$\langle w, I \rangle \vdash D$: $\langle w, I \rangle \vdash \langle \{w_{\text{REP}(I)}\}, I \rangle$ by TD6. Define a valuation h on I such that $h(w_i) = w$ and $h(w_j) = w_{\text{REP}(I)}$ for $j \neq i$. Now $h(w) = u_i$, so by TD1, $\langle w, I \rangle \vdash \langle u_i, \{w, w_{\text{REP}(I)}\} \rangle$, and by TD4, $\langle w, I \rangle \vdash \langle u_i, I' \rangle$.

$D \vdash \langle w, I \rangle$: We define a sequence of tuples $v_0, \cdots, v_m$ as follows. v_0 is $w_{\text{REP}(I)}$, $v_{i+1}[X_{i+1}] = w_{i+1}[X_{i+1}]$, and $v_{i+1}[\overline{X_{i+1}}] = v_i[\overline{X_{i+1}}]$. Observe that $v_m = w$. We show by induction on i that $D \vdash \langle I \cup \{v_i\}, I \rangle$.

Basis $(i = 0)$. Since $v_0 = w_{\text{REP}(I)}$, and by TD0, $\vdash \langle I, I \rangle$, we have that $\langle \{w_{\text{REP}(I)}\}, I \rangle \vdash \langle I \cup \{v_0\}, I \rangle$ by TD7.

Induction. Suppose that $D \vdash \langle I \cup \{v_i\}, I \rangle$. Let h be a valuation such that $h(w_{\text{REP}(I)}[\bigcup_{j=1}^{i} X_j]) = w[\bigcup_{j=1}^{i} X_j]$ and h is the identity elsewhere. Then $h(w_{\text{REP}(I)}) = v_i$ and $h(u_{i+1}) = v_{i+1}$. By TD1, $\langle u_{i+1}, I' \rangle \vdash \langle v_{i+1}, I \cup \{v_i\} \rangle$, and by TD7 and TD3, $\langle I \cup \{v_i\}, I \rangle, \langle v_{i+1}, I \cup \{v_i\} \rangle \vdash \langle I \cup \{v_{i+1}\}, I \rangle$.

It follows that $D \vdash \langle I \cup \{w\}, I \rangle$, and by TD3, $D \vdash \langle w, I \rangle$. □

Note that the dependencies in D are "weaker" than $\langle w, I \rangle$ because they are implied by $\langle w, I \rangle$, but do not, in general, imply $\langle w, I \rangle$.

We show now how Theorem 5.5 implies the above mentioned decomposition of jd's. An ejd $\langle \{w\}, \{w_1, \cdots, w_m\} \rangle$ is also written as $*[R_1, \cdots, R_m]$, where $R_i = \{A\colon w[A] = w_i[A]\}$. Thus, if $\langle w, I \rangle$ is the jd $*[Y_1, \cdots, Y_m]$, then $\langle \{w_{\text{REP}(I)}\}, I \rangle$ is the

ejd $*[Y_1 \cap \mathrm{REP}(I), \cdots, Y_m \cap \mathrm{REP}(I)]$. That the ttgd $\langle u_i, I' \rangle$ is equivalent to an mvd is less trivial.

LEMMA 5.1. $\langle u_i, I' \rangle \vDash Y_i \cap \mathrm{REP}(I) \twoheadrightarrow Y_i$ and $Y_i \cap \mathrm{REP}(I) \twoheadrightarrow Y_i \vDash \langle u_i, I' \rangle$.

Proof. Consider the ttgd $\langle u_i, \{w_i, w_{\mathrm{REP}(I)}\} \rangle$. We have $\{A: w_i[A] = w_{\mathrm{REP}(I)}[A]\} = Y_i \cap \mathrm{REP}(I)$ and $\{A: u_i[A] = w_i[A]\} = Y_i$, so this ttgd is the mvd $Y_i \cap \mathrm{REP}(I) \twoheadrightarrow Y_i$. We show now that $\langle u_i, I' \rangle \vdash \langle u_i, \{w_i, w_{\mathrm{REP}(I)}\} \rangle$ and $\langle u_i, \{w_i, w_{\mathrm{REP}(I)}\} \rangle \vdash \langle u_i, I' \rangle$.

$\langle u_i, \{w_i, w_{\mathrm{REP}(I)}\} \rangle \vdash \langle u_i, I' \rangle$ by TD2. To show the opposite direction, let h be a valuation such that $h(w_i) = w_i$ and $h(w_j) = w_{\mathrm{REP}(I)}$, for $j \neq i$. (Such h can be defined because $\langle w, I \rangle$ is a jd.) Now $h(u_i) = u_i$ and $h(I') = \{w_i, w_{\mathrm{REP}(I)}\}$, so by TD1, $\langle u_i, I' \rangle \vdash \langle u_i, \{w_i, w_{\mathrm{REP}(I)}\} \rangle$. □

6. Concluding remarks. Our model is rather restricted since it assumes that the database consists of one relation,[7] and that different attributes have disjoint underlying domains, the so-called "many-sorted" case. While these assumptions offer theoretical advantages [BBG], [Fag2], they are dubious from a practical point of view. It happens that our formal systems of § 4 can be very easily extended to the general case of many relations and nondisjoint domains by simply adjoining a relation name to each tuple. In view of this we think that claims to the naturalness of the universal many-sorted case that are based on its having a sound and complete formal system are not very convincing.

Another dubious assumption is that a relation can have an infinite set of tuples. Since a database is inherently finite, there is a strong justification to define a relation as a finite set of tuples. We say that a set of dependencies D *finitely implies* a dependency d, denoted $D \vDash_f d$, if d is satisfied by every finite relation which satisfies all dependencies in D. Unfortunately, it is easy to see that the set $\{\langle D, d \rangle: D \nvDash_f d\}$ is recursively enumerable. It follows that if the set $\{\langle D, d \rangle: D \vDash_f d\}$ is not recursive, then it is not even recursively enumerable. Since the finite implication problem for tgd's is unsolvable [BV2], [CLM], there can be no sound and complete formal system for finite implication. In contrast, if all tgd's in D are total then $D \vDash d$ iff $D \vDash_f d$ [BV2], [CLM]. Thus, our formal systems for ttgd's and egd's in § 3 are systems for finite implications as well as for implication.

In § 5.3 we have observed that if the implication problem for a class of dependencies is solvable then this class has a sound and complete formal system. Even in this case there is still an interest in finding an "elegant" formal system, one which has a small number of simple axioms and (preferably bounded) inference rules. A typical example is the propositional calculus, which is a formal system for the resursive set of tautologies of propositional logic. Likewise, the implication problem for jd's is solvable, but we would like to have an elegant formal system for that class or for a minimal class of ttgd's containing it. Another case of interest is that of implication of mvd's by ttgd's and egd's, which is probably the most general case for which an efficient implication testing algorithm does exist [BV3]. These cases will be dealt with in future papers.

Finally, since our dependencies are equivalent to first-order sentences, it is interesting to know what is the relationship between our formal systems and the known formal systems for first-order logic. It turns out that there is indeed a very strong connection between our systems and the system of resolution and paramodulation [CL]. This connection will be described in a future paper.

[7] This assumption is usually called "the universal relation assumption".

REFERENCES

[ABU] A. V. AHO, C. BEERI AND J. D. ULLMAN, *The theory of joins in relational databases*, ACM Trans. Database Systems, 4 (1979), pp. 297–314.

[Arm] W. W. ARMSTRONG, *Dependency structure of database relationships*, Proc. IFIP 74, North-Holland, Amsterdam, 1974, pp. 580–583.

[ASU] A. V. AHO, Y. SAGIV AND J. D. ULLMAN, *Equivalence among relational expressions*, this Journal, 8 (1979), pp. 218–246.

[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.

[BBG] C. BEERI, P. A. BERNSTEIN AND N. GOODMAN, *A sophisticate's introduction to database normalization theory*, Proc. International Conference on VLDB, Berlin, 1978, pp. 113–124.

[Beer] C. BEERI, *On the membership problem for multivalued dependencies*, ACM Trans. Database Systems, 5 (1980), pp. 241–259.

[Bern] P. A. BERNSTEIN, *Synthesizing third normal form relations from functional dependencies*, ACM Trans. Database Systems, 1 (1976), pp. 277–298.

[BFH] C. BEERI, R. FAGIN AND J. H. HOWARD, *A complete axiomatization for functional and multivalued dependencies in database relations*, Proc. ACM Conference on Management of Data, Toronto, 1977, pp. 47–61.

[Bir] G. BIRKHOFF, *On the structure of abstract algebras*, Proc. Cambridge Phil. Soc., 31 (1935), pp. 433–454.

[BV1] C. BEERI AND M. Y. VARDI, *On the properties of join dependencies*, in Advances in Database Theory, H. Gallaire, J. Minker and J. M. Nicolas, eds., Plenum, New York, 1981, pp. 25–72.

[BV2] ———, *The implication problem for data dependencies*, Proc. 8th ICALP, Acre, Israel, 1981, in Lecture Notes in Computer Science 115, Springer-Verlag, Berlin, 1981, pp. 73–85; also Technical Report, Dept. Computer Science, Hebrew University of Jerusalem, May 1980.

[BV3] ———, *A proof procedure for data dependencies*, Technical Report, Dept. Computer Science, Hebrew University of Jerusalem, Dec. 1980.

[BV4] ———, *On the complexity of testing implications of data dependencies*, Technical Report, Dept. Computer Science, Hebrew University of Jerusalem, Dec. 1980.

[CL] C. L. CHANG AND C. R. T. LEE, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[CLM] A. K. CHANDRA, H. R. LEWIS AND J. A. MAKOWSKY, *Embedded implicational dependencies and their inference problem*, Proc. 13th ACM Annual Symposium on Theory of Computing, 1981, pp. 342–354; J. Comput. Systems Sci., to appear.

[Codd] E. F. CODD, *Further normalization of the database relational model*, in Database Systems, R. Rustin, ed., Prentice-Hall, 1972, Englewood Cliffs, NJ, pp. 33–64.

[Del] C. DELOBEL, *Semantics of relations and the decomposition process in the relational data model*, ACM Trans. Database Systems, 3 (1978), pp. 201–222.

[Fag1] R. FAGIN, *Multivalued dependencies and a new normal form for relational databases*, ACM Trans. Database Systems, 2 (1977), pp. 262–278.

[Fag2] ———, *Horn clauses and database dependencies*, J. Assoc. Comput. Mach., 29 (1982), pp. 252–285.

[FMUY] R. FAGIN, D. MAIER, J. D. ULLMAN AND M. YANNAKAKIS, *Tools for template dependencies*, this Journal, 12 (1983), pp. 36–59.

[GL] Y. GUREVICH AND H. R. LEWIS, *The inference problem for template dependencies*, Proc. ACM Symposium on Principles of Database Systems, Los Angeles, 1982, pp. 221–229.

[HW] L. HENSCHEN AND L. WOS, *Unit refutation and Horn sets*, J. Assoc. Comput. Mach., 21 (1974), pp. 590–605.

[Ma] D. MAIER, *Minimum covers in the relational database model*, J. Assoc. Comput. Mach., 27 (1980), pp. 664–674.

[MM] D. MAIER AND A. O. MENDELZON, *Generalized mutual dependencies and the decomposition of database relations*, Proc. International Conference on VLDB, Rio de Janiero, 1979, pp. 75–82.

[MMS] D. MAIER, A. O. MENDELZON AND Y. SAGIV, *Testing implications of data dependencies*, ACM Trans. Database Systems, 2 (1977), pp. 201–222.

[Nic] J. M. NICOLAS, *First order logic formalization for functional, multivalued and mutual dependencies*, Proc. ACM SIGMOD Conference on Management of Data, 1978, pp. 40–46.

[Pa] J. PAREDAENS, *A universal formalism to express decompositions, functional dependencies and other constraints in a relational database*, Technical Report, University of Antwerp, 1980; also Theoret. Comput. Sci., 19 (1982), pp. 143–160.

[PJ] J. PAREDAENS AND D. JANSSENS, *Decompositions of relations—a comprehensive approach*, in Advances in Database Theory, H. Gallaire, J. Minker and J. M. Nicolas, eds., Plenum, New York, 1981, pp. 73–100.

[PP] D. S. PARKER, JR. AND K. PARSAYE-GHOMI, *Inferences involving embedded multivalued dependencies*, Proc. ACM SIGMOD Conference on Management of Data, Los Angeles, 1980, pp. 52–57.

[Riss] J. RISSANEN, *Theory of relations for databases—a tutorial survey*, Proc. 7th Symp. on Mathematical Foundations of Computer Science, 1978, Lecture Notes in Computer Science 64, Springer-Verlag, Berlin, pp. 537–551.

[Sc] E. SCIORE, *A complete axiomatization of full join dependencies*, J. Assoc. Comput. Mach., 29 (1982), pp. 373–393.

[Sel] A. SELMAN, *Completeness of calculi for axiomatically defined classes of algebras*, Algebra Universalis, 2 (1972), pp. 20–32.

[SU1] F. SADRI AND J. D. ULLMAN, *Template dependencies—A large class of dependencies in relational databases and its complete axiomatization*, J. Assoc. Comput. Mach., 29 (1982), pp. 363–372.

[SU2] ———, *The theory of functional and template dependencies*, Theoret. Comput. Sci., 17 (1982), pp. 317–332.

[SW] Y. SAGIV AND S. WALECKA, *Subset dependencies and a completeness result for a subclass of embedded multivalued dependencies*, J. Assoc. Comput. Mach., 29 (1982), pp. 103–117.

[TKY1] K. TANAKA, Y. KAMBAYASHI AND S. YAJIMA, *On the representability of decompositional schema design with multivalued dependencies*, Technical Report, Kyoto University, 1979.

[TKY2] ———, *Properties of embedded multivalued dependencies in relational databases*, Trans. IECE of Japan, E62, 1979.

[Va1] M. Y. VARDI, *Axiomatization of functional and join dependencies in the relational model*, M.Sc. Thesis, Weizmann Institute of Science, Rehovot, Israel, 1980.

[Va2] ———, *Inferring multivalued dependencies from functional and join dependencies*, Technical Report, Dept. Applied Science, Weizmann Institute of Science, Rehovot, Israel, 1980; Acta Informatica, to appear.

[Va3] ———, *The implication problem for data dependencies in relational databases*, Ph.D. thesis, Hebrew University of Jerusalem, 1981.

[Va4] ———, *The implication and the finite implication problem for typed template dependencies*, Proc. ACM Symposium on Principles of Database Systems, Los Angeles, 1982, pp. 230–238, revised, Technical Report STAN-CS-82-912, Dept. Computer Science, Stanford University, Stanford, CA, May 1982; J. Comput. Systems Sci., to appear.

[YP] M. YANNAKAKIS AND C. PAPADIMITRIOU, *Algebraic dependencies*, 21st IEEE Ann. Symp. on Found. of Computer Science, 1980, pp. 328–332; J. Comput. Systems Sci., 21 (1982), pp. 2–41,

[Zan] C. ZANIOLO, *Analysis and design of relational schemata for database systems*, Technical Report UCLA-ENG-7769, UCLA, 1976.

SIAM J. COMPUT.
Vol. 13, No. 1, February 1984

© 1984 Society for Industrial and Applied Mathematics
0097-5397/84/1301-0007 $01.25/0

THE TRAVELING SALESMAN PROBLEM WITH MANY VISITS TO FEW CITIES*

STAVROS S. COSMADAKIS† AND CHRISTOS H. PAPADIMITRIOU‡

Abstract. We study the version of the traveling salesman problem in which a relatively small number of cities—say, six—must be visited a huge number of times—e.g., several hundred times each. (It costs to go from one city to itself.) We develop an algorithm for this problem whose running time is exponential in the number of cities, but logarithmic in the number of visits. Our algorithm is a practical approach to the problem for instances of size in the range indicated above. The implementation and analysis of our algorithm give rise to a number of interesting graph-theoretic and counting problems.

Key words. traveling salesman problem, dynamic programming, assignment problem, transportation problem, minimal Eulerian digraph, feasible sequence, Stirling's formula, Stirling numbers of the second kind, min-cost max-flow problem, Edmonds–Karp scaling method

1. Introduction. In this paper we study the following version of the traveling salesman problem (TSP): We are given n cities, an $n \times n$ distance matrix d_{ij} (not necessarily symmetric or with zero diagonal elements), and n integers $k_1, \cdots, k_n > 0$. We are asked to find the shortest walk that visits the first city k_1 times, the second city k_2 times, and so on. (We are allowed to visit city i twice in a row, but this costs us d_{ii}.)

This problem, which we call the *many-visits* TSP, is obviously a generalization of the TSP (the TSP is our problem in the special case in which all k_i's are equal to 1). It can also be considered as a *special case* of the TSP (more precisely, a nonstandard representation of the TSP), in which clusters of k_i cities with identical rows and columns are treated as a single city to be visited k_i times. The many-visits TSP arises in connection to the applications of the TSP in scheduling. In such applications, the cities are in fact *tasks* to be executed, and d_{ij} reflects the *overhead* associated with the task j immediately following task i. Now, in certain applications, each task belongs to one of a few *types*, and tasks of the same type have identical characteristics. For example, in the scheduling of airplane landings, there could only be four types of tasks—e.g., regular, jumbo, private, and military airplanes—but several dozens of each may be pending at each time for landing. There is a certain delay between the landing of an aircraft and the landing of the next aircraft, depending on the types of the two airplanes. We wish to minimize the total delay. In the many-visits formulation of such a problem n would be 4, while the k_i's would be the number of airplanes of each type.

The many-visits TSP can be solved by extending the dynamic programming approach of [HK]; see [Ps] and § 2.3 of this paper. This algorithm, however, requires time proportional to $n^2 \prod (k_i + 1)$. For $n = 5$, for example, this is already prohibitive when the k_i's are as small as 10. In this paper we present a drastically different approach to the many-visits TSP, which results in an algorithm with running time $O(e(n) \log (\sum k_i))$, where $e(n)$ is a moderately growing exponential function of n. For reasonably small values of n—say, up to 10—our algorithm brings into the realm of realistic solution instances with virtually unlimited k_i's. As evidenced by the running

* Received by the editors June 28, 1982, and in revised form February 11, 1983.

† Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. This research was based in part on this author's Bachelor's Thesis at Massachusetts Institute of Technology.

‡ Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. The research of this author was supported in part by the National Science Foundation under grant MCS79-08965.

time of our algorithm, which is sublinear in the k_i's, the output is not the optimal walk itself, but a list of the numbers of times that each edge (i, j) participates in the optimal walk. Naturally, since for $k_i = 1$ our problem becomes the ordinary TSP, this exponential dependence on n is expected (and most probably inherent).

We shall now outline our approach. It has been one of the basic and oldest observations in the area, that the TSP can be decomposed into two problems: the *assignment problem* [Ku], [La], [PS], whose solution guarantees that each city is visited and departed from exactly once, and a *connectivity problem*, which forbids "subtours" in the solution. The first problem is easy, so the hard part of the TSP is enforcing connectivity. Many branch-and-bound algorithms [Ch], algorithms for special cases [GG3], and heuristics [Ka] are based on this decomposition. Our algorithm is based on the following very simple idea: In the many-visits version of the TSP the first problem becomes only a little harder (namely, the *transportation problem* [EK], [PS]), whereas the connectivity aspect becomes much easier, in the sense that it is a problem of size n, and therefore can be solved exhaustively if n is small—and this is our working hypothesis.

More specifically, we can restate the many-visits TSP as follows: Given an $n \times n$ distance matrix d_{ij} and n integers $k_1, \cdots, k_n$, find the shortest Eulerian directed graph with n nodes and with indegrees $k_1, \cdots, k_n$ in the corresponding nodes. Now an Eulerian digraph must be *strongly connected* and it must also be *balanced*, that is, it must satisfy at each node i indegree $(i) =$ outdegree (i). An Eulerian digraph that has no other Eulerian digraph as a proper subgraph is called *minimal*. So, a solution of the many-visits TSP can be decomposed into a minimal Eulerian digraph (this is the connectivity part) and a balanced (but possibly disconnected) digraph to bring the degrees up to the required levels of the k_i's (this is the transportation problem). The fortunate fact is that *there is a fixed number of minimal Eulerian graphs on n nodes*, independent of the k_i's. Our basic algorithm is now apparent:

1. Repeat the following step for each minimal Eulerian graph G on n nodes:
2. Let $\delta_1, \cdots, \delta_n$ be the sequence of indegrees of G. Solve the transportation problem with distance matrix d_{ij} and both capacities and requirements equal to $k_1 - \delta_1, \cdots, k_n - \delta_n$. Superpose the solution to G.
3. Among the Eulerian graphs thus generated, pick the cheapest.

Step 1, generating all minimal Eulerian graphs, can be done in a computationally feasible way only by employing some interesting graph theory, and using dynamic programming. We discuss this in § 2. In § 3 we make some calculations that are necessary for the analysis of the algorithm, solving some counting problems that are interesting in their own right. Finally, in § 3 we also outline a modification of the algorithm, which replaces the repeated solutions of the transportation problem in step 2 above by the precomputation of the solution to a "master" problem, plus the solution of (much smaller) incremental problems. This modification reduces the computational complexity from $e(n)+e'(n)\log(\sum k_i)$ to $e(n)+n^3\log(\sum k_i)+e'(n)\log n$, where $e(n)$ and $e'(n)$ are exponential function of n, specified in § 3.

2. Generating minimal Eulerian graphs.

2.1. A reduction. It is not at all clear how to implement the first step of our algorithm, i.e., enumerating all minimal Eulerian graphs on n nodes. In fact, the outlook is very bleak, because of the following rather surprising result, proven recently in [PY]:

THEOREM 1. *Testing whether a digraph is minimal Eulerian is* coNP-*complete.*

Fortunately, with a little thought we can circumvent this difficulty. Suppose that

two minimal Eulerian digraphs G and G' generated in step 1 have the same *indegree sequence* $(\delta_1, \cdots, \delta_n)$. Then the same transportation problem is solved for both in step 2. Therefore, we need only consider the *cheaper* (under d_{ij}) digraph among G, G'. Hence, for each sequence of integers which is the indegree sequence of a minimal Eulerian digraph (hereafter called a *feasible* sequence) we may compute the cheapest Eulerian digraph with this indegree sequence. If the resulting digraph is minimal, we proceed to step 2. If it is not, it can be discarded: The final solution corresponding to it will certainly be considered when we consider the indegree sequence of the minimal Eulerian digraph, which is necessarily a subgraph of the present nonminimal one. Of course, by Theorem 1, we cannot test efficiently each resulting Eulerian digraph for minimality. To improve the efficiency of our algorithm in practice, we could use a reasonably fast heuristic that detects some obvious nonminimal digraphs.

Thus we have reduced step 1 to the following two substeps:

1.1. Generate all *feasible indegree sequences* of length n.

1.2. For each such sequence, find the cheapest Eulerian graph G that has as an indegree sequence the given one.

We examine each of these substeps separately.

2.2. Feasible degree sequences. Surprisingly, although minimal Eulerian graphs are hard to recognize (Theorem 1), their degree sequences have a nice characterization:

THEOREM 2. *$(\delta_1, \cdots, \delta_n)$ is a feasible degree sequence iff it has at least* $\max \delta_i$ *1's in it.*

We prove the two directions separately.

LEMMA 1. *Let $G=(V,E)$ be a minimal Eulerian digraph, and suppose* indegree $(\nu_0)=k$ *for some $\nu_0 \in V$. Then there are at least k vertices with degree* 1 *in G.*

Proof. The lemma is obvious for $k=1$. To prove it in general, consider a set $\mathbf{C}$ of cycles whose union is G (by cycle we always mean simple directed cycle, i.e. directed cycle that does not repeat any vertex; any Eulerian digraph can be thought of, although not necessarily in a unique way, as the disjoint union of several cycles). Now construct the following finite sequence $\langle G_i \rangle$ of partial sub-digraphs of G (partial because the vertex set of each of them, with the exception of the last one, is a proper subset of V): each G_i is going to be the union of certain cycles in $\mathbf{C}$. G_0 is the union of the k cycles in $\mathbf{C}$ that contain the vertex ν_0 (for every $\nu \in V$, indegree (ν) is equal to the number of cycles in $\mathbf{C}$ that contain v). Once G_i has been constructed, $i \geqq 0$, then G_{i+1} is constructed as follows: If there are elements of $\mathbf{C}$ that have not yet been used, at least one of them must contain some vertex in G_i (else G would not be connected); pick such an element of $\mathbf{C}$ and add it to G_i to get G_{i+1}. Let $G_0, \cdots, G_m$ be a sequence that can be constructed in this way ($G_m = G$). Let $R(G_i)$, $i=0, \cdots, m$ be the property that G_i contains at least k cycles in $\mathbf{C}$ each of which satisfies the following:

(i) It contains a vertex with degree 1 in G_i.

(ii) The remaining cycles in $\mathbf{C}$ that make up G_i form a connected partial sub-digraph of G_i. (G_i may also contain cycles in $\mathbf{C}$ that do not satisfy either (i) or (ii); $R(G_i)$ says that at least k of the cycles in $\mathbf{C}$ that G_i contains satisfy both (i) and (ii).)

We shall show by induction that $R(G_i)$ is true for all i, $i=0, \cdots, m$. First, we show that $R(G_0)$ is true: G_0 contains exactly k cycles in $\mathbf{C}$, and each of these satisfies (ii) (since the remaining cycles have a common vertex, namely i_0). But also each of these cycles satisfies (i), because if one of them, say C_j, does not, then each vertex in C_j also belongs to some other cycle among the cycles that make up G_0; thus, by removing C_j from G_0 (i.e. by removing the arcs in C_j) we are left with a connected sub-digraph of G_0. Consequently, by removing C_j from G we obtain an Eulerian

proper sub-digraph of G, which contradicts our hypothesis that G is minimal Eulerian.

Suppose now that $R(G_i)$, $i \geqq 0$, is true, i.e. at least k of the cycles in $\mathbf{C}$ that G_i contains satisfy both (i) and (ii); call these cycles $C_1, \cdots, C_l$, $l \leqq k$. We shall show that $R(G_{i+1})$ is true. Call C_{l+1} the cycle in $\mathbf{C}$ that was added to G_i to obtain G_{i+1}; observe that C_{l+1} contains a vertex with degree 1 in G_{i+1}, or else we could remove C_{l+1} from G_{i+1} (and G) and obtain a proper Eulerian sub-digraph of G_{i+1} (and a proper Eulerian sub-digraph of G). Now distinguish three cases:

1. C_{l+1} does not have any vertices in common with any of $C_1, \cdots, C_l$. Then $R(G_{i+1})$ is true, since $C_1, \cdots, C_l$ satisfy both (i) and (ii) in G_{i+1}.

2. C_{l+1} has common vertices with only one of $C_1, \cdots, C_l$, say with C_j. Then C_{l+1} and any of $C_1, \cdots, C_l$ except possibly C_j satisfy both (i) and (ii) in G_{i+1}, so again $R(G_{i+1})$ is true.

3. C_{l+1} has common vertices with $C_{j_1}, \cdots, C_{j_h}$, $1 \leqq j_1, \cdots, j_h \leqq l$, $h > 1$. Then for $p \neq j_r$, $r = 1, \cdots, h$, C_p, $1 \leqq p \leqq l$, satisfies both (i) and (ii) in G_{i+1}. Consider now C_{j_r}, $1 \leqq r \leqq h$; it clearly satisfies (ii) in G_{i+1}, since it satisfies it in G_i and C_{l+1} has at least one common vertex with C_{j_s}, $s \neq r$. But then C_{j_r} also satisfies (i) in G_{i+1}, since otherwise we could remove it and obtain a proper Eulerian sub-digraph of G. Therefore, $R(G_{i+1})$ is true. Since cases 1–3 exhaust all possibilities, the inductive proof is complete.

It follows that $R(G_m)$, i.e. $R(G)$, is true; but this means that G has at least k vertices with degree 1, and we are done. □.

LEMMA 2. *Let* $(\delta_1, \cdots, \delta_n)$ *be a sequence of integers such that there are at least* $\max \delta_i$ 1'*s in it. Then it is a feasible degree sequence.*

Proof. Given such a sequence $(\delta_1, \cdots, \delta_n)$, we shall construct a minimal Eulerian graph G with degree sequence $(\delta_1, \cdots, \delta_n)$. First, suppose that the number of 1's is exactly equal to the largest δ, say k. Without loss of generality $k = \delta_1 \geqq \delta_2 \geqq \cdots \geqq \delta_{n-k} > \delta_{n-k+1} = \cdots = \delta_n = 1$. G is constructed as the union of k cycles. Each of the k cycles contains some of the vertices $1, 2, \cdots, n-k$, and a different one among the vertices $n-k+1, \cdots, n$. The δ_{n-k} first cycles are of the form $(1, 2, \cdots, n-k, j, 1)$, where $j > n-k$. The $\delta_{n-k-1} - \delta_{n-k}$ next (possibly 0) are of the form $(1, 2, \cdots, n-k-1, j, 1)$. The $\delta_{n-k-2} - \delta_{n-k-1}$ next are of the form $(1, 2, \cdots, n-k-2, j, 1)$; and so on. Finally, the $\delta_1 - \delta_2$ last are of the form $(1, j, 1)$, for a total of $(\delta_1 - \delta_2) + (\delta_2 - \delta_3) + \cdots + (\delta_{n-k-1} - \delta_{n-k}) + \delta_{n-k} = \delta_1 = k$ cycles, exhausting all k indegree-1 nodes.

The construction is illustrated in Fig. 1 for the sequence (5, 3, 2, 2, 1, 1, 1, 1, 1). It is immediate that (a) each node has the appropriate indegree, and (b) the resulting digraph is minimal Eulerian, since any cycle in it contains an indegree-1 node.

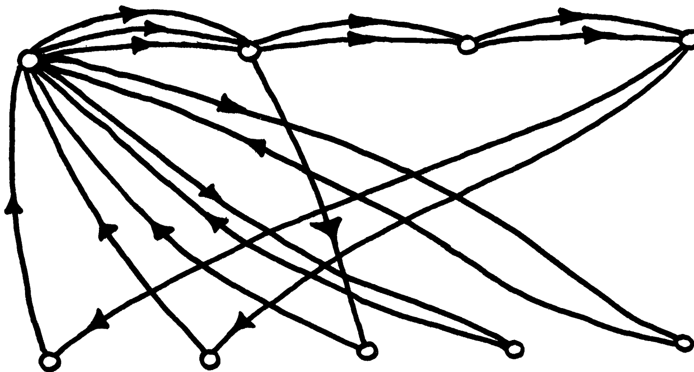


FIG. 1

For the case of more than k 1's among the δ_i's, just insert the superfluous indegree-1 nodes in one of the cycles. □

Theorem 2 follows immediately from the two lemmas.

As a consequence of this characterization, feasible degree sequences of length n can be easily enumerated as follows:

1. For $k = 2, \cdots, n$ repeat step 2.
2. For each sequence $(\delta_1, \cdots, \delta_{n-k})$ with $k \geqq \delta_1 \geqq \cdots \geqq \delta_{n-k} > 1$ repeat step 3.
3. Generate all distinct permutations of $(\delta_1, \cdots, \delta_{n-k}, 1, \cdots, 1)$.

All enumerations implicit in steps 2 and 3 are easy to do.

2.3. Optimal Eulerian graphs. Given a degree sequence $\delta = (\delta_1, \cdots, \delta_n)$, and an $n \times n$ distance matrix d_{ij}, we can use dynamic programming [HK], [Ps] in order to find the shortest Eulerian graph with this degree sequence. For each degree sequence $a \leqq \delta$ (componentwise comparison), and each i, $1 \leqq i \leqq n$, let $C(a; i)$ be the cost of the shortest possible way of starting from city 1, visiting city j a_j times, $j = 1, \cdots, n$, and ending up in city i. We then have the recurrence

$$C(a_1, \cdots, a_n; i) = \min_j [C(a_1, \cdots, a_{i-1}, a_i - 1, a_{i+1}, \cdots, a_n; j) + d_{ji}]$$

with the initial conditions $C(1, 0, \cdots, 0, 1, 0, \cdots, 0; i) = d_{1i}$ (1's in the first and ith position).

Finally, the cost of the optimal Eulerian graph with degree sequence δ is given by

$$C_{\text{opt}} = \min_j [C(\delta, j) + d_{j1}].$$

The straightforward implementation of these recurrences takes time $O(n^2 \prod (\delta_i + 1))$. As usual, we can equally easily recover the optimal Eulerian graph in the same amount of time.

3. Analysis of efficiency.

3.1. Preliminaries. Let $F(n)$ be the set of all feasible degree sequences of n nodes. Also, let us define the quantity

$$\text{DP}(n) = \sum_{(\delta_1, \cdots, \delta_n) \in F(n)} \prod_i (\delta_i + 1).$$

We can analyze the complexity of our algorithm as follows: The algorithm essentially boils down to solving an optimal Eulerian digraph problem, and an $n \times n$ transportation problem with capacities approximately k_i for each degree sequence in $F(n)$. The total effort expended in the dynamic programming algorithm is a small constant times $n^2\text{DP}(n)$. If we use the Edmonds–Karp scaling method for the transportation problem (see [EK] and § 3.3), each such problem takes time $O(n^3 \log (\sum k_i))$ for a total of $O(|F(n)| n^3 \log (\sum k_i))$. We must therefore derive asymptotic estimates for $F(n)$ and $\text{DP}(n)$. This is the subject of the next subsection.

3.2. Counting problems.

PROPOSITION 1. (a) $|F(n)| = \sum_{k=2}^n C(n, k)(k-1)^{n-k}$.

(b) $\text{DP}(n) = \sum_{k=2}^n C(n, k) 2^k [(k-1)(k+4)/2]^{n-k}$.

(Here by $C(n, k)$ we denote the number of ways for choosing k objects among n.)

Proof. (a) Suppose $\delta_i = 1$ exactly for $i = i_m$, where $m = 1, \cdots, k$; each of the other $n-k$ elements can take any value between 2 and k, so there are $(k-1)^{n-k}$ such sequences. For any given k, there are $C(n, k)$ ways to pick $i_1, \cdots, i_k$; also, k can take any value between 2 and n.

(b) Suppose $\delta_i = 1$ exactly for $i = i_m$, where $m = 1, \cdots, k$;

$$\prod_{i=i_m}^{n} (\delta_i + 1) = \prod_{\substack{i=i_m \\ 1 \leqq m \leqq k}} (\delta_i + 1) \prod_{\substack{i \neq i_m \\ \text{for all } m \\ 1 \leqq m \leqq k}} (\delta_i + 1).$$

The first factor is equal to 2^k, and in the second factor $(\delta_i + 1)$ can take any value between 3 and $k + 1$, so

$$\sum_{\substack{\delta \in M(n) \\ \delta_i = 1 \text{ iff } i = i_m \\ 1 \leqq m \leqq k}} \prod_{i=1}^{n} (\delta_i + 1) = 2^k [3 + \cdots + (k+1)]^{n-k} = 2^k [(k+1)(k+2)/2 - 3]^{n-k}$$

$$= 2^k [(k-1)(k+4)/2]^{n-k},$$

since $(k+1)(k+2) - 6 = k^2 + 3k - 4 = (k-1)(k+4)$. Again, given k there are $C(n, k)$ ways to pick $i_1, \cdots, i_k$, and k can take any value between 2 and n. □

Since the counts for $|F(n)|$ and $\mathrm{DP}(n)$ are not in closed form, we shall now derive lower and upper bounds for $|F(n)|$ and $\mathrm{DP}(n)$, to obtain some more information about their respective rates of growth.

For $n \geqq 3$, $2 \leqq \lceil n/2 \rceil < n$, and one can get lower bounds for $|F(n)|$ and $\mathrm{DP}(n)$ in a trivial way, by taking the term corresponding to $k = \lceil n/2 \rceil$ in the respective sum. Specifically,

$$|F(n)| > C(n, \lceil n/2 \rceil)(\lceil n/2 \rceil - 1)^{\lfloor n/2 \rfloor},$$

and

$$\begin{aligned} \mathrm{DP}(n) &> C(n, \lceil n/2 \rceil) 2^{\lceil n/2 \rceil} [(\lceil n/2 \rceil - 1)(\lceil n/2 \rceil + 4)/2]^{\lfloor n/2 \rfloor} \\ &\geqq C(n, \lceil n/2 \rceil) 2^{\lceil n/2 \rceil - \lfloor n/2 \rfloor} (\lceil n/2 \rceil - 1)^{2 \lfloor n/2 \rfloor} \\ &\geqq C(n, \lceil n/2 \rceil)(\lceil n/2 \rceil - 1)^{n-1}. \end{aligned}$$

Since $\lceil n/2 \rceil \geqq n/2$ and $\lfloor n/2 \rfloor > n/2 - 1$, we thus have

$$|F(n)| > C(n, \lceil n/2 \rceil)(n/2 - 1)^{n/2 - 1}, \qquad \mathrm{DP}(n) > C(n, \lceil n/2 \rceil)(n/2 - 1)^{n-1}.$$

Moreover, by Stirling's formula ($n! \approx n^n e^{-n} (2\pi n)^{1/2}$) we have

$$C(2r, r) = (2r)!/r!r! \approx (2r)^{2r} e^{-2r} (2\pi 2r)^{1/2} / (r^r e^{-r})^2 2\pi r = 2^{2r} (\pi r)^{-1/2},$$

and

$$C(2r+1, r+1) = (2r+1)!/(r+1)!r! = [(2r)!/r!r!][(2r+1)/(r+1)] \approx 2^{2r+1} (\pi r)^{-1/2},$$

so $C(n, \lceil n/2 \rceil) \approx 2^n (\pi \lfloor n/2 \rfloor)^{-1/2} \approx 2^n (\pi n/2)^{-1/2}$ ($a_n \approx b_n$ means $\lim_{n \to \infty} a_n / b_n = 1$); this gives an idea about the rate of growth of these lower bounds.

We can also obtain trivial upper bounds by replacing $(k-1)^{n-k}$ in the summation expression for $|F(n)|$ by $(n-1)^n$, and by replacing $2^k [(k-1)(k+4)/2]^{n-k}$ in the summation expression for $\mathrm{DP}(n)$ by $2^n [(n-1)(n+4)/2]^n = [(n-1)(n+4)]^n$. We thus obtain, using the well-known fact that the sum of the binomial coefficients $C(n, k)$ for $k = 0, \cdots, n$ is equal to 2^n,

$$|F(n)| < [2(n-1)]^n \quad \text{and} \quad \mathrm{DP}(n) < [2(n-1)(n+4)]^n.$$

Observe that it immediately follows from these straightforward bounds that $\log |F(n)| = \Theta(n \log n)$, and $\log \mathrm{DP}(n) = \Theta(n \log n)$.

We shall now derive some more elaborate bounds. First, we derive an upper bound for $|F(n)|$ by estimating the maximum of $(k-1)^{n-k}$ when k ranges from 2 to n.

THEOREM 3. *$|F(n)|<2^n(n/k)^{n/k \log(n/k)}$, where $k=\log n-\log\log n$ (log denotes the natural logarithm).*

Proof. Consider the function $y:(1,\infty)\to\mathbb{R}$ *defined by* $y(x)=(x-1)^{n-x}$, where $n>2$; $y'(x)=(x-1)^{n-x}g(x)$, where $g(x)=(n-x)/(x-1)-\log(x-1)=-1+(n-1)/(x-1)-\log(x-1)$. Now $g'(x)=-(n-1)/(x-1)^2-1/(x-1)<0$ $(x>1)$, and $g(2)=n-2>0$, $g(n)=-\log(n-1)<0$, so $g(x)$ has a unique root x_0 in $(2,n)$. Also $g(x)>0$ for $1<x<x_0$, $g(x)<0$ for $x>x_0$, so y has an absolute maximum $y_{\max}$ at x_0. Since $g(x_0)=0$, $(n-x_0)/(x_0-1)=\log(x_0-1)$, and $y_{\max}=y(x_0)=(x_0-1)^{n-x}0=(x_0-1)^{(x_0-1)\log(x_0-1)}$. Since $x_0-1>1$, we have that if $x_0-1<\mu$ then $y_{\max}<\mu^{\mu\log\mu}$. But now if k is such that $n>k\,e^{k-1}+1$, then $1+\log[(n-1)/k]>k$, so for $x\geqq 1+(n-1)/k$ we have $1+\log(x-1)\geqq 1+\log[(n-1)/k])k\geqq(n-1)/(x-1)$, which gives $g(x)<0$, so $x_0<1+(n-1)/k$. Thus, if $n>k\,e^{k-1}+1$, $x_0-1<(n-1)/k$ and $y_{\max}<[(n-1)/k]^{[(n-1)/k]\log[(n-1)/k]}$. Taking $k=\log n-\log\log n$, we have that $n>k\,e^{k-1}+1$ iff (after some calculations) $n[1-1/e+(\log\log n)/(e\log n)]>1$, which is true since for $n>2$ we have $n>e$ and $\log\log n>0$, and thus $n[1-1/e+(\log\log n)/(e\log n)]>n(1-1/e)>2(1-1/e)>2(1-1/2)=1$. But now for each k in $[2,n]$, $(k-1)^{n-k}\leqq y_{\max}$ by the definition of y, so $|F(n)|=\sum_{k=2}^n C(n,k)(k-1)^{n-k}\leqq y_{\max}\sum_{k=2}^n C(n,k)=y_{\max}(2^n-n-1)<2^n y_{\max}<2^n(n/k)^{n/k\log(n/k)}$, where $k=\log n-\log\log n$. □

To improve the lower bound on $|F(n)|$, we first find alternative summation expressions for $|F(n)|$ by calculating the exponential generating function of the sequence $|F(n)|$.

PROPOSITION 2.

(a) $$|F(n)|=(-1)^{n-1}+\sum_{2k_1+3k_2+\cdots=n} n!/k_1!(1!)^{k_1}k_2!(2!)^{k_2}\cdots,$$

where $n\geqq 2$ and the k_i's are nonnegative integers.

(b) $$|F(n)|=(-1)^{n-1}+\sum_{r=1}^{\lceil n/2\rceil} n!/(n-r!)S(n-r,r),$$

where $n\geqq 2$ and $S(n,k)$ is the Stirling number of the second kind which is equal to the number of partitions of an n-element set into exactly k classes.

Proof. We first calculate the exponential generating function of the sequence $|F(n)|$:

$$f(x)=\sum_{n=2}^{\infty}|F(n)|x^n/n!=\sum_{n=2}^{\infty}\left(\sum_{k=2}^{n}C(n,k)(k-1)^{n-k}\right)x^n/n!$$

$$=\sum_{n=2}^{\infty}\sum_{k=2}^{n} n!/k!(n-k)!\,(k-1)^{n-k}x^n/n!$$

$$=\sum_{k=2}^{\infty}\sum_{n=k}^{\infty}[(k-1)x]^{n-k}/(n-k)!x^k/k!$$

$$=\sum_{k=2}^{\infty}x^k/k!\,e^{(k-1)x}=e^{-x}\sum_{k=2}^{\infty}(x\,e^x)^k/k!=e^{-x}(e^{xe^x}-x\,e^x-1)=e^{x(e^x-1)}-x-e^{-x}.$$

(a) We find an alternative expression for the coefficients in the expansion of $f(x)$:

$$-x-e^{-x}=-1+\sum_{n=0}^{\infty}(-1)^{n-1}x^n/n!=-1+\sum_{n=2}^{\infty}(-1)^{n-1}x^n/n!,$$

and

$$e^{x(e^x-1)} = \sum_{n=0}^{\infty} [x(e^x-1)]^n/n!$$

$$= \sum_{n=0}^{\infty} 1/n!\left(\sum_{i=1}^{\infty} x^{i+1}/i!\right)^n$$

$$= \sum_{n=0}^{\infty} l/n! \sum_{k_1+k_2+\cdots=n} n!/k_1!k_2!\cdots \prod_{i=1}^{\infty} x^{(i+1)k_i}/(i!)^{k_i}$$

$$= \sum_{k_1\geqq 0, k_2\geqq 0,\cdots} x^{2k_1+3k_2}+\cdots/k_1!(1!)^{k_1 k_2}!(2!)^{k_2}\cdots$$

$$= \sum_{n=0}^{\infty}\left[\sum_{2k_1+3k_2+\cdots=n} n!/k_1!(1!)^{k_1}k_2!(2!)^{k_2}\cdots\right]x^n/n!$$

$$= 1+\sum_{n=2}^{\infty}\left[\sum_{2k_1+3k_2+\cdots=n} n!/k_1!(1!)^{k_1}k_2!(2!)^{k_2}\cdots\right]x^n/n!$$

Thus, since $f(x) = (-x-e^{-x})+e^{x(e^x-1)}$, we obtain

$$|F(n)| = (-1)^{n-1} + \sum_{2k_1+3k_2+\cdots=n} n!/k_1!(1!)^{k_1}k_2!(2!)^{k_2}\cdots.$$

(b) We rewrite the summation expression obtained in (a) as follows:

$$|F(n)| = (-1)^{n-1} + \sum_{r=1}^{\lfloor n/2\rfloor} \sum_{\substack{k_1+2k_2+\cdots=n-r\\ k_1+k_2+\cdots=r}} n!/(n-r)!(n-r)!/k_1!(1!)^{k_1}k_2!(2!)^{k_2}\cdots.$$

But now observe that $(n-r)!/K_1!(1!)^{k_1}k_2!(2!)^{k_2}\cdots$ is equal to the number of partitions of an $(n-r)$-element set in which there are exactly k_i classes with i elements, so the inner sum is equal to $n!/(n-r)!S(n-r,r)$. Therefore,

$$|F(n)| = (-1)^{n-1} + \sum_{r=1}^{\lfloor n/2\rfloor} n!/(n-r)!S(n-r,r). \qquad \square$$

Using (b) of Proposition 2, we can improve our lower bound as follows: We first obtain a simple estimate for $S(n,k)$:

LEMMA 3. $S(n,k) \geqq k^{n-k}/k!$

Proof. The number of ways of putting n distinct objects into k distinct boxes is equal to $k!S(n,k)$; the number of ways of putting n distinct objects into k distinct boxes such that object i is in box i is equal to k^{n-k}; clearly, $k!S(n,k) \geqq k^{n-k}$. $\square$

By considering the term corresponding to $r = \lfloor pn\rfloor$ in the summation expression for $|F(n)|$ given in Proposition 2(b), and using Lemma 3 and Stirling's approximation, we have

THEOREM 4. *For all* $0<p<\frac{1}{2}$, $|F(n)|$ *is bounded from below for large enough* n *by*

$$(c_p n^{1-2p})^n(2\pi p(1-p)n^3)^{-1/2}(e^{1-1/p}-\varepsilon)$$

where $c_p = p^{1-3p}(1-p)^{p-1}$, *and* $\varepsilon>0$.

The first few values of $|F(n)|$ are given in Table 1.

TABLE 1

n	$\|F(n)\|$	DP(n)
3	4	44
4	15	456
5	66	5,992
6	335	101,212
7	1.898	1,889,428

3.3. Solving the transportation problem. In this subsection we briefly outline the Edmonds–Karp scaling method for the min-cost network flow problem, of which the transportation problem is a special case. Recall that we wish to find the cheapest "pseudo-Eulerian" (i.e., with balanced indegrees–outdegrees but perhaps not connected) digraph with the given indegrees $c_i = k_i - \delta_i$. This is equivalent to the min-cost max-flow problem on the following network N ([FF], [La], [EK], [PS]): The nodes of N are $\{s, t\} \cup \{s_i, t_i : i = 1, \cdots, n\}$ and the arcs are $\{(s, s_i), (t_i, t) : i = 1, \cdots, n\} \cup \{(s_i, t_j) : i, j = 1, \cdots, n\}$. Arcs (s, s_i), (t_i, t) have cost 0 and capacity c_i, whereas arc (s_i, t_j) has cost d_{ij} and capacity ∞.

A flow f from s to t in N is called *extreme* if it is of minimum cost among the flows of equal value. It is called *pseudo-extreme* if there exist real numbers $u_i, v_i, i = 1, \cdots, n$ such that (a) $u_i - v_j + d_{ij} \geqq 0$ for all i, j and (b) whenever $u_i - v_j + d_{ij} > 0$ we have 0 flow in f from s_i to t_j. If we start with a pseudo-extreme initial flow we can perform flow augmentations that preserve the pseudo-extreme property. The maximum flow we end up with is therefore pseudo-extreme, and it turns out that the maximum pseudo-extreme flow is also extreme, and thus the desired solution (see [EK] for a proof).

Define now the pth *approximation* to our problem to be a min-cost max-flow problem on the same nodes, arcs and costs, only with capacities $\{\lfloor c_i/2^p \rfloor\}$. The original problem is thus the 0th approximation. If f is a pseudo-extreme flow in the pth approximation, then obviously $2f$ is a pseudo-extreme flow in the $(p-1)$th. The Edmonds–Karp scaling method computes in this way successively maximum pseudo-extreme flows for approximations $l, l-1, \cdots, 0$, where $l = \lceil \log_2 (\max_i c_i) \rceil$. Each approximation can be solved in $O(n^3)$ time, and the total complexity is $O(n^3 \log (\sum c_i))$.

For our problem we must solve $|F(n)|$ such min-cost max-flow problems, all with the same nodes, arcs and costs, and with capacities varying slightly (namely, $c_i = k_i - \delta_i$) for a total complexity $O(|F(n)| n^3 \log_2 (\sum k_i))$. Instead, however, we could solve a single "master" problem with capacities $\{\lfloor (k_i - n)/2^p \rfloor\}$, where p is to be determined. Then we solve each of the $|F(n)|$ problems by starting with the $(p-1)$th approximation, and with initial flow $2f$, where f is the optimum flow in the master problem. By taking $p = \lceil \log n \rceil$ we can solve each of the $|F(n)|$ problems in $O(n^3 \log n)$ time (notice that always $k_i - \delta_i \geqq k_i - n$). The total computation for the transportation problems is therefore reduced from $O(|F(n)| n^3 \log (\sum k_i))$ to $O(n^3 \log (\sum k_i) + |F(n)| n^3 \log n)$.

4. Discussion. A good part of our investigations has been of rather theoretical interest—e.g., the asymptotic improvement sketched in § 3.3. Nevertheless, we think that our algorithm is of practical value, since it can be used to solve instances of size far beyond those previously thought possible. One of the most attractive features of our algorithm in practice is that, if n and the distance matrix are known and fixed in

advance, then the best part of the computation (i.e., the generation of $F(n)$ and the computation of the optimal Eulerian graph for each sequence in it) can be done once and for all, and the results stored in a large table. Besides, our algorithm can be adapted to find the optimal solution of a *dynamically evolving* instance (e.g., by performing a few more augmentations in the transportation problem whenever the k_i's are increased), whereas the dynamic programming approach is not very flexible in this direction. Naturally, there is a drawback: Our approach is best suited for minimizing the length of the walk (the *makespan*, or finishing time of the last job, in scheduling terminology), while dynamic programming can be adapted to optimize other objectives as well [Ps]. We also mention in passing that our approach to the many-visits TSP is reminiscent in spirit of the classical "precomputation" approach to the cutting-stock problem [GG1].

A practical implementation of our algorithm would most probably incorporate a less sophisticated code for the transportation problem that the Edmonds–Karp scaling method, and could use a heuristic test for minimality for the digraph G produced in step 1. Of course, the ultimate heuristic would be to first solve the transportation problem with requirements and capacities k_i, and then check whether, by a stroke of luck, the resulting digraph is connected. One might expect that this should happen much more often in this problem than in the ordinary TSP.

REFERENCES

[Ch] N. CHRISTOFIDES, *Graph Theory: An Algorithmic Approach*, Academic Press, New York, 1975.

[EK] J. EDMONDS AND R. M. KARP, *Theoretical improvements in the algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach., 19 (1972), pp. 248–264.

[FF] L. R. FORD, JR. AND D. R. FULKERSON, *Flows in Networks*, Princeton Univ. Press, Princeton, NJ, 1962.

[GG1] P. C. GILMORE AND R. E. GOMORY, *A linear programming approach to the cutting stock problem*, J. Oper. Res. Soc. Amer., 9 (1961), pp. 849–859; Part II, 11 (1963), pp. 863–888.

[GG3] ———, *Sequencing a one-state variable machine: A solvable case of the traveling salesman problem*, J. Oper. Res. Soc. Amer., 12 (1964), pp. 655–679.

[HK] M. HELD AND R. M. KARP, *A dynamic programming approach to sequencing problems*, J. Soc. Ind. Appl. Math., 10 (1962), pp. 196–210.

[Ka] R. M. KARP, *A patching algorithm for the nonsymmetric traveling salesman problem*, SIAM J. Comput., 8 (1979), pp. 461–473.

[Ku] H. W. KUHN, *The Hungarian method for the assignment problem*, Naval Res. Log. Quart., 3 (1956), pp. 253–258.

[La] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt-Rinehart-Winston, New York, 1976.

[PS] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[PY] C. H. PAPADIMITRIOU AND M. YANNAKAKIS, *On minimal Eulerian graphs*, Inform. Proc. Letters, 12 (1981), pp. 203–205

[Ps] H. N. PSARAFTIS, *A dynamic programming approach for sequencing groups of identical jobs*, J. Oper. Res. Soc. Amer., 28 (1980), pp. 1347–1359.

SIAM J. COMPUT.
Vol. 13, No. 1, February 1984

© 1984 Society for Industrial and Applied Mathematics
0097-5397/84/1301-0008 $01.25/0

THE EFFECT OF NUMBER OF HAMILTONIAN PATHS ON THE COMPLEXITY OF A VERTEX-COLORING PROBLEM*

UDI MANBER†‡ AND MARTIN TOMPA†

Abstract. A generalization of Dobkin and Lipton's element uniqueness problem is introduced. For any fixed undirected graph G on vertex set $\{v_1, v_2, \cdots, v_n\}$, the problem is to determine, given n real numbers $x_1, x_2, \cdots, x_n$, whether $x_i \neq x_j$ for every edge $\{v_i, v_j\}$ in G. This problem is shown to have upper and lower bounds of $\Theta(n \log n)$ linear comparisons if G is any dense graph. The proof of the lower bound involves showing that any dense graph must contain a subgraph with many Hamiltonian paths, and demonstrating the relevance of these Hamiltonian paths to a geometric argument. In addition, we exhibit relatively sparse graphs for which the same lower bound holds, and relatively dense graphs for which a linear upper bound holds.

Key words. lower bound, decision tree, element uniqueness, Hamiltonian path, vertex-coloring, orientation

1. A problem related to vertex-coloring.

Dobkin and Lipton [1] investigated the complexity of the element uniqueness problem which is to decide, given n real numbers, whether they are pairwise distinct. They employed a geometric argument to demonstrate that $\Omega(n \log n)$ linear comparisons of the inputs are required in the worst case to determine element uniqueness.

What if we are not interested in verifying distinctness for all pairs of inputs, but only for some subset of pairs? For any (fixed) undirected graph G on the vertex set $\{v_1, v_2, \cdots, v_n\}$, define *element uniqueness with respect to G* as the problem of deciding, given n real numbers $x_1, x_2, \cdots, x_n$, whether $x_i \neq x_j$ for every edge $\{v_i, v_j\}$ in G. Viewed another way, the problem is to determine whether coloring vertex v_i with color x_i, for $1 \leqq i \leqq n$, results in a valid coloring of the graph G. Notice that this problem differs from the familiar vertex-coloring problem in two respects:

1. The graph G is fixed, and the input is instead a candidate vertex-coloring that is to be verified. Hence, unlike the standard vertex-coloring problem which is well known to be NP-complete [3], this problem has an obvious algorithm that uses only $O(n^2)$ comparisons.

2. We restrict our attention to inputs that are real numbers and algorithms based solely on comparisons of linear combinations of the inputs. Although this eliminates some obvious algorithms for verifying vertex-colorings, it is in keeping with the spirit of Dobkin and Lipton [1] and allows us to explore some interesting geometric and graph-theoretic techniques for deriving lower bounds.

Section 2 demonstrates that $\Omega(n \log n)$ linear comparisons of the inputs are required to determine element uniqueness with respect to an arbitrary graph G that has $\Omega(n^2)$ edges. Section 3 complements this result by demonstrating that $O(n \log n)$ time is sufficient for arbitrary graphs. Section 4 shows that there are reasonably dense graphs with respect to which element uniqueness can be determined in $O(n)$ time, and there are reasonably sparse graphs with respect to which element uniqueness requires $\Omega(n \log n)$ linear comparisons.

* Received by the editors March 12, 1982. This material is based upon work supported by the Office of Naval Research under contract N00014-80-C-0221, and the National Science Foundation under grants MCS-7702474, MCS-8003337, and MCS-8110089.

† Department of Computer Science, FR-35, University of Washington, Seattle, Washington 98195.

‡ Present address: Department of Computer Science, University of Wisconsin, Madison, Wisconsin 53706.

2. A lower bound on element uniqueness with respect to dense graphs. This section is devoted to proving an $\Omega(n \log n)$ lower bound on the number of linear comparisons required to determine element uniqueness with respect to any graph G that has $\Omega(n^2)$ edges. The proof consists of 2 parts. In § 2.1 it is shown that if G has h Hamiltonian paths, then the number of linear comparisons required to determine element uniqueness with respect to G exceeds $\log_2 h$. This is, of course, insufficient to demonstrate the promised lower bound, since there are certainly graphs with $\Omega(n^2)$ edges but no Hamiltonian paths. Section 2.2, however, demonstrates that any graph with $\Omega(n^2)$ edges has an induced subgraph with $n^{\Omega(n)}$ Hamiltonian paths; this is sufficient to prove the lower bound, since it implies the existence of a fixed subset of the inputs to which the result of § 2.1 applies. (Note that there are graphs with $\Omega(n^2)$ edges and no triangles, so that we cannot simply apply Dobkin and Lipton's lower bound on element uniqueness to some complete subgraph of G.)

2.1. A geometric interpretation of the problem. Dobkin and Lipton's lower bound on the element uniqueness problem is based on the following result:

THEOREM 1 [1] *Let R be a set of nonempty, open, pairwise disjoint subsets of the Euclidean space E^n. Then given real numbers $x_1, x_2, \cdots, x_n$, the number of linear (affine) comparisons required to determine if $(x_1, x_2, \cdots, x_n) \in \bigcup_{r \in R} r$ is at least* $\log_2 |R|$.

The proof of Theorem 1 is based on the observation that, in any linear comparison tree that determines membership in $\bigcup_{r\in R} r$, the inputs that terminate at any leaf in the tree form a convex subset of E^n. Hence the number of leaves is at least $|R|$ and the height of the tree at least $\log_2 |R|$. In order to apply Theorem 1 to a particular problem such as element uniqueness, one needs to determine a lower bound on the number of regions corresponding to the problem.

The regions corresponding to the element uniqueness problem are those into which E^n is divided by removing the hyperplanes

$$H_{ij} = \{(x_1, x_2, \cdots, x_n) | x_i = x_j\}$$

for all pairs i, j. Thus, determining element uniqueness is equivalent to determining membership in $\bigcup_{r\in R}$, where

$$R = \{r_\pi | \pi \text{ is a permutation on } \{1, 2, \cdots, n\}\},$$

$$r_\pi = \{(x_1, x_2, \cdots, x_n) | x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}\}.$$

Hence the number of linear comparisons required to determine element uniqueness is at least $\log_2 |R| = \log_2 (n!) = n \log_2 n - O(n)$.

Let G be an undirected graph on the vertex set $V = \{v_1, v_2, \cdots, v_n\}$, and let R_G be the set of nonempty, open, pairwise disjoint regions into which E^n is divided by removing the hyperplanes

$$H_{ij} = \{(x_1, x_2, \cdots, x_n) | x_i = x_j\}$$

for all pairs i, j such that the edge $\{v_i, v_j\}$ is in G. Determining element uniqueness with respect to G is equivalent to determining membership in $\bigcup_{r\in R_G} r$, so we proceed by characterizing the regions of R_G in order to apply Theorem 1.

An *orientation* of $G = (V, E)$ is a directed graph $D = (V, A)$ that has exactly one of the (directed) edges (u, v) or (v, u) for each (undirected) edge $\{u, v\}$ in G. For any directed graph $D = (V, A)$, define the open region

$$r_D = \bigcap_{(v_i, v_j) \in A} \{(x_1, x_2, \cdots, x_n) | x_i < x_j\}.$$

THEOREM 2 (Greene [4]). *$R_G = \{r_D | D$ is an acyclic orientation of $G\}$; furthermore, if D and D' are distinct acyclic orientations, then $r_D \neq r_{D'}$.*

Proof. Let $r \in R_G$, and choose any $(a_1, a_2, \cdots, a_n) \in r$. Construct an orientation $D = (V, A)$ of G as follows: For each edge $\{v_i, v_j\}$ in G, orient this edge as (v_i, v_j) if $a_i < a_j$ and as (v_j, v_i) if $a_j < a_i$. (It is impossible that $a_i = a_j$, since all such points lie on the deleted hyperplane H_{ij}). The transitivity of $<$ assures the fact that the orientation D is acyclic. Furthermore $r = r_D$, since

$(x_1, x_2, \cdots, x_n) \in r$

$\Leftrightarrow$ for each hyperplane H_{ij}, $(x_1, x_2, \cdots, x_n)$ and $(a_1, a_2, \cdots, a_n)$ lie on the same side of H_{ij}

$\Leftrightarrow$ for each edge $(v_i, v_j) \in A$, $x_i < x_j$

$\Leftrightarrow (x_1, x_2, \cdots, x_n) \in r_D$.

Conversely, suppose D is an acyclic orientation of G. There is some $(a_1, a_2, \cdots, a_n)$ satisfying $a_i < a_j$ for each edge (v_i, v_j) in D, since D is acyclic. As $(a_1, a_2, \cdots, a_n) \notin H_{ij}$ for any edge $\{v_i, v_j\}$, $(a_1, a_2, \cdots, a_n) \in r$ for some $r \in R_G$. As in the previous paragraph, $r = r_D$.

Finally, if D and D' are acyclic orientations that differ on their orientation of the edge $\{v_i, v_j\}$, then r_D and $r_{D'}$ are nonempty and are separated by H_{ij}, so $r_D \neq r_{D'}$. □

In the next section we will consider only those acyclic orientations that correspond to Hamiltonian paths, since it is easier to count Hamiltonian paths than general orientations. We proceed to show that every Hamiltonian path induces an acyclic orientation, so that a lower bound on the number of Hamiltonian paths yields a lower bound for the problem.

COROLLARY 3. *If G has h Hamiltonian paths, then the number of linear comparisons required to determine element uniqueness with respect to G exceeds $\log_2 h$.*

Proof. Every directed Hamiltonian path p induces a natural acyclic orientation of G, namely edge $\{u, v\}$ is oriented in the same direction as the subpath of p between u and v. Furthermore, no two distinct Hamiltonian paths p and q induce the same acyclic orientation, for if p and q differ in their orderings of vertices u and v, then u and v would lie on a directed cycle in the common orientation. Hence, by Theorem 2, $|R_G| \geqq 2h$. The result then follows from Theorem 1. □

2.2. Dense graphs have subgraphs rich in Hamiltonian paths. Given a graph G with $\Omega(n^2)$ edges, our goal is to find an induced subgraph of G with $n^{\Omega(n)}$ Hamiltonian paths. The proof consists of two parts. First we show that G contains an induced subgraph G' with minimum degree $\Omega(n)$. G' is then shown to contain an induced subgraph G'' with the desired number of Hamiltonian paths.

LEMMA 4. *Let G be a graph with n vertices and cn^2 edges. Then there exists an induced subgraph G' of G with minimum degree greater than cn.*

Proof. G' is constructed using the following algorithm:

1. Set $G' := G$.
2. Find a node v with minimum degree in G'.
3. If the degree of v in G' exceeds cn then halt.
4. Otherwise, delete v and its incident edges from G', and go to step 2.

If the algorithm terminates in step 3 then G' has minimum degree exceeding cn. We prove that the algorithm will always terminate after fewer than $n - 2cn$ vertices are deleted. Assume the contrary. For each vertex deleted in step 4, the number of edges deleted is at most cn. Hence after $n - \lfloor 2cn \rfloor$ deletions $G' = (V', E')$ satisfies

$|V'| = \lfloor 2cn \rfloor$ and $|E'| \geqq cn^2 - (n - \lfloor 2cn \rfloor)cn > 2c^2n^2 - cn$. This is a contradiction, since the complete graph with $\lfloor 2cn \rfloor$ vertices has at most $2c^2n^2 - cn$ edges. □

LEMMA 5. *Let G be a graph with n' vertices and minimum degree d. Then G has an induced subgraph G' with $d+1$ vertices and at least $d!/\binom{n'}{d+1}$ Hamiltonian paths.*

Proof. We first show that there are at least $d!$ simple paths in G of length d. Let $(v_0, v_1, \cdots, v_d)$ denote such a path. v_0 can be chosen arbitrarily. For each choice of v_0 there are at least d choices for v_1, for each v_1 at least $d-1$ choices for v_2, and in general for each v_j at least $d-j$ choices for v_{j+1}, since v_j is adjacent to at least d vertices, at most j of which have already been chosen. Hence there are at least $d!$ such paths.

Each simple path $(v_0, v_1, \cdots, v_d)$ is a Hamiltonian path in the induced subgraph G' generated by $\{v_0, v_1, \cdots, v_d\}$. Since the number of subgraphs of G with $d+1$ vertices is $\binom{n'}{d+1}$, some induced subgraph of G must have at least $d!/\binom{n'}{d+1}$ Hamiltonian paths. □

Combining the two lemmas we get the following theorem:

THEOREM 6. *Let G be a graph with n vertices and cn^2 edges. Then G has an induced subgraph G' with at least $\lceil cn \rceil!/2^n$ Hamiltonian paths.*

Proof. Follows from Lemmas 4 and 5. □

We can now combine Corollary 3 and Theorem 6 to get our main result:

THEOREM 7. *If G has n vertices and cn^2 edges, then the number of linear comparisons required to determine element uniqueness with respect to G is at least $cn \log_2 n - O(n)$.*

Proof. By Corollary 3 and Theorem 6, there is a fixed subset of the inputs that alone requires this many comparisons. □

THEOREM 8. *If G has n vertices and $\omega(n^2/\log n)$ edges, then the number of linear comparisons required to determine element uniqueness with respect to G is $\omega(n)$.*

Proof. The proof of Theorem 7 can be modified in a straightforward manner to prove the stronger statement that if G has n vertices and m edges, then the number of linear comparisons required to determine element uniqueness with respect to G is at least $(m/n) \log_2 (m/n) - O(n)$. □

3. An upper bound on element uniqueness with respect to arbitrary graphs. In this section we establish an upper bound of $O(n \log n)$ for element uniqueness with respect to any fixed graph G. This upper bound is immediate if only comparisons are counted, as was done in § 2, since sorting the input reveals (implicitly) all the information necessary to determine element uniqueness with respect to a fixed graph. The algorithm given in this section shows that this information can be collected explicitly on a random access machine using $O(n \log n)$ steps in total.

The algorithm presented is not meant to be practical, though, since it requires an exponential amount of preprocessing of G, and an exponential size data structure remains even after preprocessing. It serves only to show that the lower bounds presented in § 2 are tight.

The preprocessing of G proceeds as follows. For each induced subgraph $G' = (V', E')$ of G we check whether $E' = \emptyset$. We store the answers in a "binomial tree" [2, 8], which facilitates fast look-up. A *binomial tree* is a tree of height n with 2^n nodes. Each node is labelled with an element from $\{0, 1, \cdots, n\}$ as follows: the root is labelled 0, and each node labelled i has $n-i$ children with labels $i+1, i+2, \cdots, n$, respectively. Thus each subset V' of k vertices from V corresponds in a natural way to a node $u_{V'}$ in the binomial tree of distance k from the root. Each node $u_{V'}$ also has a flag that is set if and only if $E' \neq \emptyset$ for the induced subgraph $G' = (V', E')$.

Now, given a coloring of the vertices of G, to find whether there are two adjacent vertices with the same color do the following:

1. Sort the vertices according to their colors, thus obtaining a list of subsets $V_1, V_2, \cdots, V_k$ such that each subset V_j contains vertices with the same color. Sort the vertices within each subset according to their vertex numbers. This step takes time $O(n \log n)$.

2. For each subset V_j, consult the binomial tree to find whether the corresponding $E_j = \emptyset$. There are two adjacent vertices with the same color in G if and only if $E_j \neq \emptyset$ for some j. For each subset V_j, the look-up takes time $O(|V_j| \log n)$, so this step takes time $O(n \log n)$.

4. The number of acyclic orientations of undirected graphs. This section supplements the results of §§ 2 and 3 by demonstrating that (1) there are reasonably dense graphs with respect to which element uniqueness can be determined in $O(n)$ time, and (2) there are reasonably sparse graphs with respect to which element uniqueness requires $\Omega(n \log n)$ linear comparisons.

THEOREM 9. *There are graphs with* $\Theta(n^2/\log^2 n)$ *edges for which there is a linear time algorithm (with no preprocessing) for the element uniqueness problem.*

Proof. Let $G = G' \cup G''$ where G' is the complete graph with $n/\log_2 n$ vertices, and G'' has $n - n/\log_2 n$ vertices and no edges. The element uniqueness problem with respect to G is reduced to the ordinary element uniqueness problem for $n/\log_2 n$ elements, which can be solved in time $O(n)$ by sorting. □

Theorem 2 motivates studying the number of acyclic orientations of undirected graphs, in order to determine whether the lower bound technique of § 2 applies. Theorem 10 below tightly bounds the maximum number of acyclic orientations that a graph with n vertices and m edges can have. One corollary is that there are relatively sparse graphs with many acyclic orientations.

Let S be the set of undirected graphs with n vertices and $m \geqq n$ edges, and let

$H(m, n)$ denote the maximum number of directed Hamiltonian paths that a graph in S can have,

$A(m, n)$ denote the maximum number of acyclic orientations that a graph in S can have, and

$F(m, n)$ denote the maximum number of rooted spanning forests that a graph in S can have.

THEOREM 10. $(2m/en - 1)^n \leqq H(m, n) \leqq A(m, n) \leqq F(m, n) \leqq (2m/n + 1)^n$.

Proof. 1. $H(m, n) \geqq (2m/en - 1)^n$. Let $\mathbf{G}$ be a graph chosen at *random* from S, with each graph considered equiprobably. Let $N = \binom{n}{2}$. Then the expected number of Hamiltonian paths in $\mathbf{G}$ is the total number of possible Hamiltonian paths ($n!$) times the probability that any fixed possible Hamiltonian path is indeed a Hamiltonian path in $\mathbf{G}$, that is,

$$\begin{aligned} n!\binom{N-n+1}{m-n+1}\Big/\binom{N}{m} &\geqq n!((m-n)/N)^{n-1} \\ &\geqq (n/e)^n(2(m-n)/n^2)^n \\ &= (2(m-n)/en)^n \\ &\geqq (2m/en-1)^n. \end{aligned}$$

(A constructive proof of a slightly weaker bound was given in a preliminary version of this paper [5].)

2. $H(m, n) \leqq A(m, n)$. The proof of Corollary 3 yields the stronger result that any graph has at least as many acyclic orientations as Hamiltonian paths.

3. $A(m, n) \leqq F(m, n)$. Let G be an undirected graph with n vertices, and number the vertices $1, 2, \cdots, n$ in an arbitrary way. Each acyclic orientation D of G induces a rooted spanning forest F, namely the one that arises from a depth-first search [7] of the directed graph D (breaking ties according to the vertex numbering). Furthermore, F uniquely determines the orientation D: any edge in G connecting an ancestor and descendent in F must be oriented toward the descendent in D, since D is acyclic, and any other edge must be oriented toward the vertex visited earlier in the depth-first search. (The order in which the vertices of F were visited can be determined from the vertex numbering.) Hence any graph has at least as many rooted spanning forests as acyclic orientations.

4. $F(m, n) \leqq (2m/n+1)^n$. A rooted spanning forest is uniquely specified by stating for each vertex whether it is a root, or if not which vertex is its parent. Hence there are at most $\prod_{v \in V} (1 + \text{degree}(v))$ rooted spanning forests of any graph $G = (V, E)$ with $|V| = n$ and $|E| = m$. But $\sum_{v \in V} \text{degree}(v) = 2m$, so this product is at most $(2m/n+1)^n$. □

COROLLARY 11. *For any $0 < \varepsilon < 1$, there are undirected graphs with n vertices and $n^{1+\varepsilon}$ edges with respect to which determining element uniqueness requires $\Omega(n \log n)$ linear comparisons. (In fact, a randomly chosen graph with these parameters exhibits this behavior.)*

Proof. This follows directly from Theorem 10 and Corollary 3. □

5. Conclusions. This paper presented a geometric interpretation of a graph-theoretic problem (determining whether a vertex-coloring of a fixed graph is valid) which led to a seemingly unrelated graph-theoretic problem (counting the number of acyclic orientations of undirected graphs) that, in turn, helped to establish a lower bound for the original problem. We believe that this technique, a generalization of Dobkin and Lipton's [1], can be helpful in establishing lower bounds for similar problems. The connection between the two graph-theoretic problems mentioned above is interesting in itself; one can also invert the argument in order to get an upper bound on the number of acyclic orientations from an efficient algorithm for verifying vertex colorings.

It should be mentioned that all the lower bounds on time presented in this paper hold even if the decision tree is allowed nondeterministic guessing [6].

This paper raises some open problems:

(1) Is there an algorithm for the vertex-coloring problem that runs in $o(m)$ time and needs only polynomial time for preprocessing? There is an obvious algorithm that runs in $O(m)$ time with no preprocessing and § 3 presented an algorithm that runs in $O(n \log n)$ time but needs exponential time for preprocessing. Is it possible to prove a tradeoff of preprocessing vs. processing time for this problem?

(2) Theorem 10 demonstrates that there *exist* graphs with n vertices and m edges with respect to which element uniqueness requires $n \log_2(m/n) - O(n)$ linear comparisons, and that this is the best lower bound that can be proved by counting acyclic orientations. We know that element uniqueness with respect to any graph with n vertices and m edges can be determined in time $O(n \log(m/n))$, if $m = O(n)$ or $m \geqq n^{1+\Omega(1)}$. Does this upper bound hold for all n and m? For instance, if $m = \Theta(n \log n)$ the lower bound is $\Omega(n \log\log n)$, but the best known upper bound is $O(n \log n)$.

(3) Theorems 7 and 8 demonstrate that element uniqueness with respect to *any* graph with n vertices and m edges requires $(m/n) \log_2(m/n) - O(n)$ linear com-

parisons. We know that there exist graphs with n vertices and m edges with respect to which element uniqueness can be determined in $O((m/n)\log(m/n)+n)$ time, if $m=O(n^2/\log^2 n)$ or $m=\Theta(n^2)$. Does this upper bound hold for all n and m? For instance, are there graphs with $\Theta(n^2/\log n)$ edges with respect to which element uniqueness can be determined in $O(n)$ time?

Acknowledgment. We thank Béla Bollobás, Victor Klee, and Larry Ruzzo for helpful ideas and results.

REFERENCES

[1] 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.

[2] M. J. FISCHER, *Efficiency of equivalence algorithms*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 158–168.

[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] C. GREENE, *Acyclic orientations*, in Higher Combinatorics, M. Aigner, ed., D. Reidel, Dordrecht, The Netherlands, 1977, pp. 65–68.

[5] U. MANBER AND M. TOMPA, *The effect of number of Hamiltonian paths on the complexity of a vertex-coloring problem*, 22nd Annual Symposium on Foundations of Computer Science, Nashville, Tennessee, October 1981, pp. 220–227.

[6] ———, *Probabilistic, nondeterministic, and alternating decision trees*, Proc. Fourteenth Annual ACM Symposium on Theory of Computing, San Francisco, California, May 1982, pp. 234–244.

[7] R. TARJAN, *Depth-first search and linear graph algorithms*, this Journal, 1 (1972), pp. 146–160.

[8] J. VUILLEMIN, *A data structure for manipulating priority queues*, Comm. ACM, 21 (1978), pp. 309–315.

SIAM J. COMPUT.
Vol. 13, No. 1, February 1984

© 1984 Society for Industrial and Applied Mathematics
0097-5397/84/1301-0009 $01.25/0

OPTIMIZING CHAIN QUERIES IN A DISTRIBUTED DATABASE SYSTEM*

DAH-MING CHIU†‡, PHILIP A. BERNSTEIN† AND YU-CHI HO†

Abstract. This paper studies the problem of query optimization in a distributed database. Assuming a linear additive cost function (in volume of data moved), we present a fast algorithm for finding the optimal program that answers a class of common queries, called chain queries. The key to the problem formulation and to the derivation of an efficient algorithm is an elegant parameterization of the database state against which the query is to be answered. This parameterization then enables us to characterize the set of potentially optimal programs, which in turn leads to a fast dynamic programming algorithm. Since in practice the needed parameters may not be available to the database system, we also discuss how to deal with partial parameterizations of the database state.

Key words. distributed database, database systems, relational databases, query optimization, semijoins

1. Introduction. A principal capability of relational database systems is their ability to process queries expressed in languages based on predicate calculus. If the system were to process such queries in the most obvious manner with no optimization, the system would be unbearably slow and therefore unusable. Thus, optimization is essential.

In a distributed database system, sites are connected by a communications network. When this network has narrow bandwidth, as in ARPANET and similar geographically distributed point-to-point media, the principal potential performance bottleneck for query processing is communication cost. So, to optimize query processing in such an environment, one should try to minimize communication.

A relational query can be modelled by an expression consisting of projection, selection and join operations [7]. If each relation is stored at one site, then projection and selection operations incur negligible communication cost. Communications only arise when two relations residing at different sites need to be joined. In this situation, one relation must be moved to the other relation's site, so the join can be performed. This move operation requires communication and should therefore be optimized.

The main tactic for reducing the cost of such moves is the semijoin operation. The semijoin of relation R by relation S is the set of tuples in R each of which joins with some tuple of S. To join R and S in a distributed database system, we might decide to move R to S's site. To reduce the cost of this move, we can first perform the semijoin of R by S. This semijoin eliminates from R all tuples that do not join with S. The cost of the semijoin is the movement of S's joining column from S's site to R's site. If this cost is small and the amount by which R is reduced is large, then the semijoin is cost beneficial and should be executed before R is moved.

Not all semijoins are cost beneficial. Whether a particular semijoin is cost beneficial depends on the database state and the move operations that follow it. Hence, the selection of cost beneficial semijoins is an optimization problem: For a given query, what sequence of semijoins leads to a lowest cost execution of that query?

* Received by the editors February 3, 1981, and in revised form June 30, 1982. This work was supported by the Joint Service Electronics Program under contract N00014-75-C-0648, by the National Science Foundation under grants ENG78-15231 and MCS79-07762, and by the Computer Corporation of America.

† Division of Applied Sciences, Harvard University, Cambridge, Massachusetts 02138.

‡ Current address: Digital Equipment Corporation, Hudson, Massachusetts 01749.

General but heuristic solutions appear in [3], [9]. An optimal solution for queries that are set intersections appears in [8].

Many queries can be entirely solved by semijoins [1], [2], [10]. In particular, semijoins can completely solve queries of the form $R_1 \ltimes R_2 \ltimes \cdots \ltimes R_n$, where $R_i \neq R_j$ for $i \neq j$, $\ltimes$ is the join operation on arbitrary attributes, and each relation only joins with its neighbor(s) in the query. These queries, called *chain queries*, correspond exactly to the nested mappings of SEQUEL [4] and are a commonly observed subclass of relational queries. The optimization of chain queries by semijoins is the main subject of this paper.

The chain query optimization problem is: Given a chain query and an initial database state, find a semijoin program (i.e., a sequence of semijoins) that solves the query at lowest cost. We formalize this problem in § 2. In § 3, we characterize the intermediate database states that can be reached by a semijoin program that correctly solves the query. In § 4, we use this characterization of intermediate states to develop an efficient method for calculating the cost of a semijoin program. Our main technical result is presented in § 5: a description of a minimal set of semijoin programs that must contain the optimal one. Combined with the cost calculation method of § 4, this set leads us, in § 6, to an efficient dynamic programming algorithm for finding the optimal program.

2. Problem formulation.

2.1. Database states. A database state is a finite set of finite *relations*, denoted $R_1, \cdots, R_n$. The set of columns (or *attributes*) of relation R_i is denoted $\mathbf{R}_i$. Each attribute, $A \in \mathbf{R}_i$, has an associated set of values, dom (A), called its *domain*. The domain of a relation dom $(\mathbf{R}_i)$, is the product of the domains of its attributes, $X_{A \in \mathbf{R}_i}$ dom (A). Thus, a relation is a finite subset of its domain. Elements of a relation are called *tuples*.

We use $D = (R_1, \cdots, R_n)$ to denote the *initial state* of the database against which queries will be posed.

The procedure that solves a query produces a sequence of database states, the last one of which is an *answer state*. The transition from a given initial state to a given answer state can be accomplished by many different sequences of operations that result in different state trajectories. Our problem is to determine the "best" trajectory according to some well defined criterion. This is more fully explained in § 3 and below.

2.2. Relational algebra. We define three relational algebraic operations: projection, join and semijoin. The *projection* of tuple $t \in R_i$ on a set of attributes $A \subseteq \mathbf{R}_i$, denoted $t[A]$, is the tuple that results from deleting all columns of t not in A. The projection of R_i on $A \subseteq \mathbf{R}_i$, denoted $R_i[A]$, is the set $\{t[A] | t \in R_i\}$.

The *natural join* (or, simply, the *join*) of relations R_i and R_j, denoted $R_i \ltimes R_j$, is the set $\{t \in \text{dom}(\mathbf{R}_i \cup \mathbf{R}_j) | t[\mathbf{R}_i] \in R_i \text{ and } t[\mathbf{R}_j] \in R_j\}$. That is, it "links" together tuples $t_i \in R_i$ with $t_j \in R_j$ if $t_i[\mathbf{R}_i \cap \mathbf{R}_j] = t_j[\mathbf{R}_i \cap \mathbf{R}_j]$.

The *semijoin* of R_i by R_j, denoted $R_i \ltimes R_j$, equals $(R_i \ltimes R_j)[\mathbf{R}_i]$, or equivalently $\{t_i \in R_i | (\exists t_j \in R_j)(t_i[\mathbf{R}_i \cap \mathbf{R}_j] = t_j[\mathbf{R}_i \cap \mathbf{R}_j])\}$. A *semijoin program* (or, simply, a *program*) is a sequence of semijoins (p_i), $p = p_1 \cdots p_m$. The *length* of p, denoted $\|p\|$, is the number of semijoins in p.

2.3. Cost function. We model the costs of relational algebraic operations in a distributed database environment. We assume that the cost of moving data between sites dominates all other costs. This assumption is appropriate when network bandwidth is the system bottleneck whose use we must minimize. For example, this as-

sumption was used in the query optimizer of the SDD-1 distributed database system, where the network (ARPANET) was the slowest system component by two orders of magnitude [3].

We assume that each relation is stored entirely at a single site. Hence, the projection operation can be applied to a relation with no intersite data movement, and so has no cost. Similarly, if a semijoin is applied to two relations at the same site, it incurs no cost.

If a semijoin is applied to two relations at different sites, then data must pass between the sites. Let $p = R_i \ltimes R_j$. To perform p at R_i's site, we must send $R_j[\mathbf{R}_i \cap \mathbf{R}_j]$ to that site. Assuming that the cost of moving data is linearly proportional to the amount of data moved, we have

$$\text{Cost}\,(p, D) = c_1|R_j[\mathbf{R}_i \cap \mathbf{R}_j]| + c_2 \tag{2.1}$$

where $|R_j[\mathbf{R}_i \cap \mathbf{R}_j]|$ is the "size" of $R_j[\mathbf{R}_i \cap \mathbf{R}_j]$. If we assume $|\mathbf{R}_i \cap \mathbf{R}_j| = 1$ and every domain value has unit size, then $|R_j[\mathbf{R}_i \cap \mathbf{R}_j]|$ is the cardinality of $R_j[\mathbf{R}_i \cap \mathbf{R}_j]$.

Intuitively, c_2 is the "start-up" cost of sending a message and c_1 is the "transfer" cost per unit of data. We let $c_1 = 1$ by appropriately choosing the unit of cost.

The cost of a semijoin program is assumed to be the sum of the costs of its component semijoins.

Most of the results of this paper (everything before § 5.4.2) are independent of the cost model of this section. In fact, all results in § 5 apply as long as Cost (p, D) is monotonic in $|R_j[\mathbf{R}_i \cap \mathbf{R}_j]|$.

2.4. Queries. We are interested in a class of queries called *chain queries*. The chain query Q on D is defined by

$$(R_1 \ltimes R_2 \ltimes \cdots \ltimes R_n)[\mathbf{R}_1] \tag{2.2}$$

where

$$\mathbf{R}_i \cap \mathbf{R}_j \neq \varnothing \quad \text{if } j = i-1 \text{ or } j = i+1$$

and

$$\mathbf{R}_i \cap \mathbf{R}_j \neq \varnothing \quad \text{if } j < i-1 \text{ or } j > i+1.$$

Intuitively, a chain query joins each pair of adjacent relations, R_i and R_{i+1}, and then projects onto the attributes of R_1.

In the remainder of the paper, we use "Q" to denote the chain query on $D = (R_1, \cdots, R_n)$ defined by (2.2). Since n is arbitrary, properties of Q hold for all chain queries.

Chain queries using natural joins are isomorphic to a much larger class of queries that use nonnatural joins (i.e., joins on arbitrary attributes). Roughly speaking, the latter class includes queries of the form of (2.2) where the nonnatural join operator is substituted for $\ltimes$ and where each attribute of each relation is a joining attribute for at most one join in the query. A more complete discussion of the isomorphism between natural and nonnatural join queries appears in [2].

Chain queries are an important subclass of relational queries that commonly appear in practice. Although we will mainly be concerned with solving chain queries, we will discuss a generalization to a larger class in § 6.

2.5. Problem statement. Our goal is to find optimal semijoin programs to solve chain queries. Specifically, we wish to develop an optimization algorithm for the

following problem:

INPUT:	A chain query Q on n relations.
	An initial state.
OUTPUT:	An optimal semijoin program that solves Q.
ASSUMPTION:	Every semijoin has cost governed by (2.1).

The assumption implies that no (relevant) semijoins can be evaluated without data movement. Intuitively, this amounts to assuming each relation is at a distinct site.

3. Semijoin reachable states. A semijoin $R_i \ltimes R_j$ is *relevant* to an order-n chain query if $1 \leqq i, j \leqq n$ and either $j = i-1$ or $j = i+1$. When evaluating a chain query using semijoins, only the relevant semijoins can affect the database state. Other semijoins have no effect. That is, $R_i \ltimes R_j = R_i$ whenever $j < i-1$ or $j > i+1$, because $\mathbf{R}_i \cap \mathbf{R}_j = \varnothing$. Notationally, we abbreviate the relevant semijoins as follows:

1. x_i represents $R_{i+1} \ltimes R_i$ for $i = 1, 2, \cdots, n-1$;
2. y_i represents $R_{i-1} \ltimes R_i$ for $i = 2, 3, \cdots, n$.

A program is *relevant* (for Q) if it consists entirely of relevant semijoins.

The cost of a program containing irrelevant semijoins can be no smaller than that of a relevant program. Therefore, we will deal exclusively with relevant programs in the sequel.

Let $p = p_1 p_2 \cdots p_m$ be a relevant program and let p_l $(1 \leqq l \leqq m)$ be the semijoin $R_i \ltimes R_j$ $(j = i+1$ or $i-1)$. To evaluate the cost of p_l, we must calculate $|R'_j[\mathbf{R}_i \cap \mathbf{R}_j]|$, where R'_j is the value of R_j after $p_1 p_2 \cdots p_{l-1}$ have been applied to D. Thus, to evaluate costs, we must determine the values of those states that can be reached by sequences of relevant semijoins. Such states are called *intermediate states.* (We regard the initial state, D, as an intermediate state.) The purpose of this section is to concisely characterize those intermediate states. The application of this characterization to cost calculations appears in § 4.

To help us characterize intermediate states, we define

$$R_i(l, h) = (R_l \ltimes R_{l+1} \ltimes \cdots \ltimes R_{h-1} \ltimes R_h)[\mathbf{R}_i]$$

where $1 \leqq l \leqq i \leqq h \leqq n$. Note that the initial state, D_0, can be expressed as $D_0 = (R_1(1, 1), R_2(2, 2), \cdots, R_n(n, n))$.

LEMMA 1. *Let* $D = (R_1(l_1, h_1), \cdots, R_n(l_n, h_n))$ *where* $1 \leqq l_i \leqq i \leqq h_i \leqq n$. *Then*

$$x_i(D) = (R_1(l_1, h_1), \cdots, R_i(l_i, h_i), R_{i+1}(\min(l_i, l_{i+1}), \max(h_i, h_{i+1})), \cdots, R_n(l_n, h_n)),$$

$$y_i(D) = (R_1(l_1, h_1), \cdots, R_{i-1}(\min(l_{i-1}, l_i), \max(h_{i-1}, h_i)), \cdots, R_n(l_n, h_n)).$$

Proof. Follows from the definition of semijoin. □

From Lemma 1, any semijoin program will take a database state D into another state expressible as $(R_1(l_1, h_1), \cdots, R_n(l_n, h_n))$ where $l_i \leqq l_{i+1}$ and $h_i \leqq h_{i+1}$ for $i = 1, \cdots, n-1$, and $1 \leqq l_i \leqq i \leqq h_i \leqq n$. In fact, the set of states that can be expressed in this form is exactly the set of states reachable by semijoin programs.

THEOREM 1. *D' is an intermediate state if and only if it is expressible as* $(R_1(l_1, h_1), \cdots, R_n(l_n, h_n))$ *where* $l_i \leqq l_{i+1}$ *and* $h_i \leqq h_{i+1}$ *for* $i = 1, \cdots, n-1$ *and* $1 \leqq l_i \leqq i \leqq h_i \leqq n$.

Proof. That every intermediate state is expressible as $(R_1(l_1, h_1), \cdots, R_n(l_n, h_n))$, $l_i \leqq l_{i+1}$ and $h_i \leqq h_{i+1}$ for $i = 1, \cdots, n-1$, follows from Lemma 1. To show the converse, we let $D' = (R_1(l_1, h_1), \cdots, R_n(l_n, h_n))$ and show how to construct a relevant program

p that produces D' from D (we denote concatenation by "·"):

$p := \varnothing$;
if $l_n \neq n$ **then** $p = \langle x_{l_n} \ x_{l_n+1} \cdots x_{n-1} \rangle$;
for $i = n-1$ **to** 2 **by** -1
  if $(l_i \neq i)$ **and** $(l_i \neq l_{i+1})$
  then $p := p \cdot \langle x_{l_i} \ x_{l_i+1} \cdots x_{i-1} \rangle$
end;
if $h_1 \neq 1$ **then** $p = p \cdot \langle y_{h_1} \ y_{h_1-1} \cdots y_2 \rangle$;
for $i = 2$ **to** $n-1$ **by** 1
  if $(h_i \neq i)$ **and** $(h_i \neq h_{i-1})$
  then $p := p \cdot \langle y_{h_i} \ y_{h_i-1} \cdots y_{i+1} \rangle$
end;

p begins with a sequence of x-type semijoins to generate l_k $(k = 2, \cdots, n)$ for D followed by a sequence of y-type semijoins to generate h_k $(k = 1, \cdots, n-1)$ for D. Thus, $p(D) = D'$. □

By Theorem 1, we can represent each intermediate state by the parameters l_i and h_i for each relation. We define a *join range* to be a pair $[l, h]$ where $1 \leqq l \leqq h \leqq n$ and observe that each intermediate state $(R_1(l_1, h_1), \cdots, R_n(l_n, h_n))$ is represented by a vector of join ranges $\mathbf{J} = ([l_1, h_1], \cdots, [l_n, h_n])$.

We shall freely speak of an intermediate state $\mathbf{J}$ to mean the state represented by $\mathbf{J}$. For relevant program $p = p_1 \cdots p_m$ and initial state D, we represent the sequence of states that p reaches by

$$\mathbf{J}(t) = ([l_1(t), h_1(t)], \cdots, [l_n(t), h_n(t)]), \qquad t = 0, 1, \cdots, m.$$

Since $\mathbf{J}(0)$ represents D, $\mathbf{J}(0) = ([1, 1], \cdots, [n, n])$.

4. Calculating costs of semijoin programs. Since every intermediate state is represented by a vector of join ranges and since we need only calculate the cost of semijoins that are executed in intermediate states, we can use join ranges to express these calculations.[1]

Recall that the cost of a semijoin $R_i \ltimes R_j$ is linearly proportional to the size of the joining column in R_j, that is $|R_j[\mathbf{R}_i \cap \mathbf{R}_j]|$ (cf. (2.1)). If we want to be able to calculate the cost of every relevant semijoin in every intermediate state, then we need to know the size of every joining column in every intermediate state. We represent this information as a set of values $\mathbf{a} = \{a_{ij}(l, h)\}$ defined as follows:

$$a_{i,1}(l, h) = |R_i(l, h)[\mathbf{R}_{i-1} \cap \mathbf{R}_i]|, \qquad 1 \leqq l \leqq i \leqq h \leqq n, \quad i > 1,$$

$$a_{i,2}(l, h) = |R_i(l, h)[\mathbf{R}_i \cap \mathbf{R}_{i+1}]|, \qquad 1 \leqq l \leqq i \leqq h \leqq n, \quad i < n.$$

Then for an intermediate state represented by join range $\mathbf{J}$, we have

$$\text{Cost}\,(x_i, \mathbf{J}, \mathbf{a}) = a_{i,2}(l_i, h_i) + b, \tag{4.1}$$

$$\text{Cost}\,(y_i, \mathbf{J}, \mathbf{a}) = a_{i,1}(l_i, h_i) + b \tag{4.2}$$

where b is the fixed cost per message. (4.1) and (4.2) were obtained by substituting $\mathbf{J}$, $\mathbf{a}$ for the state D in (2.1).

[1] As we have mentioned in § 2.3, many of the results in § 5 and the general approach of this paper are independent of this particular cost model. We only require that $\text{Cost}\,(X_i, \mathbf{R}(\theta))$ be monotonic in $a_{i,2}(l_i(t), h_i(t))$ starting in § 5.4.2.

By applying (4.1) and (4.2), we can express the cost of each program $p = p_1 \cdots p_m$ in terms of $\mathbf{a}$:

$$\mathrm{COST}(p, \mathbf{a}) = \sum_{t=0}^{m-1} \mathrm{Cost}(p_t, \mathbf{J}(t), \mathbf{a}).$$

We will frequently write COST (p) when $\mathbf{a}$ is understood from context.

Given an initial state D, the value of $a_{ij}(l, h)$ for $j = 1, 2$ and $1 \leqq l \leqq i \leqq h \leqq n$ is uniquely defined (although calculating those values may be expensive). If we know these values, then we can in principle calculate the cost of any relevant program applied to D. Insofar as cost calculation is concerned, we can therefore think of the $a_{ij}(l, h)$ values as a *parameterization* of the initial state.

LEMMA 2. (i) $l_i \leqq l'_i \leqq i \leqq h'_i \leqq h_i$ *implies* $a_{ij}(l_i, h_i) \leqq a_{ij}(l'_i, h'_i)$.

(ii) $l \leqq i < h$ *implies* $a_{i,2}(l, h) = a_{i+1,1}(l, h)$.

Proof. (i) $R_i(l_i, h_i) \subseteq R_i(l'_i, h'_i)$ and (ii) $R_i(l, h)[\mathbf{R}_i \cap \mathbf{R}_j] = R_{i+1}(l, h)[\mathbf{R}_i \cap \mathbf{R}_j]$. □

Lemma 2 partitions the parameters into $n-1$ subsets[2] and implies a "$\leqq$" partial ordering on each subset. The ith subset consists of $a_{i,2}(l_i, h_i)$ for all $l_i \leqq i \leqq h_i$ and $a_{i+1,1}(l_{i+1}, h_{i+1})$ for all $l_{i+1} \leqq i+1 \leqq h_{i+1}$. The partial ordering is illustrated in Fig. 1. Note that the parameters $a_{ij}(i, i)$, which correspond to the initial state, are not bounded.

THEOREM 2. *Only* $n(n-1)(n+7)/6$ *parameters are required to calculate the costs of all correct semijoin programs for an order-n chain query.*

Proof. Let $K(i)$ be the number of distinct parameters in the ith partition. By counting, $i(n-i+1)$ distinct parameters correspond to $\{a_{i,2}(l, h) | 1 \leqq l \leqq i \leqq h \leqq n\}$. Only $n-i$ of the remaining parameters, $\{a_{i+1,1}(i+1, h) | i+1 \leqq h \leqq n\}$, are distinct (see Fig. 1). Hence,

$$K(i) = i(n-i+1) + (n-i).$$

Summing over i, the total number of distinct parameters is

$$\sum_{i=1}^{n-1} K(i) = \frac{n(n-1)(n+7)}{6}. \qquad \square$$

Let $\mathbf{a}$ be an $n(n-1)(n+7)/6$ dimension vector of positive integers corresponding to the parameters $\{a_{ij}(l, h)\}$. If $\mathbf{a}$ satisfies the two properties of Lemma 2, then $\mathbf{a}$ is called a *meaningful parameterization* of the initial state. From now on, we assume that the cost function is defined on a meaningful parameterization rather than on D; viz. Cost $(p, \mathbf{a})$ = Cost (p, D) where D is parameterized by $\mathbf{a}$.

5. Reducing the set of potentially optimal programs.

5.1. Motivation. To further simplify our optimization problem, we will characterize a small subset of all relevant programs in which the optimal program must lie. There are three types of relevant programs that we can safely eliminate without losing the optimal program: incorrect programs, those that do not correctly answer the query; redundant programs, those that contain semijoins that do not affect the result produced by the program; and dominated programs, those whose cost is greater than some other program for all initial states. In this section we will define a subset of all relevant programs that contains no incorrect, redundant, or dominated programs.

5.2. Correct semijoin programs for answering chain queries. A *subprogram* of a semijoin program $p = p_1 \cdots p_m$ is an ordered subsequence of p, $p_{t_1} p_{t_2} \cdots p_{t_k}$ where

[2] The initial database state has $n-1$ pairs of joining columns.

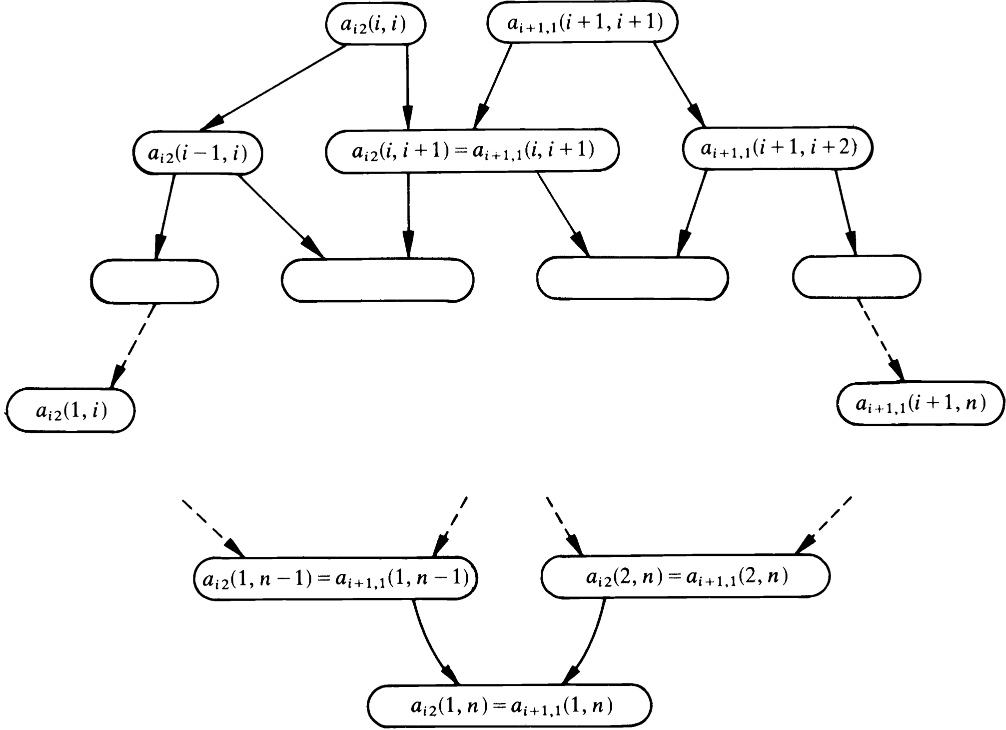


FIG. 1. *Relationships between parameters $a_{ij}(l, h)$. → indicates greater than or equal to.*

$t_i < t_{i+1}$ for $1 \leqq i \leqq k-1$ and $k \leqq m$. The set of semijoin programs that correctly solve Q are exactly those that have the subprogram $y_n y_{n-1} \cdots y_2$.

THEOREM 3. *A semijoin program p is correct for an order n chain query if and only if p has a subprogram equal to $y_n y_{n-1} \cdots y_2$.*

Proof. The if part follows directly from the definitions of chain query and semijoin (a formal proof of a more general result appears as [1, Thm. 1]). To prove the only-if part, suppose p is correct and $\|p\| = m$. Then by Theorem 1 and the definition of correctness, $p(D) = (R_1(1, n), R_2(l_2(m), h_2(m)), \cdots, R_n(l_n(m), h_n(m)))$ such that $1 \leqq l_i(m) \leqq i \leqq h_i(m) \leqq n$, $l_i(m) \leqq l_{i+1}(m)$, and $h_i(m) \leqq h_{i+1}(m)$. Since $h_1(m) = n$, by Lemma 1 there must exist some $t_2 < m$ such that $p_{t_2} = y_2$ and $h_2(t_2 - 1) = n$. Similarly, since $h_2(t_2 - 1) = n$, there must exist some $t_3 < t_2$ such that $p_{t_3} = y_3$ and $h_3(t_3 - 1) = n$. Repeating the argument, $y_n \cdots y_2$ must be a subprogram of p. □

5.3. Nonredundant programs. Let $p = p_1 \cdots p_m$ be a semijoin program. The operation p_t of p is *redundant* in p if for all D either $\mathbf{J}(t) = \mathbf{J}(t-1)$ or $J_1(t-1) = [1, n]$ (i.e., the state $\mathbf{J}(t-1)$ answers the query). For example, if $n = 3$ then the second x_1 is redundant in $x_1 y_3 x_1 y_2$ (because $\mathbf{J}(2) = \mathbf{J}(3)$) and in $x_1 y_3 y_2 x_1$ (because $J_1(3) = [1, 3]$). A program with no redundant operations is called *nonredundant*. The program $x_1 y_3 y_2$ is nonredundant.

THEOREM 4. *A correct program p is nonredundant if and only if p satisfies* (i)–(iv) *below.*

(i) *x_1 appears in p at most once.*

(ii) *y_n appears in p at most once.*

(iii) *Suppose $p = v_1 p_r v_2 p_t v_3$, where v_1, v_2, and v_3 are (possibly empty) subprograms.*

If $p_t = p_r = x_i$ *for some* $i > 1$, *then* $x_{i-1} \in v_2$. *If* $p_t = p_r = y_i$ *for some* $i < n$, *then* $y_{i+1} \in v_2$.

(iv) y_2 *is the last operation of* p.

Proof. (*If*) Suppose p satisfies (i)–(iv) and p is redundant. We show that neither of the two situations that cause redundancy can arise; hence a contradiction.

First, we show by induction that for $p_t \in p$ (arbitrary t), $\mathbf{J}(t) \neq \mathbf{J}(t-1)$. Let p_t be x_1; since x_1 appears in p only once, $\mathbf{J}(t) \neq \mathbf{J}(t-1)$. Let p_t be any x_i $(2 \leqq i \leqq n-1)$, and $\mathbf{J}(t) \neq \mathbf{J}(t-1)$. We show it must also be the case for $p_t = x_{i+1}$. Suppose the contrary. If $J(t) = J(t-1)$, then there exists some r such that $p_r = x_{i+1}$. By (iii), there exists $r < s < t$ such that $p_s = x_i$. By induction, no $p_s = x_i$ can be redundant which means $J_{i+1}(r-1) \supset_{\neq} J_{i+1}(s)$ and hence $J_{i+1}(r-1) \supset_{\neq} J_{i+1}(t-1)$; consequently $J_{i+2}(t-1) \neq J_{i+2}(t)$. So if $p_t = x_{i+1}$, p_t cannot be redundant. A symmetrical argument will show that if $p_t = y_i$ $(i = 2, \cdots, n)$ then $\mathbf{J}(t) \neq \mathbf{J}(t-1)$.

Second, we show that for any $p_t \in p$, $J_1(t-1) \neq [1, n]$. Suppose the contrary is true. Then without loss of generality we can assume $p_t = y_2$. If $J_1(t-1) = [1, n]$ then by Theorem 3 there exist $t_n < t_{n-1} < \cdots < t_3 < t_2 < t$ such that $p_{t_k} = y_k$. Due to (iii), there exist $t_2 < t_3' < t$ and $p_{t_3'} = y_3$. By repeatedly invoking (iii), there must exist $t_{n-1}' < t_{n-1}$ such that $p_{t_{n-1}'} = y_{n-1}$ and so y_n must appear more than once. A contradiction.

(*Only if*) Suppose p is nonredundant and one of (i)–(iv) is violated. We show this leads to a contradiction.

(i) Suppose $p = v_1 p_t v_2 p_r v_3$, where $v_1 v_2$ and v_3 are subprograms and $p_t = p_r = x_1$ (for some $r > t$). Then $\mathbf{J}(t)$ must be such that $J_2(t) = [1, h_2(t)]$. $\mathbf{J}(r)$, however, cannot be different from $\mathbf{J}(r-1)$ except in $l_2(r)$. Since $l_2(r) = l_2(r-1)$, the second x_1 is redundant.

(ii) Follows an argument symmetric to (i).

(iii) Suppose $p = v_1 p_t v_2 p_r v_3$, $p_t = p_r = x_i$ for $i > 1$ and $x_{i-1} \notin v_2$. Then we have

$$l_i(t) = l_i(t+1) = l_i(t+2) = \cdots = l_i(r-1),$$

hence $l_{i+1}(r) = l_{i+1}(r-1)$. This implies $\mathbf{J}(r) = \mathbf{J}(r-1)$. Hence p_r is redundant. For $p_t = p_r = y_i$ and $y_{i+1} \notin v_2$ the symmetrical argument suffices.

(iv) By Theorem 3, p is correct if and only if p has a subprogram $y_n y_{n-1} \cdots y_2$. Let the y_2 of this subprogram be the tth operation in p (and $t < m$); then clearly $J_1(t) = [1, n]$. Hence all operations after p_t, namely $p_{t+1}, \cdots, p_m$, are redundant. □

5.4. A noninferior set of programs. Many correct, nonredundant programs can still be eliminated from consideration. First, if two correct programs have the same cost for all initial states, only one need be considered. For example, $x_1 x_3 y_4 y_3 y_2$ and $x_3 x_1 y_4 y_3 y_2$ are two such programs. And second, if one program is always more costly than another, it too can be eliminated from consideration. For example, let $p = x_2 x_1 y_3 y_2$ and $p' = x_1 x_2 y_3 y_2$ be two correct programs for an order-three chain query. We have

$$\text{COST}(p) = a_{22}(2, 2) + a_{12}(1, 1) + a_{31}(2, 3) + a_{21}(1, 3) + 4b,$$

$$\text{COST}(p') = a_{12}(1, 1) + a_{22}(1, 2) + a_{31}(1, 3) + a_{21}(1, 3) + 4b,$$

$$\text{COST}(p') - \text{COST}(p) = (a_{31}(1, 3) - a_{31}(2, 3)) + (a_{22}(1, 2) - a_{22}(2, 2)) \leqq 0.$$

Since $a_{22}(2, 2)$ can be arbitrarily large, the inequality is strict for some meaningful parameterization $\mathbf{a}$. Thus, p' need not be considered.

We formalize these observations by the concept of dominance (of cost). Given two programs p and p', p *dominates* p' if p and p' are both correct and for all initial states D, $\text{COST}(p) \leqq \text{COST}(p')$. If p and p' dominate each other, then they are

equivalent. If p dominates p' and for some D Cost $(p, D) <$ Cost (p', D), then p *strictly dominates* p'.

A set of programs S is called *noninferior* (with respect to an order-n chain query) if

(i) all programs in S are correct and nonredundant;

(ii) no program in S is strictly dominated by a program not in S; and

(iii) no program in S dominates another program in S.

In this section, we will concisely characterize a noninferior set of programs.

To simplify this characterization, we shall regard every relevant semijoin program as a sequence of *blocks* and write $p = B_1 \cdots B_M$, where B_k denotes the kth block. Each block is required to contain all x-type or all y-type semijoins. Moreover, B_i and B_{i+1} must contain semijoins of different types, for $i = 1, \cdots, M-1$. For example, $p = y_2x_1x_2y_3y_2$ can be written $p = B_1B_2B_3$ where $B_1 = y_2$, $B_2 = x_1x_2$ and $B_3 = y_3y_2$. Notice that each program has a unique representation in terms of blocks. Every relevant program has such a representation, so the notation does not limit the set of programs being considered.

Define NPROG to be the set of relevant programs, $p = B_1 \cdots B_M$, that satisfy the following three conditions.

(i) (boundary property) $B_M = y_ny_{n-1} \cdots y_2$.

(ii) (ordering property) For each $k = 1, \cdots, M$, if B_k is x-type, then the x_i operations in B_k appear in strictly ascending order of their subscripts without duplications. If B_k is y-type, then the y_i operations in B_k appear in strictly descending order of their subscripts without duplication.

(iii) (covering property) For $k = 1, \cdots, M-1$ and $i = 2, \cdots, n-1$:

(a) if x_i is in B_k then y_{i+1} and y_i are in B_{k+1}; x_1 can only be in B_{M-1}; and

(b) if y_i is in B_k then x_{i-1} and x_i are in B_{k+1}; y_n appears only in B_M.

The boundary property ensures that only correct programs are considered. The ordering and covering properties eliminate redundant and dominated programs. To visualize the effect of these latter two properties on programs, let us represent programs in a more graphical notation.

Let $p = B_1 \cdots B_M$ be a program where each B_i $(i = 1, \cdots, M)$ is a block and p satisfies the ordering property. Arrange p as n rows where (1) B_i is written on row i, (2) for each B_i, operations are written in ascending sequence of subscripts, and (3) for each $j \in \{1, \cdots, n\}$, all operations with subscript j are written in the same column.

For example, $p' = x_3y_4y_3x_1x_2x_3x_4y_5y_4y_3y_2$ is written as

$$
\begin{array}{ll}
B_1 & \quad\;\; x_3 \\
B_2 & \quad\;\; y_3y_4 \\
B_3 & x_1x_2x_3x_4 \\
B_4 & \;\; y_2y_3y_4y_5.
\end{array}
$$

Every program represented in this way corresponds to a unique program satisfying the ordering property. Notice that by representing programs in this way, the covering property can be easily checked by ensuring (with the exception of x_1) that each B_i covers B_{i-1} $(i = 2, \cdots, M)$.

THEOREM 5. *For initial database states with meaningful parameterization* **a**, NPROG *is a noninferior set.*

We will prove Theorem 5 in three steps. First, we prove that NPROG is correct and nonredundant (§ 5.4.1). Second, we prove that every relevant program not in NPROG is dominated by one in NPROG (§ 5.4.2). And third, we prove that no program in NPROG is strictly dominated by another program in NPROG (§ 5.4.3).

5.4.1. NPROG is correct and nonredundant. Since every program in NPROG ends with the block $y_n y_{n-1} \cdots y_2$, by Theorem 3 every such program is correct. To prove that every program in NPROG is nonredundant, we must show that the conditions of Theorem 4 are satisfied by every program in NPROG.

LEMMA 3. *Every program in* NPROG *satisfies the conditions of Theorem* 4.

Proof. The boundary and covering properties imply that x_1 and y_n can appear at most once (conditions (i) and (ii) of Theorem 4). Following the ordering property, no two operations in the same block can be the same. If x_i appears more than once, then the covering property requires that x_{i-1} precedes the second x_i; similarly for y_i. That y_2 is the last operation follows from the boundary property (conditions (iii) and (iv) of Theorem 4). □

5.4.2. NPROG dominates all other relevant programs. To show that every relevant program p not in NPROG is dominated by a program in NPROG, we will show how to transform p into a program p' such that $\|p\| = \|p'\|$, p' dominates p and either $p' \in$ NPROG or p' is redundant. If $p' \in$ NPROG then we are done. If p' is redundant, then the redundant operations are removed, producing a program p'', and the transformation process is now applied to p''. At each iteration the program shrinks, so we eventually terminate with a program in NPROG that dominates p. To follow this plan, we first introduce the appropriate program transformations and then apply them to relevant programs not in NPROG.

We will consider transformations that switch the relative order of adjacent operations. For a program p_1 let $f_r(p)$ be the program obtained by swapping p_r and p_{r+1}. That is, if $p = p_1 \cdots p_m$, then $f_r(p) = p_1 \cdots p_{r-1} p_{r+1} p_r \cdots p_m$. To compare the cost and effect of p and $p' = f_r(p)$, let us compare the intermediate states for p and p', $\mathbf{J}(t)$ and $\mathbf{J}'(t)$ respectively ($t = 0, 1, \cdots, m$).

Clearly, $\mathbf{J}(t) = \mathbf{J}'(t)$ for $t = 0, 1, \cdots, r-1$. However, for $t \geqq r$ the relationship between $\mathbf{J}(t)$ and $\mathbf{J}'(t)$ depends on p. The useful transformations on a correct program are given in the following lemma.

LEMMA 4. *Given a correct program* p, $f_r(p)$ *is correct and dominates* p *when neither* (1) *nor* (2) *below is true*:

(1) $P_r = x_i$ *and* ($p_{r+1} = x_{i+1}$ *or* $p_{r+1} = y_{i+1}$);

(2) $P_r = y_i$ *and* ($p_{r+1} = x_{i-1}$ *or* $p_{r+1} = y_{i-1}$).

Proof. For all cases other than (1) and (2), the cost of the rth operation of $f_r(p)$ will be the same as the $(r+1)$th operation of p. The cost of the $(r+1)$th operation of $f_r(p)$ (and those succeeding) will not cost more than the rth operation (and those succeeding $(r+1)$th of p. □

Note that $f_r(p)$ *strictly* dominates p when $p_r = x_i$ and $p_{r+1} = x_{i-1}$, and when $p_r = y_i$ and $p_{r+1} = y_{i+1}$.

Example. (i) $p_a = x_2 x_1 y_4 y_3 y_2$. $f_1(p_a) = x_1 x_2 y_4 y_3 y_2$. $p_a \notin$ NPROG but $f_1(p_a) \in$ NPROG and following Lemma 4, $f_1(p_a)$ dominates p_a.

(ii) $p_b = x_3 y_4 y_3 x_1 y_2$, $p'_b = f_1 f_2 f_3(p_b) = x_1 x_3 y_4 y_3 y_2$. $p_b \notin$ NPROG, but $p'_b \in$ NPROG and dominates p_b.

(iii) $P_c = y_3 x_3 y_4 y_3 y_2$. $P_c = \notin$ NPROG. $p'_c = f_2 f_1(p_c) = x_3 y_4 y_3 y_3 y_2$ is clearly redundant. Following Lemma 4, p'_c dominates p_c. If we remove the redundant operation y_3 from p'_c, then $p''_c = x_3 y_4 y_3 y_2$ is in NPROG. Note p''_c dominates p_c too.

LEMMA 5. *Every relevant program* p *that does not satisfy the following conditions is dominated by some redundant strategy*:

(1) *if* $p_r = p_t = x_i$ *for some* $r < t$, $i > 1$, *then there exist* $r < u < v < w < t$ *such that* $p_u = y_{i+1}$, $p_v = y_i$, $p_w = x_{i-1}$; *and*

(2) *if $p_r = p_t = y_i$ for some $r < t$, $i < n$, then there exist $r < u < v < w < t$ such that $p_u = x_{i-1}$, $p_v = x_i$, $p_w = y_{i+1}$.*

Proof. (1) If $p_w = x_{i-1}$ is missing, then p_t is redundant by Theorem 4, so we have $p_w = x_{i-1}$ for some $r < w < t$. Let p de denoted $P_1 x_i P_2 x_i P_3$; we know there exists $x_{i-1} \in P_2$. However, if there does not exist y_i in between $p_r = x_i$ and (any) $x_{i-1} \in P_2$, then we can move the subprogram of P_2 containing all x_k, y_k, $k \leqq i-1$, to positions preceding $p_r(=x_i)$ by successively applying swapping transformations described in Lemma 4. Suppose the resultant program is p'. p' dominates p (by Lemma 4). But due to Theorem 4, p' is redundant. By the same token, if $p_u = y_{i+1}$ is missing, then the subprogram of P_2 containing x_k, $k \leqq i-1$, and y_k, $k \leqq i$, can be moved to positions preceding $p_r(x_i)$ without increasing the cost. Thus p is again shown to be dominated by a redundant program.

(2) The proof follows a symmetric argument like that above. □

Now we are ready to describe a procedure that, for each relevant program $p \notin$ NPROG, systematically finds a program p', via the transformations described in Lemma 4 and by removing redundancies in p such that p' is in NPROG. That p' dominates p will then follow from Lemma 4.

This procedure is described by Algorithm 1. For blocks of different type, we write B_i *covers* B_j if

$$x_k \in B_j \Rightarrow y_{k+1} y_k \in B_i \quad \forall k \text{ (except } x_1 \in B_j \Rightarrow y_2 \in B_i), \tag{1}$$

$$y_k \in B_j \Rightarrow x_{k-1} x_k \in B_i \quad \forall_k. \tag{2}$$

ALGORITHM 1. An algorithm to transform a relevant program $p \notin$ NPROG into $p' \in$ NPROG.

if $p \in$ NPROG **then return** p
else begin
  $p' :=$ sort (p);
  $p' :=$ dered (p');
  while (p' does not satisfy the covering property) **do**
    $k := N(p)$;
    while (B_k cover B_{k-1}) **do** $k := k-1$ **end**;
    if $k = 1$ **then return** p;
    else begin
      $p' :=$ filter$_k$ (p');
      $p' :=$ sort (p');
      $p' :=$ dered (p');
      end
  end
end

$N(p)$ denotes the number of blocks in p. The functions sort, dered and filter$_k$ are compositions of transformations in Lemma 4:

(a) sort (p): Given any $p = B_1 B_2 \cdots B_M$, sort (p) is a program $B'_1 \cdots B'_m$ such that for each k, B'_k contains the same operations as B_k but B'_k is ordered. In other words, if B'_k is x-type then the x_i operations in B_k appear in strictly ascending order of their subscripts (with possible duplications), and similarly for blocks of y-type with descending order. Sort (p) is implemented by applying f_r (i.e. swapping p_r with p_{r+1}), whenever p_r and p_{r+1} are not in order. From Lemma 4, each such f_r transformation

produces a correct and less (or equally) costly program. Therefore sort (p) is correct and dominates p.

(b) dered (p): Given any program p, if p does not satisfy Lemma 5, then dered (p) is the program derived in the proof of Lemma 5 with the redundant operations removed. In other words, dered (p) is a program that dominates p and is nonredundant in the sense of Lemma 5.

(c) filter$_k$ (p): Given $p = B_1B_2 \cdots B_M$, filter$_k$ is a sequence of swapping transformations applied to operations in blocks B_{k-1} and B_k. If $x_i \in B_{k-1}$ and $y_{i+1} \notin B_k$, then x_i is swapped with all the operations in B_k; similarly, if $y_i \in B_{k-1}$ and $x_{i-1} \notin B_k$, then y_i is swapped with B_k. This sequence of swaps starts with the last operation of B_{k-1}. Following Lemma 4, all such swaps will leave the resulting program dominating p.

LEMMA 6. *Algorithm* 1 *terminates and returns a* p' *that dominates* p *in* NPROG.

Proof. If the algorithm terminates and returns p', then it is easy to see that $p' \in$ NPROG. That p' dominates p follows because each of sort, dered and filter$_k$ is a transformation of p into a dominant program. So we only have to show that the procedure terminates. Suppose B'_k does not cover B'_{k-1}. If B'_k is the last block, then filter (p') allows the redundant operations in B'_{k-1} to be identified and eliminated. If B'_k is not the last block, then we claim $B'_{k-1} \cap B_{k+1}$ must be empty since otherwise it implies a violation of the condition in Lemma 5 and the redundancy should have been removed by the previous dered $(\cdot)$ transformation. If $B'_{k-1} \cap B'_{k+1}$ is empty, then filter$_k$ will merge B'_{k-1} with B'_{k+1} and B'_{k-2} with B'_k, and p has two fewer blocks. So we have shown that whenever B'_k does not cover B'_{k-1} there is redundancy, and each time we branch through filter$_k$, redundancy is removed and the program is shorter. Since there are only finitely many redundant operations for finite $\|p\|$, the procedure must terminate. □

Lemma 6 concludes part 2 of our proof of Theorem 5.

5.4.3. No program in NPROG strictly dominates another program in NPROG. We need to show that for any p^1 and p^2 in NPROG, we can find a state D with meaningful parameterization $\mathbf{a}$ such that COST $(p^1, \mathbf{a}) <$ COST $(p^2, \mathbf{a})$. The simpler case is when p^2 is a subprogram of p^1.

LEMMA 7. *For* $p^1, p^2 \in$ NPROG *if* p^2 *is a subprogram of* p^1 *then* p^1 *does not dominate* p^2.

Proof. Let $\mathbf{a}$ be such that $a_{ij}(l, h) = a_{ij}(i, i)$ for all j and $l \leqq i \leqq h$. Then clearly the cost of each operation p_t of program p is independent of t and is constant. Since p^1 has all the operations in p^2, we must have

$$\text{COST}(p^1, \mathbf{a}) > \text{COST}(p^2, \mathbf{a}). \qquad \square$$

LEMMA 8. *If* p^2 *is not a subprogram of* p^1, *then* p^1 *does not dominate* p^2.

Proof. Suppose

$$p^1 = B_1^1 B_2^1 \cdots B_{M1}^1 = p_1^1 p_2^1 \cdots p_{m1}^1,$$

$$p^2 = B_1^2 B_2^2 \cdots B_{M2}^2 = p_1^2 p_1^2 \cdots p_{m2}^2.$$

Since p^2 is not a subprogram of p^1, there must exist $k < m2$ such that $p^1_{m1-t} = p^2_{m2-t}$ for $t = 0, 1, \cdots, k-1$ and $p^1_{m1-k} \neq p^2_{m2-k}$. We show that there exists $j < k$ such that the cost of p^1_{m1-j} can be arbitrarily higher than that of p^2_{m2-j} because of p^2_{m2-k}. Let $p^2_{m2-k} = x_i$. Then we can find $p^2_{m2-j} = p^1_{m1-j} = y_{i+1}$ or x_{i+1} (when $p^2_{m2-k} = y_i$ then we can find $p^2_{m2-j} = p^1_{m1-j} = y_{i-1}$ or x_{i-1}). However, for the cost of p^1_{m1-j} not to be bounded by the cost of any other operation in p^2, then p^2_{m2-j} must be the first operation of its kind to appear in p^2. (In other words, there does not exist $r < m2-j$ such that

$p_r^2 = p_{m2-j}^2$.) We can choose j as follows:

(a) If $p_{m2-k}^2 = x_i$ and $x_{i+1} \cdots x_{i+r}$ are in the same block and x_{i+r+1} is not in the same block, then for $r>0$ $p_{m2-j}^2 = x_{i+r}$, and for $r=0$ $p_{m2-j}^2 = y_{i+1}$.

(b) If $p_{m2-k}^2 = y_i$ and $y_{i-1} \cdots y_{i-r}$ are in the same block and y_{i-r-1} is not in the same block, then for $r>0$ $p_{m2-j}^2 = y_{i-r}$, and for $r=0$ $p_{m2-j}^2 = x_{i-1}$.

The existence of such a j is guaranteed by the covering and boundary properties of p. Now we can find **a** such that

$$\text{Cost}\,(p_{m1-j}^1, \mathbf{J}^1(m1-j), \mathbf{a})\ \text{COST}\,(p^2, \mathbf{a}). \qquad \square$$

This completes the proof of Theorem 5.

5.5. Some properties of NPROG. Some interesting properties of NPROG follow from Theorems 4 and 5 and Lemma 5.

COROLLARY 1. *If* $p = B_1B_2 \cdots B_M \in \text{NPROG}$, *then* $M \leqq n$.

Proof. Condition 2 of Lemma 5 and the covering property requires each block to increase in length by at least 1, except that the last block may have the same length as the $(M-1)$st block. Since the maximum length a block can have is $n-1$, we have the result. □

COROLLARY 2. *Let* $p_{\max}$ *and* $p_{\min}$ *be the longest and shortest programs of* NPROG. *Then we have*

$$p_{\max} = B_1B_2 \cdots B_n$$

where

$$B_n = y_ny_{n-1} \cdots y_2, \qquad B_{n-1} = x_1x_2 \cdots x_{n-1},$$

$$B_{n-k} = \begin{cases} y_{n-k/2} \cdots y_{k/2}, & k = 2, 4, \cdots, n-1\ (\textit{even}), \\ x_{(k+1)/2} \cdots x_{n-(k-1)/2}, & k = 1, 3, \cdots, n-1\ (\textit{odd}), \end{cases}$$

$$P_{\min} = y_ny_{n-1} \cdots y_2, \qquad \|P_{\max}\| = \frac{n(n+1)}{2} - 1, \qquad \|p_{\min}\| = n-1,$$

and both $p_{\max}$ *and* $p_{\min}$ *are unique.* □

The proof is straightforward and is omitted. By adapting the proof technique of Theorem 5, we can verify that $p_{\max}$ is optimal for some meaningful parameterization **a**. This is a somewhat unexpected result of the analysis.

COROLLARY 3. *Let* N_n *denote the number of distinct programs in* NPROG. *Then* N_n *is bounded by*

$$\frac{5+\sqrt{5}}{10}\left(\frac{3+\sqrt{5}}{2}\right)^{n-1} + \frac{5+\sqrt{5}}{10}\left(\frac{3-\sqrt{5}}{2}\right)^{n-1} \leqq N_n \leqq 2^{n(n+1)/2-1}.$$

Proof. The upper bound can be obtained by observing that every program of NPROG is a subprogram of $p_{\max}$. Since the converse is not true, the bound is not tight.

The lower bound can be obtained by converting only those programs in which each block is a consecutive sequence of semijoins. This gives us the recursive equation

$$f(n) = 1 + \sum_{j=1}^{n} \sum_{i=1}^{j-1} f(j-1) = 1 + \sum_{k=1}^{n-1} kf(n-k)$$

where $f(n)$ is the lower bound. Since we know $f(1)=1$ and $f(2)=2$, the solution $f(n)$ is easily obtained. □

Corollary 3 suggests that a brute force exhaustive search for the optimal program from the minimal noninferior set NPROG is feasible only for small n. In the next section, we present an efficient algorithm that finds the optimal program for arbitrary n.

6. Searching for the optimal program.

6.1. Search by dynamic programming. The dynamic programming algorithm that we will describe searches for the optimal program by comparing all programs in the noninferior set in the most efficient way. The algorithm recursively computes and compares the costs of those subprograms that solve the same subquery. The inferior subprograms are abandoned since they cannot be part of the optimal program. This process terminates with the computed cost of only one program, which must be the optimal program.

Let $P_i(l, h)$ denote the *optimal subprogram* that terminates and computes $R_i(l, h)$, as defined in § 3. We know that the optimal subprogram to compute $R_i(i, i)$ is to do nothing, viz. $P_i(i, i) = \emptyset$, the empty program, for all i. This is the boundary condition for our recursive algorithm.

LEMMA 9. *$P_i(l, h)$ is one of the following subprograms:*

(i) $P_{i-1}(l, h) \cdot x_{i-1}$, *provided* $l < i$;

(ii) $P_{i+1}(l, h) \cdot y_{i+1}$, *provided* $i < h$;

(iii) $P_{i-1}(l, i-1) \cdot x_{i-1}$, *provided* $i = h$;

(iv) $P_{i+1}(i+1, h) \cdot y_{i+1}$, *provided* $i = l$;

(v) $P_{i-1}(l, i-1) \cdot x_{i-1} \cdot P_{i+1}(i+1, h) \cdot y_{i+1}$, *provided* $l < i < h$ *(or arranged in reverse*: $P_{i+1}(i+1, h) \cdot y_{i+1} \cdot P_{i-1}(l, i-1) \cdot x_{i-1}$; *the cost of these two programs is the same).*

Proof. The proof consists of two parts. First, the (sub)programs (i)–(v) all compute $R_i(l, h)$; second, no other (sub)program that computes $R_i(l, h)$ is a subprogram of a noninferior program. The first part follows simply from induction. If we assume that each subprogram in (i)–(v) computes the corresponding subquery, then the composition $P_i(l, h)$ clearly computes $R_i(l, h)$. To prove the second part, let us examine those programs that compute $R_i(l, h)$ but are not included in (i)–(v). First, $P_i(l, h)$ cannot be composed of a subprogram that is not an optimal subprogram. This leaves us with programs only of the following form:

(vi) $P_{i-1}(l, k) \cdot x_{i-1} \cdot P_{i+1}(j, h) \cdot y_{i+1}$ where $l \leqq j \leqq i$, $i-1 \leqq k \leqq h$, or $l \leqq j \leqq i+1$, $i \leqq k \leqq h$.

(vii) $P_{i+1}(j, h) \cdot y_{i+1} \cdot P_{i-1}(l, k) \cdot x_{i-1}$ where $l \leqq j \leqq i$, $i-1 \leqq k \leqq h$, or $l \leqq j \leqq i+1$, $i \leqq k \leqq h$.

Consider (vi). If $P_{i+1}(j, h)$ has the subprogram x_i followed by $y_h y_{h-1} \cdots y_{i+2}$, then $P_{i-1}(l, k) \cdot x_{i-1} \cdot P_{i+1}(j, h)$ computes $R_{i+1}(l, h)$ and must be dominated by $P_{i+1}(l, h)$. Else, we can show that (vi) violates the covering property as follows. $P_{i-1}(l, k)$ must contain

$$x_l x_{l+1} \cdots x_{i-2} \quad \text{and} \quad y_k y_{k+1} \cdots y_i$$

and $P_{i+1}(j, h)$ must contain

$$x_j x_{j+1} \cdots x_i \quad \text{and} \quad y_h y_{h-1} \cdots y_{i+2}.$$

If we have $l \leqq j \leqq i$ and $i-1 \leqq k \leqq h$, then $x_j x_{j+1} \cdots x_i$ (which is in a different block from $x_l x_{l+1} \cdots x_{i-2}$ due to the earlier argument) cannot "cover"[3] $x_l x_{l+1} \cdots x_{i-2}$ since

[3] An x-block "covers" another x-block means that the former covers a y-block which in turn covers the latter.

$j \geqq l$. On the other hand, if $l \leqq j \leqq i+1$ and $i \leqq k \leqq h$, then $y_k y_{k+1} \cdots y_i$ cannot be covered by $x_l x_{l+1} \cdots x_{i-2}$ even with x_{i-1} in the same block.

So we have shown that (vi) either is dominated by $P_{i+1}(l, h) \cdot y_{i+1}$ or violates the covering property, and therefore cannot be a candidate for $P_i(l, h)$. A similar argument will verify that (vii) cannot be $P_i(l, h)$ either. □

Lemma 9 essentially outlines the algorithm. $P_i(l, h)$ is obtained recursively by comparing costs of those candidates listed in Lemma 9, namely:

$$\text{COST}(P_i(l,h)) = \begin{cases} \min\{\text{COST}(P_{i-1}(l,h)\cdot x_{i-1}), \text{COST}(P_{i-1}(l,i-1)\cdot x_{i-1})\} & \text{if } i=h, \quad l \leqq i-1 \\ \min\{\text{COST}(P_{i+1}(l,h)\cdot y_{i+1}), \text{COST}(P_{i+1}(i+1,h)\cdot y_{i+1})\} & \text{if } i=l, \quad h \geqq i+1 \\ \min\{\text{COST}(P_{i-1}(l,h)\cdot x_{i-1}), \text{COST}(P_{i+1}(l,h)\cdot y_{i+1}), \\ \qquad \text{COST}(P_{i-1}(l,i-1)\cdot x_{i-1}\cdot P_{i+1}(i+1,h)\cdot y_{i+1})\} & \text{if } l<i<h. \end{cases}$$

The costs on the right-hand side satisfy respectively the equations

$$\text{COST}(P_{i-1}(l,h)\cdot x_{i-1}) = \text{COST}(P_{i-1}(l,h)) + a_{i+1,2}(l,h) + b,$$

$$\text{COST}(P_{i+1}(l,h)\cdot y_{i+1}) = \text{COST}(P_{i+1}(l,h)) + a_{i+1,1}(l,h) + b,$$

$$\text{COST}(P_{i-1}(l,i-1)x_{i-1}\cdot P_{i+1}(i+1,h)\cdot y_{i+1}) \\ = \text{COST}(P_{i-1}(l,i-1)\cdot x_{i-1}) + \text{COST}(P_{i+1}(i+1,h)\cdot y_{i+1}).$$

With the boundary condition $\text{COST}(P_i(i,1)) = 0$ for all i. The ultimate goal is to find $P_1(1,n)$ and $\text{COST}(P_1(1,n))$.

Let us illustrate the algorithm by an example. Consider an order-3 chain query. The optimal strategy $P_1(1,3)$ can be derived by recursively applying Lemma 9 and comparing the different alternatives. The result is the creation of a decision tree as shown in Fig. 2. The leaves of the tree are composed either of $P_i(i,i)$, which we know is the empty program, or of its grandparent subprogram (indicated by dashed back-arrows). In the latter case, it is not necessary to look any further since the optimal $P_i(l,h)$ cannot contain itself as a (strict) subprogram.

The optimal program is found based on the database state parameterization $a_{ij}(l,h)$, $i = 1, \cdots, n$, $l \leqq i$, $i \leqq h$ and $j = 1, 2$, and on the given initiation cost. For our example, suppose we have

$$\begin{array}{ll} a_{12}(1,1) = 100, & a_{22}(2,2) = 400, \\ a_{21}(2,2) = 500, & a_{31}(3,3) = 350, \\ a_{21}(1,2) = a_{21}(1,2) = 100, & a_{22}(1,2) = 200, \\ a_{21}(2,3) = 400, & a_{22}(2,3) = a_{31}(2,3) = 300, \\ a_{12}(1,3) = a_{21}(1,3) = 50 & a_{22}(1,3) = a_{31}(1,3) = 100, \end{array}$$

$$b = 100.$$

Note that this $\mathbf{a} = \{a_{ij}(l,h)\}$ satisfies the properties of Lemma 2 and is thus a meaningful parameterization of the database state. Now the costs of the optimal subprograms

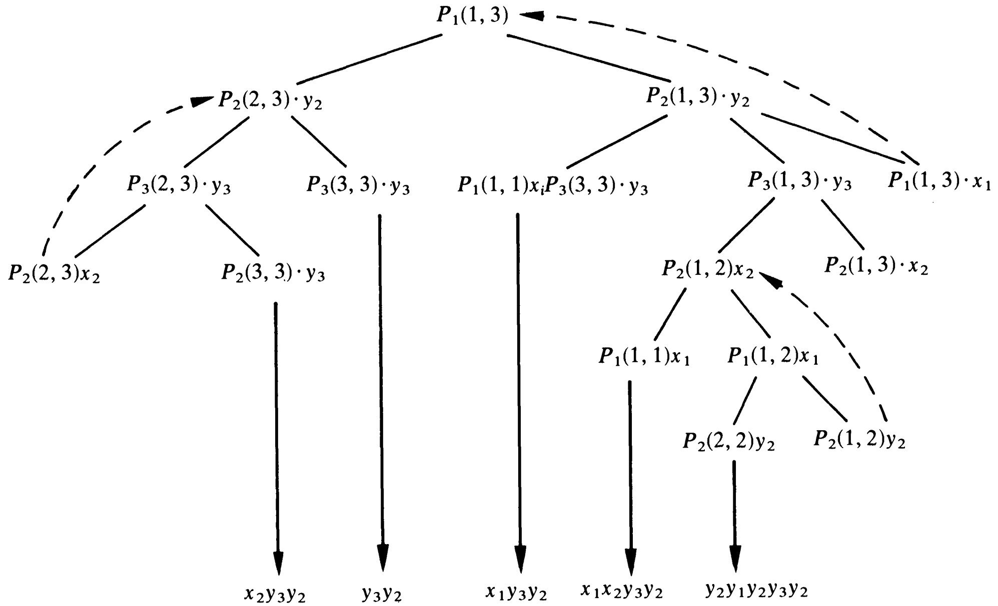


FIG. 2. *Decision tree for optimal subprograms.*

can be recursively calculated (see Fig. 3). For this example, the optimal program is $x_1y_3y_2$ and the cost is 800. By using dynamic programming, we avoid having to compute the cost of each (noninferior) program. Instead, we compute the *optimal* costs for *all* the subprograms of noninferior programs, $P_i(l, h)$, $1 \leqq l \leqq i \leqq h \leqq n$. The

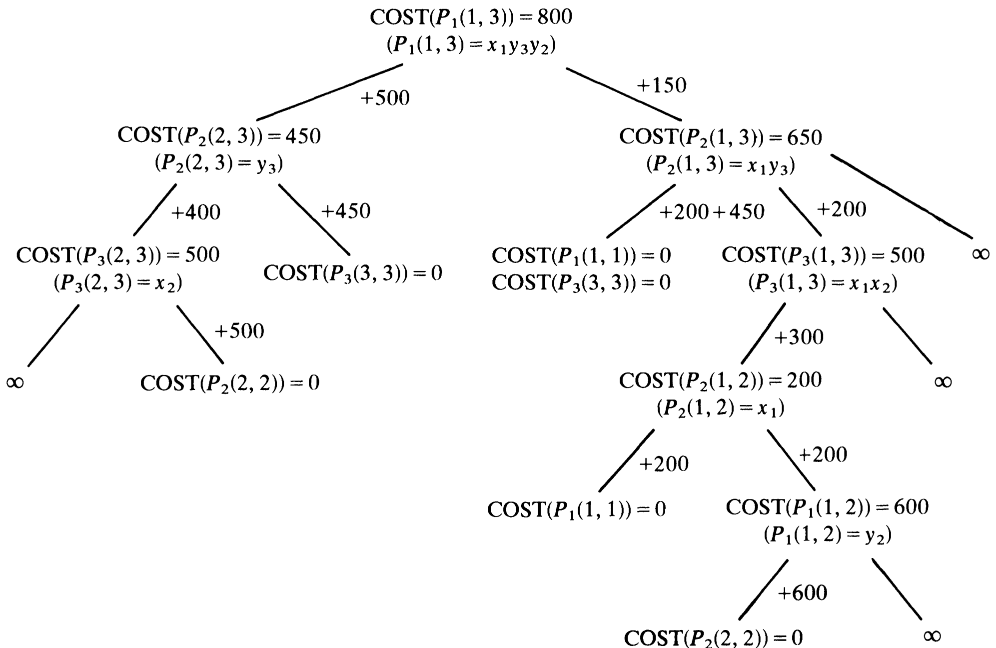


FIG. 3. *Cost of optimal subprograms.*

number of such optimal subprograms is

$$K = \sum_{i=1}^{n} i(n-i+1) = \frac{n(n+1)(n+2)}{6}, \qquad (*)$$

and the cost of each $P_i(l, h)$ can be calculated based on at most 4 subprograms whose costs are already known (due to Lemma 9). So we can make the following statement:

THEOREM 6. *There is a dynamic programming algorithm that finds the optimal strategy and the cost of an order n chain query. The time complexity of the algorithm is linear in the input size.*

The time complexity follows from the fact that input size is $O(n^3)$ (Theorem 2) and the number of comparisons in the dynamic programming algorithm is also $O(n^3)$ (Lemma 9 and eq. (*)). Also, notice that the analysis in the previous sections on noninferior sets not only serves a pedagogic purpose in our understanding of the problem, but also is important in making the dynamic programming algorithm efficient (through Lemma 9).

6.2. Optimization based on partial parameterization. In practice, of course, it is unrealistic to assume that the complete parameterization of the database state is available to the query processor. In this section, we briefly discuss some approaches to optimization based on partial information about the database state.

Let the parameterization $\mathbf{a}$ be composed of two components $\mathbf{a} = (\mathbf{a}_1, \mathbf{a}_2)$, where $\mathbf{a}_1$ corresponds to those parameters known (exactly) to the query processor and $\mathbf{a}_2$ corresponds to parameters that are unknown, or for which only some statistical behavior is known. For chain queries, it is reasonable to assume that

$$\mathbf{a}_1 = \{a_{ij}(i, i), i = 1, \cdots, j = 1, 2\}.$$

This subset of the parameters corresponds to the number of distinct values in each column of the relations before any semijoin is applied. The following two measures can be taken for such an incomplete characterization.

First, knowing $\mathbf{a}_1$ may have severely restricted the values $\mathbf{a}_2$ could take. Let

$$\text{MP}(A) \triangleq \{\mathbf{a}: \mathbf{a} = (\mathbf{a}_1, \mathbf{a}_2) | \mathbf{a}_1 \in A \text{ and } \mathbf{a} \text{ is a meaningful parameterization}\}.$$

NPROG is the minimal noninferior set of programs for database states parameterized by meaningful parameterizations $\mathbf{a}$. Let NPROG (MP (A)) be the corresponding set for states $\mathbf{a} \in \text{MP}(A)$. Depending on MP (A), NPROG (MP (A)) may be left with a handful of candidates. For example, the set A for which the shortest program $p_{\min}$ is optimal can be explicitly described.

THEOREM 7. *Let A be the set of $\mathbf{a}_1 = \{a_{ij}(i, i), i = 1, \cdots, n, j = 1, 2\}$ such that*

$$a_{i+1,1}(i+1, i+1) \leqq b + \frac{a_{i,2}(i, i)}{(n-i+1)}, \qquad i = 1, 2, \cdots, n-1.$$

Then NPROG (MP (A)) *has only one member, $p_{\min}$, and therefore $p_{\min}$ is optimal.*

The proof is in [4]. We will just give a sketch here. Suppose that n is large, so $a_{i+1,1}(i+1, i+1) \leqq b$. Then

$$\begin{aligned} \text{COST}(p_{\min}, \mathbf{a}) &= \sum_{k=2}^{n} a_{k,1}(k, n) + (n-1)b \\ &\leqq \sum_{k=2}^{n} a_{k,1}(k, k) + (n-1)b \\ &\leqq 2(n-1)b. \end{aligned}$$

In fact, as a rule of thumb, when the size of the database is "small" relative to the set-up cost b (measured in appropriate units), it is sufficient to consider only the shorter programs (for example, programs with no more than two blocks) to find the optimal, or a good suboptimal, program.

Secondly, knowing $\mathbf{a}_1$, we may consider a probabilistic model by treating $\mathbf{a}_2$ as random variables conditioning on $\mathbf{a}_1$. In this case, the optimal program is only optimal in a statistical sense, viz. in minimizing

$$E[\text{COST}(p, \mathbf{a})|\mathbf{a}_1] = \text{COST}(p, E[a_2|\mathbf{a}_1]).$$

Here $g(\mathbf{a}_1) \triangleq E[\mathbf{a}_2|\mathbf{a}_1]$ is but a function of $\mathbf{a}_1$. Again, g defines a subset of meaningful parameterizations,

$$\{\mathbf{a}|\mathbf{a} = (\mathbf{a}_1, g(\mathbf{a}_1)) \text{ is a meaningful parameterization}\} \triangleq \text{MP}(g).$$

Let us denote the minimal noninferior set of programs for the states $\mathbf{a} \in \text{MP}(g)$ by NPROG (MP (g)). For a given g, NPROG (MP (g)) is usually a much smaller subset of NPROG. We demonstrate this point again by describing a function g such that NPROG (MP (g)) $= \{p_{\min}\}$.

THEOREM 8. *Let* $g(\mathbf{a}_1)$ *be given by*

$$a_{i,1}(i, k) = a_{i,1}(i, i) \quad \text{for } i = 1, 2, \cdots, n-1, \quad k > i,$$

$$a_{i,2}(k, i) = a_{i,2}(i, i) \quad \text{for } i = 2, 3, \cdots, n, \quad k < i.$$

Then NPROG (MP (g)) *contains* $p_{\min}$ *only, and* $p_{\min}$ *is therefore optimal.*

Proof. $p_{\min}$ incurs cost

$$\text{COST}(p_{\min}, (\mathbf{a}_1, g(\mathbf{a}_1))) = \sum_{k=2}^{n} a_{k,1}(k, n) + (n-1)b$$

$$= \sum_{k=2}^{n} a_{k,1}(k, k) + (n-1)b.$$

Any other program would clearly only incur higher cost. □

Following this example, another rule of thumb is: If the intermediate relations are not expected to be reduced much in comparison to the initial database, then the shortest program is the optimal or a good suboptimal one.

7. Generalizations and concluding remarks. The methodology that we used to study the problem

QUERY = Chain queries
STATE = Arbitrary finite relational database
PROG = Semijoin programs
COST = Linear function of "data moved"

can be applied to study more general problems. One generalization is to consider QUERY to be the larger set of queries called *tree queries*, defined in [1]. It can be inferred from [1] that the parameterization of the initial state for this problem will still be a function of n. A detailed exposition is given in [6].

We believe that we have made a viable preliminary study of the query optimization problem. Future work should let cost reflect local processing cost, and allow programs to contain other operations, such as join. We would also like to see more complicated parameterizations of the initial state, and increased dimensionality of the correct programs. The key should again be in obtaining the suitable NPROG for different meaningful parameterizations of the database state.

REFERENCES

[1] P. A. BERNSTEIN AND D. M. CHIU, *Using semijoins to solve relational queries*, J. Assoc. Comput. Mach., 28 (1981), pp. 25–40.

[2] P. A. BERNSTEIN AND N. GOODMAN, *The power of natural semijoins*, this Journal, 10 (1981), pp. 751–771.

[3] P. A. BERNSTEIN, N. GOODMAN, E. WONG, C. L. REEVE AND J. B. ROTHNIC, JR., *Query processing in a system for distributed database* (SDD-1), ACM Trans. Database Systems, 6 (1981), pp. 602–625.

[4] D. D. CHAMBERLIN et al., *Sequel* 2: *A unified approach to data definition, manipulation and control*, IBM J. Res. Develop., 20, 6 (Nov. 1976), pp. 560–575.

[5] D. M. CHIU, *Optimal query interpretation for distributed database*, Ph.D. Thesis, Division of Applied Sciences, Harvard Univ., Cambridge, MA, December 1979.

[6] D. M. CHIU AND Y. C. HO, *A methodology for interpreting tree queries into optimal semi-join expressions*, Proc. 1980 SIGMOD Conference, Association for Computing Machinery, New York, 1981.

[7] E. F. CODD, *Relational Completeness of Data Base Sublanguages*, Courant Computer Symposia Series, 6, Prentice-Hall, Englewood Cliffs, NJ, 1972.

[8] A. R. HEVNER AND S. B. YAO, *Query processing in distributed database systems*, IEEE Trans. Software Engrg., SE-S (1979), pp. 177–187.

[9] E. WONG, *Retrieving dispersed data in* SDD-1: A system for distributed database, in Proc. 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, May 1977, pp. 217–235.

[10] C. T. YU AND M. Z. OZSOYOGLU, *An algorithm for tree query membership of a distributed query*, in Proc. IEEE Compsac, Chicago, Nov. 1979, pp. 306–312.

SIAM J. COMPUT.
Vol. 13, No. 1, February 1984

© 1984 Society for Industrial and Applied Mathematics
0097-5397/84/1301-0010 $01.25/0

ALTERNATING PUSHDOWN AND STACK AUTOMATA*

RICHARD E. LADNER†, RICHARD J. LIPTON‡ AND LARRY J. STOCKMEYER§

Abstract. The classes of languages accepted by alternating pushdown automata, alternating stack automata, and alternating nonerasing stack automata, both with and without an auxiliary space bounded worktape, are characterized in terms of complexity classes defined by time bounded deterministic Turing machines. It is also shown that alternating 2-way finite state machines accept only regular languages.

Key words. alternation, pushdown automata, stack automata, 2-way finite state automata, computational complexity

1. Introduction. Alternating Turing machines were introduced by Chandra, Kozen and Stockmeyer [2] as an interesting generalization of nondeterministic Turing machines. In this paper we investigate the effect of adding alternation to auxiliary pushdown automata, first investigated by Cook [3], and to auxiliary stack automata, first investigated by Ibarra [12]. We characterize the power of alternating auxiliary pushdown automata (Alt-Aux-PDA), alternating auxiliary stack automata (Alt-Aux-SA) and alternating auxiliary nonerasing stack automata (Alt-Aux-NESA) in terms of time bounded deterministic Turing machines. The word "auxiliary" in these models refers to an auxiliary space bounded worktape that the machine has in addition to the pushdown store or stack. A *stack* is like a pushdown store except that the interior contents of the stack can be read, but not changed except by the usual pushing or popping. In a *nonerasing* stack automaton, the stack cannot be popped. See Table 1 for a summary of our results in comparison to known results concerning deterministic and nondeterministic versions of these classes of automata. There are several interesting things to note about Table 1. For each type of auxiliary machine the deterministic and nondeterministic versions have exactly the same power while the alternating version has strictly more power. Alternating auxiliary stack automata and nonerasing stack automata have exactly the same power while it is open whether or not nondeterministic auxiliary stack automata are more powerful than their nonerasing counterpart. Chandra, Kozen and Stockmeyer [2] proved that alternating Turing machines with space bounded by $s(n)$ have exactly the power of deterministic Turing machines that run in time $2^{cs(n)}$ for some $c>0$. It is open whether or not DTIME $(2^{cs(n)})$ properly includes DSPACE $(s(n))$. So it is not known whether or not the addition of a pushdown store alone or alternation alone increases the power of space bounded Turing machines. However, if *both* alternation and a pushdown store are added to a space bounded Turing machine then a more powerful device results.

Two-way pushdown automata and stack automata without auxiliary storage were first investigated by Gray, Harrison and Ibarra [7] and Ginsburg, Greibach, and Harrison [6], respectively. There is no known characterization of 2-way pushdown automata (either deterministic or nondeterministic) in terms of time or space bounded

* Received by the editors June 2, 1982, and in revised form March 11, 1983. Portions of this paper have been reprinted with permission from R. E. Ladner, R. J. Lipton and L. J. Stockmeyer, *Alternating pushdown automata*, Proceedings of the 19th Annual Symposium on Foundations of Computer Science, Ann Arbor, MI, 1978, © 1978 IEEE. This work was supported in part by the National Science Foundation under grants MCS 77-02474 and MCS 78-81486.

† Department of Computer Science, University of Washington, Seattle, Washington 98195.

‡ Department of Electrical Engineering and Computer Science, Princeton University, Princeton, New Jersey 08544.

§ Mathematical Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

TABLE 1

Characterization of auxiliary pushdown and stack automata in terms of time or space bounded Turing machines. The function $s(n)$ bounds the space used on the auxiliary worktape of the pushdown and stack automata. Unions are over all constants $c > 0$, and $s(n)$ is assumed to be at least $\log n$.

	Deterministic	Nondeterministic	Alternating
AUX-PDA(s(n))	$\cup$ DTIME($2^{cs(n)}$) Cook	$\cup$ DTIME($2^{cs(n)}$) Cook	$\cup$ DTIME($2^{2^{cs(n)}}$)
AUX-SA(s(n))	$\cup$ DTIME($2^{2^{cs(n)}}$) Ibarra	$\cup$ DTIME($2^{2^{cs(n)}}$) Ibarra	$\cup$ DTIME($2^{2^{2^{cs(n)}}}$)
AUX-NESA(s(n))	$\cup$ DSPACE($2^{cs(n)}$) Ibarra	$\cup$ DSPACE($2^{cs(n)}$) Ibarra	$\cup$ DTIME($2^{2^{2^{cs(n)}}}$)

Turing machines. Hopcroft and Ullman have characterizations of both deterministic and nondeterministic nonerasing stack automata in terms of time or space bounded Turing machines [10] and Cook has such characterizations of the erasing versions of stack automata [3]. Chandra, Kozen and Stockmeyer [2] show that alternating PDA's have at least the power of deterministic exponential time. Table 2 summarizes our results and past results concerning pushdown automata and stack automata. There are also several interesting things to notice about Table 2. For each type of machine

TABLE 2

Characterization of 2*-way pushdown and stack automata in terms of time or space bounded Turing machines.*

	Deterministic	Nondeterministic	Alternating
PDA	———	———	$\cup$ DTIME(2^{cn})
SA	$\cup$ DTIME($2^{cn \log n}$) Cook	$\cup$ DTIME(2^{cn^2}) Cook	$\cup$ DTIME($2^{2^{cn}}$)
NESA	DSPACE(n log n) Hopcroft and Ullman	NSPACE(n^2) Hopcroft and Ullman	$\cup$ DTIME($2^{2^{cn}}$)

the alternating version is more powerful than either the deterministic or nondeterministic version. This is true for pushdown automata because the languages accepted by nondeterministic PDA's are also accepted by deterministic Turing machines that run in polynomial time. Alternating stack automata and their nonerasing versions are equivalent while it is not known whether or not deterministic (nondeterministic) stack automata and their nonerasing versions are equivalent.

We also investigate alternating 2-way finite state automata (Alt-2-FA) and show that these automata accept only regular languages. Moreover, an m-state Alt-2-FA can be simulated by a 2^{m2^m}-state deterministic 1-way finite state automaton. This result is applied in our simulation of alternating stack automata; we mirror arguments as in [3], [10], [12] for nondeterministic stack automata where reading into the stack is eliminated in a way similar to the elimination of the left moves of a 2-way finite automaton.

Apart from providing fairly precise characterizations of the power of alternating PDA's and stack automata, another motivation for this work was the hope that the results could be applied to help understand the computational complexity of certain problems, in the same way that alternating Turing machines [2] have been applied in classifying the computational complexity of certain problems in logic (Berman [1], Fischer and Ladner [4], Kozen [14]), combinatorial games (Fraenkel and Lichtenstein [5], Stockmeyer and Chandra [19]), and communicating sequential processes (Ladner [15]). One application of our results has been made: Lewis [16] has used the characterizations of alternating pushdown automata and alternating stack automata (without an auxiliary worktape) to prove exponential and double-exponential lower bounds on the time complexity of certain decidable subcases of first-order predicate calculus.

In several instances our proofs for alternating machines generalize in a natural way corresponding proofs for nondeterministic machines. The concept of a computation tree for alternating machines is quite analogous to the concept of a computation sequence for nondeterministic machines. This analogy is the basis for our generalized proofs. This is further evidence that alternation is a valid generalization of nondeterminism. Alternation is truly a robust concept in complexity theory.

Regarding other related work, other authors, including Gurari and Ibarra [8], King [13], and Ruzzo [17] give relationships between alternating Turing machines on the one hand and deterministic or nondeterministic auxiliary pushdown or stack automata on the other. However, these papers do not consider alternating auxiliary pushdown or stack automata. Gurari and Ibarra [9] place upper bounds on the power of alternating auxiliary pushdown and stack automata when the auxiliary space bound is augmented by bounds on other complexity measures such as stack reversals or acceptance tree size.

In § 2 we discuss the precise meaning of an alternating computation and apply it to Turing machines. Sections 3, 4 and 5 contain the results on alternating PDA's, finite automata and stack automata, respectively.

2. Alternating automata. Because alternation is a fairly new concept it pays to take time to study the meaning of an alternating computation and in particular the meaning of acceptance by an alternating machine. It helps to contrast the definition of alternating automata with that of nondeterministic automata. Think of an *automaton* as a set $\mathcal{I}$ of *instantaneous descriptions* (ID's) together with

(i) an *initialization function* INIT which takes inputs to ID's,

(ii) a set $\mathcal{A} \subseteq \mathcal{I}$ of *accepting* ID's,

(iii) a *transition relation* $\Rightarrow \subseteq \mathcal{I} \times \mathcal{I}$ with the property that for all I, $\{J \mid I \Rightarrow J\}$ is finite.

An *alternating automaton* has the added property

(iv) a set $\mathcal{U} \subseteq \mathcal{I}$ of *universal* ID's.

The members of $\mathcal{I} - \mathcal{U}$ are called *existential* ID's. If $I \Rightarrow J$, we say that J is a *successor* of I.

Let M be an automaton. A *computation* of M is a finite nonempty sequence $I_0, I_1, \cdots, I_n$ such that $I_i \Rightarrow I_{i+1}$ for $0 \leqq i < n$. An *accepting computation* of M on an input x is a computation $I_0, I_1, \cdots, I_n$ such that $I_0 = \text{INIT}(x)$ and $I_n \in \mathcal{A}$. We say M *accepts* x if there is an accepting computation of M on input x.

Let N be an alternating automaton. By contrast, a *computation* or *computation tree* of N is a finite, nonempty labeled tree with the properties:

(a) each node π of the tree is labeled with an ID, $l(\pi)$;

(b) if π is an internal node (a nonleaf) of the tree, $l(\pi)$ is universal and $\{J \mid l(\pi) \Rightarrow J\} = \{J_1, \cdots, J_k\}$, then π has exactly k children $\rho_1, \cdots, \rho_k$ such that $l(\rho_i) = J_i$;

(c) if π is an internal node of the tree and $l(\pi)$ is existential then π has exactly one child ρ such that $l(\pi) \Rightarrow l(\rho)$.

An *accepting computation* (*tree*) of N on an input x is a computation tree whose root is labeled with INIT (x) and whose leaves each have labels in $\mathscr{A}$. We say N *accepts* x if there is an accepting computation of N on input x. Define

$$L(N) = \{x \mid N \text{ accepts } x\}.$$

Deterministic and nondeterministic automata are special cases of alternating automata. An alternating automaton is nondeterministic if $\mathscr{U} = \varnothing$ and is deterministic if for each I there is at most one J such that $I \Rightarrow J$.

We extend the complexity concepts of space and time to alternating computations. With each alternating automaton N we associate a *space complexity function* SPACE which takes ID's to natural numbers. Informally, SPACE (I) is the storage "used" by the ID I. We say that N is $s(n)$*-space bounded* if for all n and for all x of length n, if x is accepted by N then there is an accepting computation tree of N on input x such that for each node π of the tree, SPACE $(l(\pi)) \leqq s(n)$. We say that N is $t(n)$*-time bounded* if for all n and for all x of length n, if x is accepted by N then there is an accepting computation tree of N on input x of height $\leqq t(n)$.

We now apply these concepts to the Turing machine. Formally an alternating Turing machine is an object of the form

$$M = (Q, q_0, U, F, \Sigma, \Gamma, \delta)$$

where

Q is a finite set of states,
$q_0 \in Q$ is the start state,
$U \subseteq Q$ is the set of universal states,
$F \subseteq Q$ is the set of accepting states,
Σ is the input alphabet (¢, $\$ \notin \Sigma$ serve as left and right endmarkers),
Γ is the tape alphabet ($\# \in \Gamma$ is the blank symbol),
$\delta: Q \times (\Sigma \cup \{¢, \$\}) \times \Gamma \to \mathscr{P}(Q \times \{R, L\}^2 \times (\Gamma - \{\#\}))$ is the transition function,

where $\mathscr{P}$ denotes the power set operation, and $R(L)$ signifies a right (left) shift of a head. Of course, we are defining a Turing machine with a read-only input tape and one read/write worktape.

An ID has the form (q, x, i, α, j) where $q \in Q$ is the state, $x \in \Sigma^*$ is the input, i with $0 \leqq i \leqq |x| + 1$ is the position of the input head, $\alpha \in (\Gamma - \{\#\})^*$ is the nonblank portion of the worktape, and j with $0 \leqq j \leqq |\alpha| + 1$ is the position of the worktape head. An accepting ID has its first coordinate in F while a universal ID has its first coordinate in U. INIT (x) is $(q_0, x, 0, \lambda, 0)$, where λ denotes the empty word. The transition relation between ID's is straightforward to define (see, for example, [11]). Finally SPACE $((q, x, i, \alpha, j)) = |\alpha|$. Define

ASPACE $(s(n)) = \{L(M) \mid M$ is an $s(n)$-space bounded alternating Turing machine$\}$.

Chandra, Kozen and Stockmeyer [2] prove the following useful theorem.

THEOREM 2.1 (Chandra, Kozen, Stockmeyer). *If* $s(n) \geqq \log n$ *then*

$$\text{ASPACE}(s(n)) = \bigcup \text{DTIME}(2^{cs(n)}).$$

Throughout the paper, unions $\bigcup$ are over all constants $c > 0$. Here, DTIME $(t(n))$ is the class of languages recognized by $t(n)$-time bounded deterministic Turing machines. Note that since we define deterministic machines as a special case of

alternating machines, the time bound $t(n)$ is not imposed for rejected inputs; indeed, a deterministic machine is not required to halt on rejected inputs. Of course, if $t(n)$ is real-time countable, the machine can be modified to halt within $t(n)$ steps for all inputs.

Remark 2.2. Theorem 2.1 continues to hold even if the input head on the alternating machine is only allowed to move in one direction as can be seen from the proof of this result in [2].

To see this another way let M be a normal 2-way $s(n)$-space bounded alternating Turing machine with $s(n) \geqq \log n$. To construct a 1-way alternating machine M' we have M' simulate M step by step. In order to simulate the reading of an input symbol, M' maintains on its tape a count of the input head position of M. When M reads an input symbol, M' guesses the symbol using an existential state and then enters a universal state to choose one of two further actions: one action is to continue the simulation of M and the other is to check that the symbol guessed is actually in the input position indicated by the count. This latter action is the only time the input head of M' moves and it can do so 1-way.

In stating our results, the time bounds involve exponentiation iterated to two or three levels, so we define notation for iterated exponentiation. Define the function $E(k, z)$ for integer $k \geqq 0$ by

$$E(0, z) = z, \qquad E(k+1, z) = 2^{E(k,z)}.$$

3. Alternating auxiliary pushdown automata. The definition of an alternating auxiliary pushdown automaton is similar to Cook's definition of a nondeterministic auxiliary pushdown automaton [3] except that, as described in § 2, a subset of the states are designated as universal states. Formally, an *alternating auxiliary pushdown automaton* (Alt-Aux-PDA) is an object of the form

$$M = (Q, q_0, U, F, \Sigma, \Gamma, \Delta, ¢, \delta)$$

where

Q is a finite set of states,
$q_0 \in Q$ is the start state,
$U \subseteq Q$ is the set of universal states,
$F \subseteq Q$ is the set of accepting states,
Σ is the input alphabet ($¢, \$ \notin \Sigma$),
Γ is the auxiliary worktape alphabet ($\# \in \Gamma$ is the blank symbol),
Δ is the pushdown store alphabet,
$¢ \in \Delta$ is the bottom symbol on the pushdown store,
δ is the transition function where

$$\delta: Q \times (\Sigma \cup \{¢, \$\}) \times \Gamma \times \Delta \rightarrow \mathcal{P}(Q \times \{L, R, S\}^2 \times (\Gamma - \{\#\}) \times ((\Delta - \{¢\}) \cup \{\text{POP}, \text{IDLE}\})).$$

Recall that $\mathcal{P}$ is the power set operator. The input is read-only and is delimited on the left by ¢ and the right by \$. If the machine is in state q scanning σ on the input tape, γ on the auxiliary worktape, and p on the top of the pushdown store, and if $(q', d_1, d_2, \gamma', p')$ belongs to $\delta(q, \sigma, \gamma, p)$, then the machine can enter state q', shift the input head in direction d_1 (Left, Right, or Stationary), write γ' on the auxiliary worktape and shift the worktape head in direction d_2, and manipulate the pushdown store by either (i) *pushing* p' onto the pushdown store if $p' \in \Delta - \{¢\}$, (ii) *popping* the top symbol if $p' = \text{POP}$, or (iii) *idling*, that is, not changing the pushdown store if $p' = \text{IDLE}$. We assume that δ has been constrained so that the bottom symbol ¢ is never popped nor pushed, and that the input head is never shifted outside the area

delimited by the endmarkers (inclusive). Note also that the blank symbol cannot be written on the auxiliary worktape. Other inessential assumptions which help smooth out our proofs are that the machine enters an accepting state only if it is reading the bottom pushdown symbol ¢, and the machine behaves deterministically while it is either pushing or popping, that is, if D belongs to the range of δ and if D contains more than one move, then all moves in D are of the form $(q', d_1, d_2, \gamma', \text{IDLE})$.

An ID has the form $(q, x, i, \alpha, j, \beta)$ where

$q \in Q$ indicates the current state,
$x \in \Sigma^*$ is the input,
i, where $0 \leqq i \leqq |x|+1$, indicates the input head position,
$\alpha \in (\Gamma - \{\#\})^*$ indicates the contents of the nonblank portion of the auxiliary worktape,
j, where $0 \leqq j \leqq |\alpha|+1$, indicates the worktape head position,
$\beta \in ¢\Delta^*$ indicates the contents of the pushdown store.

The function INIT is defined by

$$\text{INIT}(x) = (q_0, x, 0, \lambda, 0, ¢).$$

Accepting ID's are those of the form $(q, x, i, \alpha, j, ¢)$ where $q \in F$. Universal ID's are those of the form $(q, x, i, \alpha, j, \beta)$ where $q \in U$. The transition relation is straightforward but tedious to define formally. The space of an ID is counted only on the auxiliary worktape; that is

$$\text{SPACE}((q, x, i, \alpha, j, \beta)) = |\alpha|.$$

The definition of $L(M)$, the set of words in Σ^* that M accepts, and the definition of M being $s(n)$-space bounded now follow from the general definitions given in § 2. Define

$$\text{ALT-AUX-PDA}(s(n)) = \{L(M) | M \text{ is an Alt-Aux-PDA that is } s(n)\text{-space bounded}\}.$$

Our principal result concerning alternating auxiliary PDA's is the following characterization.

THEOREM 3.1. *If* $s(n) \geqq \log n$, *then*

$$\text{ALT-AUX-PDA}(s(n)) = \bigcup \text{DTIME}(E(2, cs(n))).$$

Proof. (1) We first show that for any $c > 0$

$$\text{DTIME}(E(2, cs(n))) \subseteq \text{ALT-AUX-PDA}(s(n)).$$

We do this indirectly by showing that

$$\text{ASPACE}(2^{cs(n)}) \subseteq \text{ALT-AUX-PDA}(s(n))$$

and appealing to Theorem 2.1.

Let M be an alternating Turing machine which is $2^{cs(n)}$-space bounded. Let Q be the states and Γ be the worktape alphabet of M. We can of course assume that M has only one tape which is 1-way infinite to the right and that $c \geqq 1$. Therefore, the ID's of M for an input of length n can be viewed as words of length $2^{cs(n)}$ in $\Gamma^* \cdot (Q \times \Gamma) \cdot \Gamma^*$; the meaning of ID $\mu(q, \gamma)\nu$, where $\mu, \nu \in \Gamma^*$, $q \in Q$, $\gamma \in \Gamma$, is that $\mu\gamma\nu$ is written on the first $2^{cs(n)}$ squares of the tape, and M is in state q scanning γ.

For example

$$\text{INIT}(x) = (q_0, x_1)x_2x_3 \cdots x_n \#^d$$

where $x = x_1x_2 \cdots x_n$ and $d = 2^{cs(n)} - n$.

We can assume without loss of generality that each ID of M has at most two successors. This is enforced by assuming that the transition function of M is a partial function of the form

$$\delta : Q \times \Gamma \to (Q \times \{L, R, S\} \times (\Gamma - \{\#\}))^2.$$

For certain $(q, \gamma) \in Q \times \Gamma$, the two components of $\delta(q, \gamma)$ can be identical (indicating a deterministic move). $\delta(q, \gamma)$ is undefined iff q is an accepting state of M. For ID's α and β we write $\alpha \Rightarrow_1 \beta$ (resp., $\alpha \Rightarrow_2 \beta$) iff $|\alpha| = |\beta|$ and α can reach β in one move according to the first (second) component of δ. (The condition $|\alpha| = |\beta|$ ensures that ID's remain $2^{cs(n)}$-space bounded. For example, if $\alpha = \mu(q, \gamma)$ and the first component of $\delta(q, \gamma)$ moves the head right, then there is no β such that $\alpha \Rightarrow_1 \beta$.)

Assume for the moment that $s(n)$ is space constructible [11]. Fix an input x of length n. The Alt-Aux-PDA M' which simulates M first lays off a block of $s(n)$ tape squares on its auxiliary worktape. In general, the pushdown store of M' will contain a string of the form

$$\alpha_0 m_1 \alpha_1 m_2 \alpha_2 m_3 \alpha_3 \cdots$$

where $\alpha_0, \alpha_1, \cdots$ are ID's of M, $m_i \in \{1, 2\}$ for $i \geqq 1$ (we assume $1, 2 \notin Q \cup \Gamma$), $\alpha_0 =$ INIT (x), and $\alpha_{i-1} \Rightarrow_{m_i} \alpha_i$ for $i \geqq 1$. The symbol m_i is chosen by a universal (existential) branch of M' if α_{i-1} is a universal (existential) ID of M. The words α_i for $i \geqq 1$ are chosen by existential branching (i.e., the usual nondeterministic "guessing"). After each extension of α_i by a new guessed symbol, say γ, M' enters a universal state to choose one of two further actions: one action is to continue choosing α_i; the other action is to check that γ is consistent with a legal move of M according to $\Rightarrow_{m_i}$ and to accept or reject accordingly. To perform this check we use the fact that if $\alpha_{i-1} \Rightarrow_m \alpha_i$, then the jth symbol of α_i is uniquely determined by m and the $(j-1)$th, jth, and $(j+1)$th symbols of α_{i-1}. The auxiliary worktape of M' is used as a counter to measure the distance (roughly) $2^{cs(n)}$ between the jth symbol of α_i and the jth symbol of α_{i-1}. Of course, the pushdown store must be popped to perform the check. Notice that the alternation of M' is being used in two ways. First, in choosing the m_i it is used directly to simulate the alternation of M. Second, alternation is used to existentially guess and universally check the ID's of M. Universal branching permits one branch of the computation to read into the pushdown store, popping and destroying information, while another branch retains the information for future use.

It is useful to make more precise the fact that $\alpha \Rightarrow_m \beta$ can be checked by performing local checks within α and β. For a word α, let $\alpha(j)$ denote the jth symbol of α for $1 \leqq j \leqq |\alpha|$. For $m = 1, 2$ there is a function

$$f_{M,m} : (\Gamma \cup Q \times \Gamma \cup \{1, 2\})^3 \to \Gamma \cup Q \times \Gamma$$

such that if α and β are ID's of M with $|\alpha| = |\beta| = l$, then $\alpha \Rightarrow_m \beta$ iff

$$\beta(j) = f_{M,m}(\alpha(j-1), \alpha(j), \alpha(j+1)) \quad \text{for } 1 < j < l,$$

$$\beta(1) = f_{M,m}(1, \alpha(1), \alpha(2)) = f_{M,m}(2, \alpha(1), \alpha(2)),$$

$$\beta(l) = f_{M,m}(\alpha(l-1), \alpha(l), 1) = f_{M,m}(\alpha(l-1), \alpha(l), 2).$$

We now describe the procedures of M' more carefully. The *and*'s ($\wedge$) and *or*'s ($\vee$) in these procedures are implemented by using alternation; that is, $A \wedge B$ (resp., $A \vee B$) means to enter a universal (resp., existential) state to choose which one of A or B to perform.

INIT: Push INIT (x) onto the pushdown store;
(The $s(n)$ auxiliary storage is used as a counter to ensure that INIT (x) is the correct length $2^{cs(n)}$);
Call TOP.

TOP: If the top ID is accepting, then accept; else
If the top ID is universal then ((push 1 $\wedge$ push 2); call NEW); else
If the top ID is existential then ((push 1 $\vee$ push 2); call NEW).

NEW: $l \leftarrow 0$;
C: $l \leftarrow l+1$;
If $l > 2^{cs(n)}$ then call TOP;
Existentially guess a $\gamma \in \Gamma \cup Q \times \Gamma$ and push γ;
(call CHECK $\wedge$ go to C).

CHECK: Remember the top pushdown symbol, say γ, in the finite control;
Pop $2^{cs(n)}$ symbols off the pushdown store, and at the point when a symbol $m \in \{1, 2\}$ is popped, remember m in the finite control;
Pop and remember three more symbols, say γ_3, γ_2, and γ_1;
If $\gamma = f_{M,m}(\gamma_1, \gamma_2, \gamma_3)$ then accept; else reject.

M' executes INIT. A proof that M accepts x iff M' accepts x can be based on the obvious correspondence between computation trees of M and computation trees of M'. Figure 1 shows the correspondence for a universal configuration α of M with two successors β_1 and β_2. Although the existential branch in the procedure NEW can push any symbol γ onto the pushdown store, only the correct choice will survive the subsequent call to CHECK. Further details of the proof that M' correctly simulates M are left to the reader.

If $s(n)$ is not constructible, then M' guesses the value of $s(n)$ using existential branching. If M accepts x, then M' accepts x after guessing the correct value $s(n)$, so M' is $s(n)$-space bounded. If M does not accept x, then for no guessed value of $s(n)$ does M' accept x.

(2) To prove that

$$\text{ALT-AUX-PDA}\,(s(n)) \subseteq \bigcup \text{DTIME}\,(E(2, cs(n)))$$

we generalize the proof of Cook [3] that an $s(n)$-space bounded nondeterministic auxiliary PDA can be simulated by a $2^{cs(n)}$-time bounded deterministic Turing machine. Let M be an $s(n)$-space bounded Alt-Aux-PDA and let x be an input of length n. A *surface* ID is an object of the form (q, x, i, α, j, B) where q indicates the current state, x is the input, i indicates the input head position, α indicates the contents of the auxiliary worktape, j indicates the auxiliary worktape head position, and B indicates the *top* symbol on the pushdown store. Let

$$\text{top}\,(q, x, i, \alpha, j, B) = B.$$

Recall that the PDA behaves deterministically when pushing or popping. Hence only when the pushdown store is idle can there be true branching in the automaton. With this in mind we can think of each surface ID as being in one of three possible *modes*:

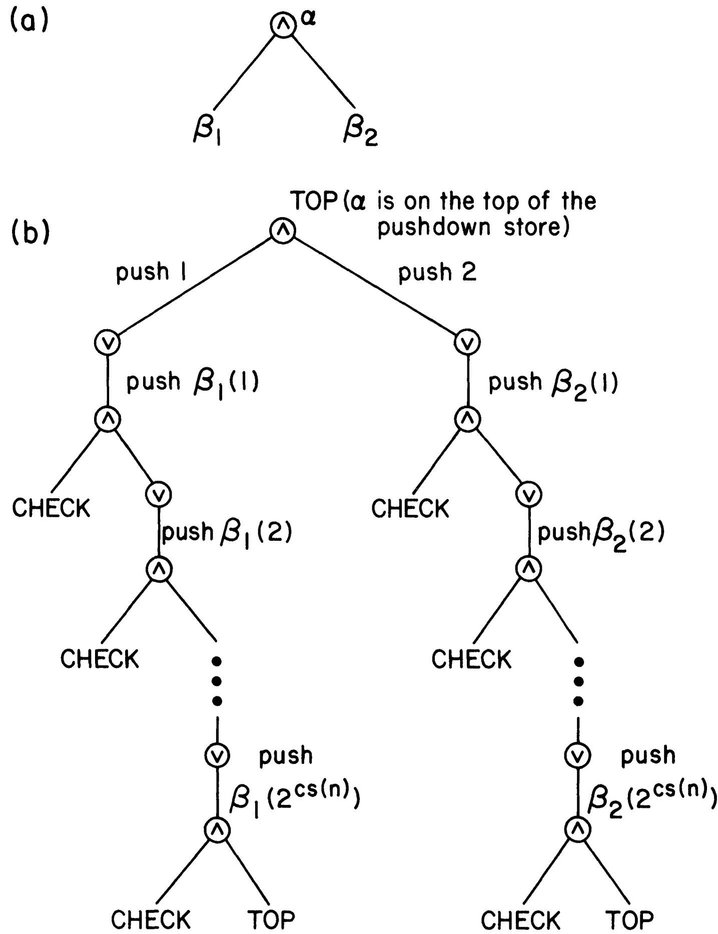


FIG. 1. (a) *A universal branch of the Turing machine M*; (b) *the corresponding portion of an accepting computation of the* Alt-Aux-PDA *M'*.

PUSH, POP, or IDLE. IDLE surface ID's are partitioned into U-IDLE and E-IDLE surface ID's depending on whether they are universal or existential. We can view the transition relation $\Rightarrow$ of M as a relation on surface ID's provided that the first component of the relation is not in POP mode.

A *surface computation* is a finite rooted tree whose nodes are labeled with surface ID's and which has the following properties:

(a) On each path from the root to a leaf, the sequence of modes traversed (including the mode of the root but not the mode of the leaf) can be generated by the following grammar SURFACE:

$$\langle S\rangle \to \lambda$$

$$\langle S\rangle \to \text{IDLE}$$

$$\langle S\rangle \to \langle S\rangle\langle S\rangle$$

$$\langle S\rangle \to \text{PUSH}\langle S\rangle\text{POP}.$$

That is, the PUSH's and POP's are matched analogously to left and right parentheses.

(b) A PUSH node has exactly one child labeled with the surface ID obtained by running the PDA for one move. A POP node has exactly one child whose label is obtained by running the PDA for one move and looking back along the path to the root to the matching PUSH node to obtain the top symbol of the pushdown store (this is possible because of (a)).

(c) If a node is labeled with a U-IDLE surface ID r, then, for each w such that $r \Rightarrow w$, the node has a child with label w. If a node is labeled with an E-IDLE surface ID r, then the node has exactly one child and the child is labeled w for some w with $r \Rightarrow w$.

The surface ID (q, x, i, α, j, B) is *$s(n)$-bounded* iff $|\alpha| \leqq s(|x|)$. A surface computation is *$s(n)$-bounded* iff all surface ID's in the computation are $s(n)$-bounded. If r is an $s(n)$-bounded surface ID and W is a set of $s(n)$-bounded surface ID's, then we write

$$\vDash r \rightarrow W$$

iff there is an $s(n)$-bounded surface computation T whose root is labeled r and whose leaf labels are contained in W (see Fig. 2). Because M accepts only when it reads

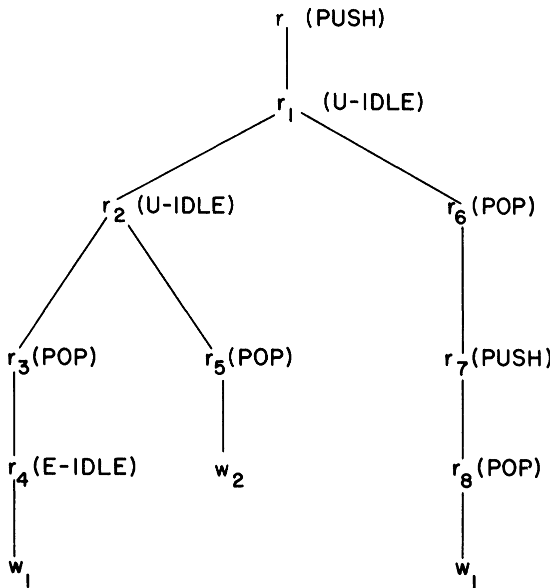


FIG. 2. *A surface computation which witnesses* $\vDash r \rightarrow \{w_1, w_2, w_9\}$.

the bottom symbol ¢, then x is accepted by M iff $\vDash I(x) \rightarrow A$ for some set A of accepting surface ID's, where

$$I(x) = (q_0, x, 0, \lambda, 0, ¢)$$

is the initial surface ID.

We now define a kind of proof system. In this system we will "prove" *terms* of the form $r \rightarrow \{w_1, \cdots, w_k\}$ where $r, w_1, \cdots, w_k$ are $s(n)$-bounded surface ID's. The

system has the following *proof rules*:

1. $$\frac{}{r \to \{r\}}$$

2. (a) $$\frac{r \to W,\ W \subseteq V}{r \to V}$$

(b) $$\frac{r \to W \cup \{w\},\ w \to V}{r \to W \cup V}$$

3. (a) $$\frac{\begin{array}{l} r\ \text{(E-IDLE)} \\ \quad | \\ w \end{array}}{r \to \{w\}}$$

(b) $$\frac{\begin{array}{c} r\ \text{(U-IDLE)} \\ \swarrow\ \ |\ \ \searrow \\ w_1\ \ w_2, \cdots, w_k \end{array}}{r \to \{w_1, \cdots, w_k\}}$$

4. $$\frac{\begin{array}{lllll} r\ \text{(PUSH)} & & v_1\ \text{(POP)} & & v_k\ \text{(POP)} \\ | & & | & & | \\ w, & w \to \{v_1, \cdots, v_k\}, & z_1, & \cdots, & z_k, \quad \text{top}(z_i) = \text{top}(r) \text{ for } 1 \leqq i \leqq k \end{array}}{r \to \{z_1, \cdots, z_k\}}.$$

In each proof rule, the term below the horizontal bar can be concluded from the information above the bar. Lower case letters stand for $s(n)$-bounded surface ID's and upper case letters stand for sets of $s(n)$-bounded surface ID's. In 3(a), r is an existential surface ID and $r \Rightarrow w$. In 3(b), r is a universal surface ID and $\{w_1, \cdots, w_k\} = \{w | r \Rightarrow w\}$; in order to apply this rule, *all* successors of r must be $s(n)$-bounded. In (4), w is the successor of the surface ID r which is in PUSH mode; furthermore, z_i is the successor of v_i which is in POP mode and top $(z_i) = $ top (r) for all i. We write

$$\vdash r \to W$$

if the term $r \to W$ is provable in this system.

For x fixed, there are at most

$$bn \cdot s(n) \cdot 2^{b's(n)} \leqq 2^{b''s(n)}$$

$s(n)$-bounded surface ID's for some constants $b, b', b'' > 0$ (recall that $s(n) \geqq \log n$), so there are at most $E(2, ds(n))$ terms for some $d > 0$. This is the key to our $E(2, cs(n))$-time bounded simulation of M. In order to prove that the simulation is correct, we first show that the proof rules are sound and complete.

LEMMA 3.2. *Let r be a surface* ID *and let G be a set of surface* ID'*s. Then*

$$\vdash r \to G \textit{ iff } \models r \to G.$$

Proof. (*if*) Let T be an $s(n)$-bounded surface computation whose root is labeled r and whose leaf labels are contained in G. The proof is by induction on the *size* (i.e., the number of nodes) of T. If the size of T is 1, then $r \in G$, so $\vdash r \to G$ by rules 1 and 2(a). If the size of T is greater than 1, we consider three cases. In each case, we describe how to break T into pieces such that there is a proof for each piece by induction and such that the pieces can be put together using the proof rules to give a proof of $\vdash r \to G$.

(i) r is an E-IDLE surface ID. Let ω be the (unique) child of the root of T, and let w be the label of ω. Let T' be the maximal subtree of T with root ω. Note that T' witnesses $\models w \to G$. Since the size of T' is less than the size of T, it follows by induction that $\vdash w \to G$. But $\vdash r \to \{w\}$ by rule 3(a), so $\vdash r \to G$ by rule 2(b).

(ii) r is a U-IDLE surface ID. Let $\omega_1, \cdots, \omega_k$ be the children of the root of T, and let w_i be the label of ω_i for $1 \leqq i \leqq k$. Let T_i be the maximal subtree of T with root ω_i, and note that T_i witnesses $\models w_i \to G$. Now $\vdash r \to \{w_1, \cdots, w_k\}$ by rule 3(b) and $\vdash w_i \to G$ by induction, so $\vdash r \to G$ by rule 2(b).

(iii) r is a PUSH surface ID. Let ρ be the root and ω be the child of the root of T. Let w be the label of ω. If τ and ξ are nodes of T, we write surf (τ, ξ) to indicate that either $\tau = \xi$, or ξ is a proper descendant of τ and the sequence of modes traversed on the path from τ to ξ (including the mode of τ but not the mode of ξ) can be generated by the grammar SURFACE defined above. Let T' be the (unique) subtree of T such that the root of T' is ρ and, for all nodes ξ of T' with $\xi \neq \rho$, surf (ρ, ξ) iff ξ is a leaf of T'. In other words, we obtain T' by pruning each branch of T at the node where the pushdown store first returns to the same level as the root r. Let $\zeta_1, \cdots, \zeta_m$ be the leaves of T'. For $1 \leqq i \leqq m$, let ν_i be the parent of ζ_i and let v_i be the label of the node ν_i. (Note that the surface ID's $v_1, \cdots, v_m$ are not necessarily distinct. We let $\{v_1, \cdots, v_m\}$ denote the set, not the multiset, of surface ID's among $v_1, \cdots, v_m$.) Since surf (ρ, ζ_i) but not surf (ρ, ν_i), it follows that surf (ω, ν_i) and v_i must be in POP mode for all i. Therefore, for $1 \leqq i \leqq m$, the label of ζ_i must be z_i where v_i pops to yield z_i and top $(z_i) =$ top (r) for all i. By deleting the nodes $\rho, \zeta_1, \cdots, \zeta_m$ from T' we obtain a surface computation which witnesses $\models w \to \{v_1, \cdots, v_m\}$, so $\vdash w \to \{v_1, \cdots, v_m\}$ by induction. By applying rule 4

$$\vdash r \to \{z_1, \cdots, z_m\}. \tag{3.1}$$

Now for $1 \leqq i \leqq m$, let T_i be the maximal subtree of T with root ζ_i; note that T_i witnesses $\models z_i \to G$, so $\vdash z_i \to G$. Combining this with (3.1) using rule 2(b) gives $\vdash r \to G$.

The root of T cannot be in POP mode by condition (a) in the definition of surface computation, so the proof by induction is complete.

(*Only if*) This is proved by induction on the length of a proof that $r \to G$. If the last step of the proof (the step from which $r \to G$ is concluded) is an application of rule 1, 3(a), or 3(b), then it is immediate by the definition of surface computation that $\models r \to G$. If the last step is an application of rule 2(a), then the surface computation which witnesses the antecedent term also witnesses the conclusion term. If the last step uses rule 2(b) with $G = W \cup V$, then by induction there are surface computations T and T' which witness $\models r \to W \cup \{w\}$ and $\models w \to V$, respectively. If each leaf of T which is labeled w is replaced by the root of T' (which is also labeled w), the resulting tree witnesses $\models r \to W \cup V$. In the case that the last step uses rule 4, the induction step follows by a similar pasting of surface computations, and we leave this case to the reader. □

Assuming that $s(n)$ is space constructible, the deterministic Turing machine M' which simulates M generates all provable terms simply by applying the proof rules until no new terms can be proved. Initially the set $\mathcal{S}$ of provable terms is empty. At a given stage, M' attempts to generate a new term by applying the proof rules to $\mathcal{S}$ in all possible ways. If t is the number of terms, then the time for each stage is polynomial in t. Since there are at most t stages, the total time is polynomial in t, that is, $E(2, cs(n))$ for some $c > 0$. M' accepts x iff there is a provable term $I(x) \to A$ where A is a set of accepting surface ID's. So $L(M) = L(M')$ by Lemma 3.2.

If $s(n)$ is not constructible, then M' iterates the above procedure for $s(n)= 1, 2, 3, \cdots$ until it accepts. If M does accept x, then M' will discover this fact when $s(n)$ reaches its correct value. The total time is a geometric series which is dominated by the last term $E(2, cs(n))$. This completes the proof of Theorem 3.1. □

Theorem 3.1 can easily be generalized to cases where $s(n)$ grows slower than $\log n$. In part (1) of the proof, the $s(n)$ storage is used only as a counter with capacity $2^{cs(n)}$. For general $s(n)$, the input head can be used in conjunction with the $s(n)$ storage to provide a counter with capacity $n2^{cs(n)}$. In part (2) of the proof, the key quantity is the number of $s(n)$-bounded surface ID's (for x fixed); in general, $n2^{cs(n)}$ is an upper bound on their number. Therefore, we have actually proved the following.

THEOREM 3.3. *If* $s(n) \geqq 1$ *then*

$$\text{ALT-AUX-PDA}\ (s(n)) = \bigcup \text{DTIME}\ (E(1, n2^{cs(n)})).$$

Letting ALT-PDA denote the class of languages accepted by alternating PDA's without an auxiliary worktape, and taking $s(n)=1$ in Theorem 3.3, we obtain the following corollary claimed in Table 2.

COROLLARY 3.4. ALT-PDA $= \bigcup$ DTIME (2^{cn}).

4. Alternating 2-way finite automata. In this section we show how to simulate an alternating 2-way finite automaton with m states by a deterministic 1-way finite automaton with 2^{m2^m} states. Our construction combines ideas in Chandra, Kozen and Stockmeyer's construction of a deterministic 1-way finite automaton which simulates an alternating 1-way finite automaton [2] and Shepherdson's construction of a deterministic 1-way finite automaton which simulates a deterministic 2-way finite automaton [18]. Our simulation will be used in § 5 where we show that an alternating auxiliary stack machine with $s(n)$ auxiliary storage can be simulated by an alternating auxiliary PDA with $2^{cs(n)}$ auxiliary storage for some $c>0$.

Formally an *alternating 2-way finite automaton* (Alt-2-FA) is an object of the form

$$M=(Q, q_0, U, F, \Sigma, \delta)$$

where

Q is a finite set of states,
$q_0 \in Q$ is the start state,
$U \subseteq Q$ is the set of universal states,
$F \subseteq Q$ is the set of accepting states,
Σ is the input alphabet (¢, $\$ \notin \Sigma$ serve as left and right endmarkers),
δ is the transition function where

$$\delta: Q \times (\Sigma \cup \{¢, \$\}) \to \mathcal{P}(Q \times \{L, R, S\}).$$

The input is delimited on the left by ¢ and on the right by \$. For convenience we assume that the machine starts in the start state while reading the right endmarker. The machine accepts only if it has just read the right endmarker and moved right into an accepting state. Further, the machine moves right or left deterministically, that is, if $(p, d) \in \delta(q, a)$ and $d \in \{R, L\}$ then $\delta(q, a)=\{(p, d)\}$. Hence the automaton can only do universal or existential branching while the reading head is stationary. It is useful to give a precise definition of acceptance by alternating 2-way finite automata using the terminology of § 2. An ID has the form (q, z, i) where $q \in Q$, $z \in ¢\Sigma^*\{\lambda, \$\}$ and $1 \leqq i \leqq |z|+1$. A universal ID has the form (q, z, i) where $q \in U$. An accepting ID has the form $(q, z, |z|+1)$ where $q \in F$. The initialization function INIT mapping Σ^*

into the set of ID's is defined by

$$\text{INIT}(x) = (q_0, ¢x\$, |x|+2).$$

Let $z = a_1 a_2 \cdots a_n$ where $a_i \in \Sigma \cup \{¢, \$\}$. The transition relation is defined by

$$(q, z, i) \Rightarrow (p, z, i) \quad \text{if } (p, S) \in \delta(q, a_i) \text{ and } 1 \leqq i \leqq n,$$
$$(q, z, i) \Rightarrow (p, z, i+1) \quad \text{if } (p, R) \in \delta(q, a_i) \text{ and } 1 \leqq i \leqq n,$$
$$(q, z, i) \Rightarrow (p, z, i-1) \quad \text{if } (p, L) \in \delta(q, a_i) \text{ and } 1 < i \leqq n.$$

Let $M = (Q, q_0, U, F, \Sigma, \delta)$ be an Alt-2-FA. We now begin the description of a deterministic 1-way FA M' which simulates M. For each $q \in Q$ define a new symbol $\bar{q}$ and let $\bar{Q} = \{\bar{q} | q \in Q\}$. If $A \subseteq Q$, then $\bar{A} = \{\bar{r} | r \in A\}$. Define a *term* to be an object of the form $q \to A$ where $q \in Q$ and $A \subseteq Q \cup \bar{Q}$. A term $q \to A$ is *closed* if $A \subseteq \bar{Q}$. A *partial response* is a set of terms while a *response* is a set of closed terms. The states of M' are exactly the responses.

At this point we give a nonconstructive definition of the rest of the components of M'. Later in this section we show how to construct it. For each $z \in ¢\Sigma^*\{\lambda, \$\}$ we define a response $\mathcal{R}(z)$. A closed term $q \to \bar{A}$ is in $\mathcal{R}(z)$ if $q \in Q$, $A \subseteq Q$ and there is a computation tree of M whose root is labeled with $(q, z, |z|)$ and each leaf is labeled with $(p, z, |z|+1)$ for some $p \in A$. In other words there is a computation tree of M starting in state q and reading the rightmost symbol of z such that each branch ends in a state of A while moving off the right end of z.

LEMMA 4.1. *If* $\mathcal{R}(w) = \mathcal{R}(z)$ *then* $\mathcal{R}(wa) = \mathcal{R}(za)$ *for* $a \in \Sigma \cup \{¢, \$\}$.

Before proving Lemma 4.1 we complete the definition of M'. The transition function δ' is defined by

$$\delta'(\mathcal{R}, a) = \begin{cases} \mathcal{R}(za) & \text{if } \mathcal{R} = \mathcal{R}(z), \\ \varnothing & \text{if } \mathcal{R} \neq \mathcal{R}(z) \text{ for any } z. \end{cases}$$

δ' is well defined by Lemma 4.1. The start state of M' is $\mathcal{R}(¢)$, and $\mathcal{R}$ is an accepting state of M' if $q_0 \to \bar{F} \in \delta'(\mathcal{R}, \$)$. Now x is accepted by M if and only if $q_0 \to \bar{F} \in \mathcal{R}(¢x\$)$ if and only if $\delta'(\mathcal{R}(¢), x)$ is an accepting state of M'. Hence M and M' accept the same language.

Proof of Lemma 4.1. In order to prove this it is useful to associate with each $z \in ¢\Sigma^*\{\lambda, \$\}$ a partial response $\mathcal{PR}(z)$. A term $q \to A$ is in $\mathcal{PR}(z)$ if $q \in Q$, $A \subseteq Q \cup \bar{Q}$ and there is a computation tree T of M whose root is labeled with $(q, z, |z|)$ and each leaf is labeled with either $(p, z, |z|)$ for some $p \in A$ or $(p, z, |z|+1)$ for some $\bar{p} \in A$. We say that T *witnesses* $q \to A \in \mathcal{PR}(z)$. In words, $q \to A \in \mathcal{PR}(z)$ if there is a computation tree of M starting in state q and reading the rightmost symbol of z such that each branch ends in a state $p \in A$ while reading the rightmost symbol of z again or ends in a state p where $\bar{p} \in A$ while moving off the right end of z. The response $\mathcal{R}(z)$ is exactly the set of closed terms in $\mathcal{PR}(z)$. Since $\mathcal{PR}(wa) = \mathcal{PR}(za)$ implies $\mathcal{R}(wa) = \mathcal{R}(za)$ then we can prove the lemma by showing that $\mathcal{R}(w) = \mathcal{R}(z)$ implies $\mathcal{PR}(wa) = \mathcal{PR}(za)$.

We now introduce a constructive method of producing $\mathcal{PR}(wa)$ from $\mathcal{R}(w)$ and a. We consider a proof system similar to the one in the proof of Theorem 3.1. Let $\mathcal{R}$ be a response and let $a \in \Sigma \cup \{¢, \$\}$.

Proof system for $(\mathcal{R}, a)$.

1. $\dfrac{}{q \to \{q\}}$

2. (a) $\dfrac{q \to B, B \subseteq C}{q \to C}$

   (b) $\dfrac{q \to B \cup \{p\}, p \to C}{q \to B \cup C}$

3. $\dfrac{\delta(q, a) = \{(p_1, S), \cdots, (p_k, S)\}, q \text{ universal}}{q \to \{p_1, \cdots, p_k\}}$

4. $\dfrac{(p, S) \in \delta(q, a), q \text{ existential}}{q \to \{p\}}$

5. $\dfrac{\delta(q, a) = \{(p, R)\}}{q \to \{\bar{p}\}}$

6. $\dfrac{\delta(q, a) = \{(p, L)\}, p \to \bar{A} \in \mathcal{R}, A \subseteq Q}{q \to A}$.

Let TH $(\mathcal{R}, a)$ be the set of terms "provable" using the proof system for $(\mathcal{R}, a)$.

CLAIM. $\mathcal{PR}(wa) = \text{TH}(\mathcal{R}(w), a)$.

We prove that $\mathcal{PR}(wa) \subseteq \text{TH}(\mathcal{R}(w), a)$ inductively on the size of witnesses for terms in $\mathcal{PR}(wa)$. Suppose $q \to A \in \mathcal{PR}(wa)$ is witnessed by a single node tree with the label $(q, wa, |wa|)$. Clearly $q \in A$ so that $q \to A$ is provable using rules 1 and 2(a). Suppose now that $q \to A \in \mathcal{PR}(wa)$ is witnessed by a tree T with more than one node. There are four cases to consider.

(i) $\delta(q, a) = \{(p_1, S), \cdots, (p_k, S)\}$ and q is universal. In this case the root of T is labeled with $(q, wa, |wa|)$ with k immediate children $\pi_1, \cdots, \pi_k$ labeled respectively with $(p_1, wa, |wa|), \cdots, (p_k, wa, |wa|)$. The complete subtree rooted at π_i witnesses $p_i \to A \in \mathcal{PR}(wa)$. By the induction hypothesis $p_i \to A \in \text{TH}(\mathcal{R}(w), a)$. By 3 and 2(b) we obtain $q \to A \in \text{TH}(\mathcal{R}(w), a)$.

(ii) $\delta(q, a) = \{(p_1, S), \cdots, (p_k, S)\}$ and q is existential. This case is just like (i) except we use 4 and 2(b).

(iii) $\delta(q, a) = \{(p, R)\}$. We must have $\bar{p} \in A$ for T to witness $q \to A \in \mathcal{PR}(wa)$. Hence by rules 5 and 2(a) we have $q \to A \in \text{TH}(\mathcal{R}(w), a)$.

(iv) $\delta(q, a) = \{(p, L)\}$. The child π of the root of T has the label $(p, wa, |wa| - 1)$. Every path from π to a leaf of T must pass through a node with label of the form $(r, wa, |wa|)$. That is, the computation must return to reading the rightmost symbol of wa again. Let $P = \{\rho_1, \cdots, \rho_k\}$ be the descendants of π with the properties (a) ρ_i is labeled $(r_i, wa, |wa|)$, (b) no node between π and ρ_i has a label with third coordinate $|wa|$ and (c) every path from π to a leaf passes through a node of P. Let T' be the unique subtree of T whose root is π and whose set of leaves is P. If we change the second coordinate of every node label of T' from wa to w then T' witnesses $p \to \{\bar{r}_1, \cdots, \bar{r}_k\} \in \mathcal{PR}(w)$. Since this is a closed term then $p \to \{\bar{r}_1, \cdots, \bar{r}_k\} \in \mathcal{R}(w)$. By 6, $q \to \{r_1, \cdots, r_k\} \in \text{TH}(\mathcal{R}(w), a)$. The complete subtree of T rooted at ρ_i witnesses $r_i \to A \in \mathcal{PR}(wa)$. So by the induction hypothesis $r_i \to A$ is provable for all i. Hence by 2(b), $q \to A \in \text{TH}(\mathcal{R}(w), a)$.

We prove that $\text{TH}(\mathcal{R}(w), a) \subseteq \mathcal{PR}(wa)$ by induction on proof length. Suppose $q \to A \in \text{TH}(\mathcal{R}(w), a)$ has a proof of length l. If the last step of the proof (the step

from which $q \to A$ is concluded) is an application of rule 1, 3, 4, or 5, then it is immediate that there is a computation tree which witnesses $q \to A \in \mathcal{PR}(wa)$. If the last step is an application of rule 2(a) then the same computation tree that witnesses the antecedent term also witnesses the conclusion term. If the last step is an application of rule 2(b) then suppose that $q \to B \cup \{p\}$ and $p \to C$ are the antecedents from which $q \to B \cup C$ is concluded ($A = B \cup C$). By the induction hypothesis there are computation trees T and T' which witness $q \to B \cup \{p\}$ and $p \to C$ respectively. If each leaf of T labeled $(p, wa, |wa|)$ is replaced with the tree T' (whose root is labeled $(p, wa, |wa|)$ then the resulting tree witnesses $q \to B \cup C \in \mathcal{PR}(wa)$.

If the last step is an application of rule 6 then suppose $q \to A$ is concluded from $\delta(q, a) = \{(p, L)\}, p \to \bar{A} \in \mathcal{R}(w)$ and $A \subseteq Q$. Since $p \to \bar{A} \in \mathcal{R}(w)$, then there is a computation tree T' with root labeled $(p, w, |w|)$ and each leaf labeled $(r, w, |w|+1)$ for some $r \in A$. First modify T' so that the second coordinate of each node label is wa instead of w. Next make the root of the modified T' the child of a node labeled $(q, wa, |wa|)$. The resulting tree witnesses $q \to A$.

This concludes the proof of the claim. We are now ready to complete the proof of Lemma 4.1. Assume $\mathcal{R}(w) = \mathcal{R}(z)$. By the claim, TH $(\mathcal{R}(w), a) = \mathcal{PR}(wa)$ and TH $(\mathcal{R}(z), a) = \mathcal{PR}(za)$. Thus $\mathcal{PR}(wa) = \mathcal{PR}(za)$. □

We now know how to construct the transition function δ' of M'. If $\mathcal{R}$ is a response and $a \in \Sigma \cup \{¢, \$\}$ then first using the proof system for $(\mathcal{R}, a)$ construct the set TH $(\mathcal{R}, a)$. Let CTH $(\mathcal{R}, a)$ be the set of closed terms in TH $(\mathcal{R}, a)$. Now $\delta'(\mathcal{R}, a) = \text{CTH}(\mathcal{R}, a)$.

We summarize the main results of this section in the following theorem. Parts (2) and (3) are used in the next section.

THEOREM 4.2. (1) *If M is an m-state* Alt-2-FA, *then $L(M)$ is accepted by a 2^{m2^m}-state deterministic* 1-*way finite automaton.*

(2) *There is a deterministic Turing machine which, when given the description of an m-state* Alt-2-FA, *responses $\mathcal{R}$ and $\mathcal{R}'$, and an input symbol a, checks whether $\mathcal{R}' = \text{CTH}(\mathcal{R}, a)$ within time 2^{dm} for some constant d. A deterministic Turing machine can also check, within time 2^{dm}, whether $\mathcal{R} = \mathcal{R}(¢)$.*

(3) *Any response of an m-state* Alt-2-FA *can be written in space 2^{cm} for some constant c.*

Proof. (1) There are at most $m2^m$ closed terms. Since a response is a set of closed terms, there are at most 2^{m2^m} responses.

(2) The deterministic Turing machine generates the terms in TH $(\mathcal{R}, a)$ by applying the rules in the proof system for $(\mathcal{R}, a)$ until no more terms can be generated. The machine then checks that $\mathcal{R}'$ is the set of closed terms in TH $(\mathcal{R}, a)$. As in the previous section, the time is polynomial in the number of terms. To check that $\mathcal{R} = \mathcal{R}(¢)$, note that $\mathcal{R}(¢)$ is generated by a proof system similar to the one above, except that there is no rule 6.

(3) This is obvious. □

5. Alternating stack automata. An (alternating) stack automaton is just like an (alternating) pushdown automaton except that the interior contents of the pushdown store may be read, but not changed except by normal pushing or popping. A stack can also be viewed as a Turing machine tape, 1-way infinite to the right, with the restriction that symbols can be changed only on the right end of the nonblank tape contents. In the special case of a nonerasing stack automaton, the stack cannot be popped, or, equivalently, stack symbols cannot be erased. As in [12], we consider stack automata with a space bounded auxiliary storage tape. Formally, an *alternating*

auxiliary stack automaton (Alt-Aux-SA) is of the form

$$M = (Q, S, q_0, U, F, \Sigma, \Gamma, \Delta, ¢, \delta)$$

where Q (the states), q_0 (the start state), U (the universal states), F (the accepting states), Σ (the input alphabet), Γ (the auxiliary storage alphabet), Δ (the stack alphabet), and ¢ (the bottom, or leftmost, stack symbol) are as in the definition of Alt-Aux-PDA's. In addition, $S \subseteq Q$ and the states in S are called *scan states*. We also define $P = Q - S$ and refer to states in P as *pushdown states*. The blank symbol # belongs to Δ as well as to Γ. The transition function is of the form

$$\delta: Q \times (\Sigma \cup \{¢, \$\}) \times \Gamma \times \Delta$$
$$\to \mathscr{P}(Q \times \{L, R, S\}^2 \times (\Gamma - \{\#\}) \times ((\Delta - \{\#, ¢\}) \cup \{\text{POP}, \text{IDLE}, L, R\})),$$

where R (L) in the last component signifies a right (left) shift of the stack head.

When the state of the machine is in S (P) the machine is said to be in *scan mode* (*pushdown mode*). Initially, the machine is in pushdown mode (i.e., $q_0 \in P$), the stack contains $¢\#\#\# \cdots$ and the stack head is scanning ¢. Generally, when in pushdown mode the stack contains $\beta\#\#\# \cdots$ for some $\beta \in ¢ \cdot (\Delta - \{\#, ¢\})^*$ and the stack head is scanning the rightmost symbol of β; the machine behaves just as an Alt-Aux-PDA, manipulating the stack by pushing, popping, or idling. At some point, the machine can enter scan mode without moving the stack head. While in scan mode the machine can read stack symbols and move the stack head left and right (and idle), but it cannot push or pop. When in scan mode the machine behaves much like an alternating Turing machine with an auxiliary storage tape, and with the stack providing a read-only "input" (in addition to the original read-only input in Σ^*). The machine must remain in scan mode until it first reads a blank (which must be the blank just to the right of β). Then it must shift the stack head left and enter pushdown mode. Furthermore, we require that $F \subseteq P$, but we now allow M to accept when reading stack symbols other than ¢. We assume that the machine behaves deterministically when either pushing, popping, moving the stack head left or right, or changing from pushdown mode to scan mode or vice versa. Some of these conventions for stack automata differ from those in [10], [12], but they are convenient for our proofs. An *alternating auxiliary nonerasing stack automaton* (Alt-Aux-NESA) is an Alt-Aux-SA which cannot pop the stack.

An ID of an Alt-Aux-SA has the form $(q, x, i, \alpha, j, \beta, k)$ where q, x, i, α, j, and β have the same meaning as for ID's of Alt-Aux-PDA's, and k, $1 \leqq k \leqq |\beta| + 1$, indicates the position of the stack head. The stack positions are numbered from left to right so, for example, $k = 1$ if M is reading the leftmost symbol ¢, or $k = |\beta|$ if M is reading the top nonblank stack symbol. Now

$$\text{INIT}(x) = (q_0, x, 0, \lambda, 0, ¢, 1),$$

the accepting ID's have their first coordinate in F, and the universal ID's have their first coordinate in U. The definition of the transition relation should be clear from the discussion above. Let

$$\text{SPACE}((q, x, i, \alpha, j, \beta, k)) = |\alpha|.$$

Define

$$\text{ALT-AUX-SA}(s(n)) = \{L(M) | M \text{ is an Alt-Aux-SA which is } s(n)\text{-space bounded}\},$$

$$\text{ALT-AUX-NESA}(s(n)) = \{L(M) | M \text{ is an Alt-Aux-NESA which is } s(n)\text{-space bounded}\}.$$

One consequence of our characterization of space bounded alternating auxiliary stack automata is that the ability to erase the stack is inessential.

THEOREM 5.1. *Let* $s(n) \geqq \log n$.

$$\text{ALT-AUX-SA}(s(n)) = \text{ALT-AUX-NESA}(s(n)) = \bigcup \text{DTIME}(E(3, cs(n))).$$

Since obviously ALT-AUX-NESA $(s(n)) \subseteq$ ALT-AUX-SA $(s(n))$, Theorem 5.1 follows immediately from Lemmas 5.2 and 5.3.

LEMMA 5.2. *If* $s(n) \geqq \log n$, *then*

$$\bigcup \text{DTIME}(E(3, cs(n))) \subseteq \text{ALT-AUX-NESA}(s(n)).$$

Proof. By Theorem 2.1, it is sufficient to prove that

$$\bigcup \text{ASPACE}(E(2, cs(n))) \subseteq \text{ALT-AUX-NESA}(s(n)).$$

The proof is very similar to the first part of the proof of Theorem 3.1. The only difference is that now the ID's of the alternating Turing machine are words of length $E(2, cs(n))$. The extra exponential is handled by preceding each symbol of an ID by a binary address; for each ID, the addresses run consecutively from 0 to $E(2, cs(n)) - 1$. Since the length of an address is only $2^{cs(n)}$, $s(n)$ storage is sufficient to record a pointer to a particular bit-position within an address. Therefore an $s(n)$-space bounded Alt-Aux-NESA can check that two physically consecutive addresses are numerically consecutive (in fact, this can be done deterministically), and it can check that the address of a symbol deep in the stack which is being scanned in scan mode matches the address of the symbol on the top of the stack (universal branching is used here). This ability to match addresses is used to implement a procedure similar to CHECK in the proof of Theorem 3.1. More precisely, the procedures NEW and CHECK are replaced by:

NEW:
- If the topmost address is all 1's, then call TOP;
- Existentially guess a binary word α with $|\alpha| = 2^{cs(n)}$ and push it onto the stack;
- Existentially push a symbol $\gamma \in \Gamma \cup Q \times \Gamma$;
- (call ADDCHECK $\wedge$ call CHECK $\wedge$ call NEW).

ADDCHECK:
- Let α be the address on the top of the stack;
- Let α' be the address just below α on the stack;
- If $\alpha = \alpha' + 1 \bmod E(2, cs(n))$ then accept; else reject.

CHECK:
- Remember the top pushdown symbol γ;
- In scan mode, existentially choose a symbol γ_2 in the ID just below the ID currently being guessed on the top of the stack, let γ_3 and γ_1 be the ID symbols just above and below γ_2, and let α_2 be the address of γ_2;
- If $\gamma \neq f_{M,m}(\gamma_1, \gamma_2, \gamma_3)$ then reject;
- Universally choose a j with $1 \leqq j \leqq 2^{cs(n)}$;
- Let b be the jth bit of α_2;
- Return to the top address α and check whether or not b equals the jth bit of α, and accept or reject accordingly.

INIT must also be changed in the obvious way to incorporate the addresses. □

LEMMA 5.3. *If* $s(n) \geqq \log n$, *then*

$$\text{ALT-AUX-SA}(s(n)) \subseteq \bigcup \text{DTIME}(E(3, cs(n))).$$

Proof. By Theorem 3.1, it is sufficient to prove that

$$\text{ALT-AUX-SA}(s(n)) \subseteq \bigcup \text{ALT-AUX-PDA}(2^{cs(n)}).$$

Let

$$M = (Q, S, q_0, U, F, \Sigma, \Gamma, \Delta, ¢, \delta)$$

be an Alt-Aux-SA which is $s(n)$-space bounded. A $2^{cs(n)}$-space bounded Alt-Aux-PDA M' will perform a step-by-step simulation of M when M is in pushdown mode. During this simulation, M' will maintain on its pushdown store the response of the stack contents of M (see § 4); this will allow M' to simulate M when M is in scan mode.

Fix an input x of length n. When in scan mode, M can be viewed as an alternating 2-way finite automaton $\mathscr{A}_M$ with about $2^{es(n)}$ states for some constant $e>0$. The states of $\mathscr{A}_M$ are of the form (q, i, α, j) where $q \in S$, i indicates M's input head position $(0 \leqq i \leqq n+1)$, $\alpha \in (\Gamma - \{\#\})^*$ with $|\alpha| \leqq s(n)$ indicates the contents of M's auxiliary tape, and j indicates the auxiliary tape head position. The universal states of $\mathscr{A}_M$ are those of the form (q, i, α, j) where $q \in U$. The input alphabet of $\mathscr{A}_M$ is Δ, the stack alphabet of M. For our purposes here, we need not specify an initial state or accepting states for $\mathscr{A}_M$. It should be obvious how the transition function of $\mathscr{A}_M$ is obtained from that of M; note that the ID

$$((q, i, \alpha, j), \beta, k) \text{ of } \mathscr{A}_M$$

corresponds to the ID

$$(q, x, i, \alpha, j, \beta, k) \text{ of } M.$$

To describe the simulation, suppose that M is in some ID

$$r_0 = (q, x, i, \alpha, j, \beta, |\beta|).$$

M' will maintain q, i, α, and j on its auxiliary storage tape. Furthermore, the pushdown store of M' will contain

$$\mathscr{R}(\rho_1)\beta_1\mathscr{R}(\rho_2)\beta_2 \cdots \mathscr{R}(\rho_z)\beta_z$$

where $\beta = \beta_1\beta_2 \cdots \beta_z$, $\beta_i \in \Delta$ for all i, ρ_i is the length i prefix of β (in particular, $\rho_1 = ¢$ and $\rho_z = \beta$), and the response $\mathscr{R}$ is with respect to $\mathscr{A}_M$. The concept of a *response* is defined and discussed in § 4. To recapitulate briefly in the context of this proof, if $q \in S$ and $(q, i, \alpha, j) \to \bar{A}$ is a term in $\mathscr{R}(\beta)$ where A is a set of states of $\mathscr{A}_M$, then there is a computation tree of M with root r_0 such that if $(q', x, i', \alpha', j', \beta', k)$ is a leaf of the computation, then $(q', i', \alpha', j') \in A$, $\beta' = \beta$, and $k = |\beta|+1$; in particular, at each leaf the stack scan has just finished and M must reenter pushdown mode.

We now describe how M' simulates M in an ID r_0 in various cases. If $q \in F$ then M' accepts. If r_0 has more than one successor, then M' simulates this directly by using its own alternation (recall that the stack head cannot move in this case). If M pops the stack, then M' pops $\mathscr{R}(\beta)\beta_z$ off the pushdown store (while updating q, i, α, j as necessary).

If M pushes a symbol $a \in \Delta$ onto the stack, then M' existentially pushes some string of symbols in the alphabet used to encode responses, and then pushes a. M' then enters a universal state to choose one of two further actions. One action is to continue the simulation as though $\mathscr{R}(\beta a)$ was guessed correctly. The other action is

to verify that the guess really was correct. We must argue that this can be done using $2^{cs(n)}$ space. By Theorem 4.2(2), the set

$$\text{UPDATE} = \{(\mathcal{R}, \mathcal{R}', a) | \mathcal{R}' = \text{CTH}(\mathcal{R}, a)\}$$

can be accepted by a deterministic Turing machine within time 2^{dm} where $m \leqq 2^{cs(n)}$ is the number of states of $\mathcal{A}_M$; this is done by applying the proof system for $(\mathcal{R}, a)$. Recall that CTH is the transition function of a deterministic 1-way finite automaton equivalent to $\mathcal{A}_M$. So by Theorem 2.1 and Remark 2.2, UPDATE is accepted by an alternating Turing machine M'' which is $2^{cs(n)}$-space bounded; moreover M'' has a 1-way input head which we can assume starts on the right end of the input and moves left. (We are using here the fact that the space $2^{cs(n)}$ is at least logarithmic in the length of the "input" $(\mathcal{R}, \mathcal{R}', a)$. This is true because, as noted in Theorem 4.2(3), a response can be written in space $2^{c'm}$ for some constant c'.) So M' can simulate M'' where the pushdown head of M' plays the role of the left-moving input head of M''. (By a similar argument, M' can guess $\mathcal{R}(¢)$ and check the correctness of the guess at the start of the computation.)

If M has entered scan mode (i.e., if $q \in S$ in r_0), then M' existentially guesses a term

$$(q, i, \alpha, j) \rightarrow \bar{A}$$

where A is a set of states of $\mathcal{A}_M$ and writes it on the auxiliary storage tape. M' then enters a universal state to choose one of two further actions. One action is to check that $(q, i, \alpha, j) \rightarrow \bar{A} \in \mathcal{R}(\beta)$; this is done in the obvious way by popping the pushdown store. The other action is to universally choose some state $(q', i', \alpha', j') \in A$ and continue the simulation as though M were in the ID $(q', x, i', \alpha', j', \beta, |\beta|+1)$. By convention, from this ID M must move the stack head left and reenter pushdown mode, so M is again in an ID of the form r_0. □

As discussed at the end of § 3, we have actually proved the following result for general $s(n)$.

THEOREM 5.4. *Let* $s(n) \geqq 1$.

$$\text{ALT-AUX-SA}(s(n)) = \text{ALT-AUX-NESA}(s(n)) = \bigcup \text{DTIME}(E(2, n2^{cs(n)})).$$

Let ALT-SA and ALT-NESA be the classes of languages accepted by alternating stack automata and alternating nonerasing stack automata, respectively, without an auxiliary worktape.

COROLLARY 5.5. $\text{ALT-SA} = \text{ALT-NESA} = \bigcup \text{DTIME}(2^{2^{cn}})$.

REFERENCES

[1] L. BERMAN, *The complexity of logical theories*, Theoret. Comput. Sci., 11 (1980), pp. 71–77.

[2] A. K. CHANDRA, D. C. KOZEN AND L. J. STOCKMEYER, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114–133.

[3] S. A. COOK, *Characterizations of pushdown machines in terms of time-bounded computers*, J. Assoc. Comput. Mach., 18 (1971), pp. 4–18.

[4] M. J. FISCHER AND R. E. LADNER, *Propositional dynamic logic of regular programs*, J. Comput. Sys. Sci., 18 (1979), pp. 194–211.

[5] A. S. FRAENKEL AND D. LICHTENSTEIN, *Computing a perfect strategy for n × n chess requires time exponential in n*, J. Combin. Theory A, 31 (1981), pp. 199–214.

[6] S. GINSBURG, S. A. GREIBACH AND M. A. HARRISON, *Stack automata and compiling*, J. Assoc. Comput. Mach., 14 (1967), pp. 172–201.

[7] J. GRAY, M. A. HARRISON AND O. H. IBARRA, *Two-way pushdown automata*, Inform. Control, 11 (1967), pp. 30–70.

[8] E. M. GURARI AND O. H. IBARRA, *Path systems: constructions, solutions and applications*, this Journal, 9 (1980), pp. 348–374.

[9] ———, *(Semi)alternating stack automata*, Math. Systems Theory, 15 (1982), pp. 211–224.

[10] J. E. HOPCROFT AND J. D. ULLMAN, *Nonerasing stack automata*, J. Comput. Sys. Sci., 1 (1967), pp. 166–186.

[11] ———, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA, 1979.

[12] O. H. IBARRA, *Characterizations of some tape and time complexity classes of Turing machines in terms of multihead and auxiliary stack automata*, J. Comput. Sys. Sci., 5 (1971), pp. 88–117.

[13] K. N. KING, *Measures of parallelism in alternating computation trees*, Proc. 13th Annual ACM Symposium on the Theory of Computing, 1981, pp. 189–201.

[14] D. KOZEN, *Complexity of Boolean algebras*, Theoret. Comput. Sci., 10 (1980), pp. 221–247.

[15] R. E. LADNER, *The complexity of problems in systems of communicating sequential processes*, J. Comput. Sys. Sci., 21 (1980), pp. 179–194.

[16] H. R. LEWIS, *Complexity results for classes of quantificational formulas*, J. Comput. Sys. Sci., 21 (1980), pp. 317–353.

[17] W. L. RUZZO, *Tree-size bounded alternation*, J. Comput. Sys. Sci., 21 (1980), pp. 218–235.

[18] J. C. SHEPHERDSON, *The reduction of two-way automata to one-way automata*, IBM J. Res., 3 (1959), pp. 198–200.

[19] L. J. STOCKMEYER AND A. K. CHANDRA, *Provably difficult combinatorial games*, this Journal, 8 (1979), pp. 151–174.

SIAM J. COMPUT.
Vol. 13, No. 1, February 1984

© 1984 Society for Industrial and Applied Mathematics
0097-5397/84/1301-0011 $01.25/0

LIMITATIONS ON EXPLICIT CONSTRUCTIONS OF EXPANDING GRAPHS*

MARIA KLAWE†

Abstract. Expanding graphs are the basic building blocks in constructions of many types of graphs with special connectivity properties which arise in a variety of applications including switching networks, sorting networks and establishing time-space trade-offs for numerous computational problems. Only one explicit method of constructing arbitrarily large expanding graphs with a linear number of edges is known (Margulis [13], Gabber and Galil [8]), but the number of edges used is much greater than the number known to be sufficient via probabilistic arguments. In this paper we show that various other constructions which have been proposed to obtain expanding graphs, including one-dimensional analogues of the Gabber–Galil construction and some pseudorandom constructions, cannot ever yield expanding graphs.

Key words. network, expander, superconcentrator

1. Introduction. For any bipartite graph, whose two vertex sets are called inputs and outputs, if X is a subset of inputs we will use ΓX to denote the neighborhood of X, i.e. the set of outputs which are adjacent to some input in X. Moreover, we will denote the cardinality of any set A by $|A|$. For the purposes of this paper, we will call a bipartite graph with n inputs and n outputs an expanding graph, if there exist positive constants α and δ, such that for any subset X of inputs with $|X| \leqq \alpha n$ we have $|\Gamma X| \geqq (1+\delta)|X|$. (There are many slight variations in the definitions of expanding graphs in the applications we will mention, but the basic idea is always that every set in some class of subsets of inputs is guaranteed to expand by some fixed amount.)

It is obvious that expanding graphs exist since the complete bipartite graph is an expanding graph for any α and δ with $(1+\delta)\alpha \leqq 1$. What is more surprising is that there are families of expanding graphs with only a linear number of edges. In fact, for any α and δ such that $(1+\delta)\alpha < 1$, there is some constant k such that for every n there is a bipartite graph with n inputs, n outputs and at most kn edges, which is an expanding graph with respect to α and δ. Pinsker [19] gave a fairly simple probabilistic proof of this for a particular α, δ and k in 1973; similar arguments have been used in subsequent papers to prove this fact for other combinations of α and δ, and it not hard to see that these probabilistic arguments succeed in general.

Over the past ten years expanding graphs with a linear number of edges have been used as building blocks in constructions of graphs appearing in a broad spectrum of applications. As motivation for the importance of obtaining good explicit constructions, and consequently for the significance of the results in this paper, we give a brief survey of these applications.

The study of the complexity of graphs with special connectivity properties originated in switching theory, motivated by problems of designing networks able to connect many disjoint sets of users, while only using a small number of switches. An example of this type of graph is a superconcentrator, which is an acyclic directed graph with n inputs and n outputs such that given any pair of subsets A and B of the same size, of inputs and outputs respectively, there exists a set of disjoint paths joining the inputs in A to the outputs in B. Some other examples are concentrators, nonblocking connectors and generalized connectors (see [6], [21] for more details). There is a large body of work searching for optimal constructions of these graphs (Pinsker [19],

* Received by the editors July 14, 1982, and in revised form February 8, 1983.
† Computer Science Department, IBM Research, San Jose, California 95193.

Bassalygo and Pinsker [3], Cantor [5], Ofman [15], Masson and Jordan [14], Pippenger [20], [21], Chung [6]). So far all optimal explicit constructions depend on expanding graphs of some sort.

Superconcentrators have also proved to be useful in theoretical computer science. By showing that the computation graphs of straight line programs for problems such as polynomial multiplication, the Fourier transform and matrix inversion must be superconcentrators, it has been possible to establish nonlinear lower time bounds and time-space trade-offs for these problems assuming certain models of computation (Valiant [25], Abelson [1], Ja'Ja' [9], Tompa [24]).

These space-time trade-offs are obtained via a game known as pebbling which is played on acyclic directed graphs and mimics the storage of temporary results during a straight-line computation. In considering the problem of pebbling an arbitrary acyclic directed graph, expanding graphs have been used in several instances to construct graphs which are (in some sense) hardest to pebble, hence establishing lower bounds in space-time trade-offs (Lengauer and Tarjan [12], Paul and Tarjan [17], Paul, Tarjan and Celoni [18], Pippenger [22]).

Expanding graphs have also been used to construct sparse graphs with dense long paths (Erdos, Graham and Szemeredi [7]). Interest in sparse graphs with dense long paths stems from studying the complexity of Boolean functions, and more recently from problems of designing fault-tolerant microelectronic chips. Paul and Reischuk strengthened this result by constructing (still using expanding graphs) sparse graphs of bounded in-degree with dense long paths, which is of interest since computation graphs have bounded in-degree.

Perhaps the most practical applications of expanding graphs occur in the two most recent results. Ajtai, Komlos and Szemeredi [2] have announced the construction of an oblivious sorting network using $O(n \log n)$ comparators, and having depth $O(\log n)$. Again, expanding graphs form the basic components, and of course, the explicit construction of the sorting network depends on the explicit construction of expanding graphs. The problem of constructing such a sorting network has been open for twenty years [4], which perhaps illustrates best the unexpected power of expanding graphs. Finally, expanding graphs have been used by Karp and Pippenger [10] to design an algorithm which can be applied to virtually all the well-known Monte-Carlo algorithms to reduce the number of uses of a randomization resource (i.e. coin-flips or calls to a random number generator) while still maintaining polynomial running time.

In several of the applications mentioned above the usefulness of expanding graphs depends on the existence of an explicit construction of expanding graphs with a linear number of edges. In 1973 Margulis [13] gave an explicit construction, but, although he was able to prove that the constant δ was greater than zero, he was not able to bound δ strictly away from zero. In 1979, after slightly modifying Margulis's construction, Gabber and Galil [8] were able to obtain a positive lower bound for δ, and thus obtained the first usable explicit construction, which we now present for future reference. Let Z_m denote the integers mod m, and let f_i for $i=0, 1, \cdots, 6$ be the functions on $Z_m \times Z_m$ defined by

$$f_0(s, t) = (s, t),$$

$$f_1(s, t) = (s, 2s+t) \bmod m,$$

$$f_2(s, t) = (s, 2s+t+1) \bmod m,$$

$$f_3(s, t) = (s, 2s+t+2) \bmod m,$$

$$f_4(s, t) = (s + 2t, t) \bmod m,$$

$$f_5(s, t) = (s + 2t + 1, t) \bmod m,$$

$$f_6(s, t) = (s + 2t + 2, t) \bmod m.$$

The graph $G(m)$ is defined as the bipartite graph with inputs $\{x(s, t): 1 \leqq s, t \leqq m\}$ and outputs $\{y(s, t): 1 \leqq s, t \leqq m\}$ such that $x(s, t)$ is adjacent to $y(f_i(s, t))$ for $i = 0, 1, \cdots, 6$.

There are two aspects of this construction which make it less than completely satisfactory. The first, and most important, is that the combination of α and δ for which Gabber and Galil are able to prove that $G(m)$ expands is significantly worse than those combinations which can be proved to exist by probabilistic methods. As a result, for example, the best construction of superconcentrators using their expanding graphs has $261.5n$ edges, which compares unfavorably with the fact that it is known (via probabilistic methods) that superconcentrators exist with $(38.5n + O(\log n))$ edges (Chung [6]). The second is that the proof that their construction succeeds is fairly sophisticated mathematically. One might hope for a more elementary and intuitively satisfying proof. Consequently the search has continued for explicit constructions of expanding graphs with a linear number of edges.

The most obvious approach is to look for some variant of the Gabber–Galil construction which would either yield a better combination of α, δ and k, or at least yield a simpler proof of expansion. Another possibility which has occurred to many people, is that since it can be shown that for any α and δ there exists k such that almost all random bipartite graphs with kn edges expand with respect to α and δ, one could use pseudorandom number generators to construct a bipartite graph with kn edges. Then, presumably with high probability, this graph should be an expanding graph with respect to α and δ. We will refer to this type of construction as a pseudorandom construction. Yet another direction has been proposed by Tanner [23]. He observed that if λ_1 and λ_2 are the two largest eigenvalues of MM^T, where M is the incidence matrix of a regular bipartite graph G, then G is an expanding graph with respect to α and $(\lambda_1/(\alpha\lambda_1 + (1-\alpha)\lambda_2)) - 1$. Thus it suffices to construct regular bipartite graphs with a linear number of edges such that the two largest eigenvalues of MM^T are widely separated. Tanner also showed that a class of graphs known as generalized n-gons have this property, but unfortunately generalized n-gons only exist for finitely many n.

The results in this paper show that at least the most obvious examples of the first two above approaches cannot succeed. We will define a class of constructions which both is a natural variant of the Gabber–Galil construction, and includes all the graphs which can be obtained by pseudorandom constructions when linear congruential pseudorandom number generators are used in the following fashion. Given a finite set $\{f_i\}$ of pseudorandom number generators, the edges of the pseudorandom graph are all pairs of the form $(x, f_i(x))$ where $1 \leqq x \leqq n$.

Notice that each f_i in the Gabber–Galil construction is the restriction mod m of a two-dimensional linear function, all of whose coefficients are either 0, 1 or 2. In an analogous manner, for any finite set $F = \{a_i x + b_i: 1 \leqq i \leqq k\}$ of one-dimensional linear mappings, we can define a bipartite graph $G(n, F)$ with inputs $\{x(i): 1 \leqq i \leqq n\}$ and outputs $\{y(i): 1 \leqq i \leqq n\}$ such that $x(i)$ is adjacent to $y(j)$ if and only if $j = \lfloor f(i) \rfloor \bmod n$ for some f in F. By choosing the coefficients a_i to be integers, it is easy to see that this class includes all graphs which could be obtained by pseudorandom constructions using linear congruential pseudorandom number generators. Suppose $0 < \alpha < 1$ and

let F be a finite family of one-dimensional linear functions with rational coefficients. The main result of this paper is the following.

THEOREM. *There exist functions* $N(\alpha, |F|)$ *and* $\delta(\alpha, F, n)$ *such that the limit of* $\delta(\alpha, F, n)$ *as* n *goes to infinity is* 0, *and such that for each* $n \geqq N(\alpha, |F|)$ *there is a subset* X *of the inputs of* $G(n, F)$ *with* $\alpha n/2 < |X| \leqq \alpha n$ *and* $|\Gamma X| < (1+\delta(\alpha, F, n))|X|$.

Since $\lim_{n\to\infty} \delta(\alpha, F, n) = 0$, there is no $\delta > 0$ such that $G(n, F)$ is an expanding graph with respect to α and δ for all n. Moreover, if the coefficients of the functions in F are integers, we can prove a stronger result. Namely that $\delta(\alpha, F, n)$ depends only on α, $|F|$ and n. This strengthening is particularly important when applying the result to pseudorandom constructions using linear congruential number generators since it means that even if the multipliers are chosen as a function of n, expanding graphs cannot be obtained.

The theorem above is proved by explicitly constructing a nonexpanding subset X, and establishing a number of its properties. In an earlier version of this paper [11], we proved similar results using an entirely different construction of nonexpanding subsets. There are two ways in which this paper's construction improves upon the previous one. First of all, the old construction did not yield the stronger result for integer coefficients. The second improvement is that in the new construction the size of the nonexpanding subset can be specified fairly precisely, whereas previously the size of the nonexpanding subset was $O(n^{2/3})$ and thus could not be applied to situations where one is only interested, for example, in the expansion of sets of approximately half the inputs. We should, however, point out one aspect in which the old construction may dominate the new one. Let F be the family $\{(p_i/q_i)x + b_i : 1 \leqq i \leqq k\}$. The δ function in the old construction is $(\sum_{1\leqq i\leqq k} \log p_i + \log q_i)(\sum_{1\leqq i \leqq k} |p_i/q_i| + |b_i|)/\log n$, whereas the δ function in the new construction (for rationals) is $\sum_{1\leqq i\leqq k} ((3+q_i)/s + (q_i(p_i^2+1))/\tau)$ where

$$s = \lfloor (\log \alpha n/\log\log \alpha n)^{1/(3k+2)} \rfloor \quad \text{and} \quad \tau = \lfloor (s/\alpha)^{(k+1)s^{k+2}} \rfloor.$$

It is not hard to see that for some sets F and choices of α and n the value of the old δ function is much smaller than the value of the new δ function, and hence in those cases the old construction would give a stronger nonexpansion result. The new δ function for integers is $3k/s$, and again in some cases the old δ function is less than this.

There are two major questions about one-dimensional linear constructions which remain unsettled. The first is whether it is possible to obtain expanding graphs using real coefficients, and the second is whether it is possible to extend the stronger integer result to rational coefficients. Of course a positive answer to the second would also imply a negative answer to the first, since for any fixed n and finite set F of one-dimensional linear mappings with real coefficients there is a set F' with rational coefficients such that $G(n, F) = G(n, F')$. However, as n increases so must the numerators and denominators in the rational coefficients in F', and so the kind of result for rational coefficients given in this paper has no implication for real coefficients.

The next section describes the construction of the nonexpanding subset X and establishes sufficiently many of its properties to prove the result for integer coefficients. In § 3, we continue to explore the properties of X, finally achieving the result for rational coefficients. We are also able to apply this construction to shuffle-exchange graphs, thus proving that shuffle-exchange graphs cannot be expanding graphs either.

2. Integer coefficients. Given integers a_i and b_i for $1 \leqq i \leqq k$ for each i let us define a mapping f_i on Z_n by $f_i(x) = a_i x + b_i \bmod n$. For sets A and B we use $A\backslash B$ to denote the difference set of elements which are in A but not B. This section is devoted to proving the following theorem.

THEOREM 2.1. *For each real number α between 0 and 1 there is a constant N depending only on α and k, such that for each $n \geqq N$, there exists a subset X of Z_n with $\alpha n/2 \leqq |X| \leqq \alpha n$, and $|f_i(X)\backslash X| < 3|X|/\lfloor(\log \alpha n/\log\log \alpha n)^{1/(3k+2)}\rfloor$ for $1 \leqq i \leqq k$.*

We begin by introducing some notation and conventions that we will use. For any numbers x and p in Z_n, unless otherwise noted we will understand px and $x+p$ to mean $px \bmod n$ and $(x+p) \bmod n$ respectively. The greatest common divisor of p and x is denoted by (p, x), and if X is a subset of Z_n then $p^{-1}X$ is the subset of Z_n defined by $p^{-1}X = \{z : pz \in X\}$. For subsets X and Y we will use XY to denote the product subset, i.e. $XY = \{z : z = xy$ for some x in X and y in $Y\}$. Similarly $\prod_{1 \leqq i \leqq j} X_i$ denotes the product set $X_1X_2 \cdots X_j$. Finally, let s denote $\lfloor(\log \alpha n/\log\log \alpha n)^{1/(3k+2)}\rfloor$, let τ denote $\lfloor(s/\alpha)^{(k+1)s^{k+2}}\rfloor$, let $\nu = \max\{4k+4, (1/\alpha)^{3k+2}\}$, and let $N = 2^{2^\nu}/\alpha$. For the remainder of this section we will assume that n and t are integers satisfying $n \geqq N$ and $\tau \leqq t \leqq n$. The next lemma states the inequalities involving these numbers which we will require in the remainder of this section.

LEMMA 2.2.
(i) $s/\alpha \geqq s \geqq 2$.
(ii) $\alpha^{3k+2} \log\log \alpha n \geqq 1$.
(iii) $\tau \geqq (2s(s-1)(s/\alpha)^s)/(s-2)$ *for* $s \geqq 3$.
(iv) $\tau \geqq 2s^k(s/\alpha)^{sk}2^{(k+1)s^k}$.
(v) $s^k(s/\alpha)^{sk}\tau^{(k+1)s^k} \leqq (s/\alpha)^{(k+2)s^{3k+2}}$.
(vi) $(s/\alpha)^{(k+2)s^{3k+2}} \leqq \alpha n$.

Proof. (i) and (ii) are consequences of our assumption that $n \geqq N$, $k \geqq 1$ and $\alpha < 1$. (iii), (iv) and (v) can be established in a straightforward manner by applying (i) and the inequalities $k \geqq 1$ and $k+1 \leqq s^k$ in a variety of circumstances. Finally (vi) follows from (ii) and the identity $\alpha n = (\log \alpha n)^{(\log \alpha n/\log\log \alpha n)}$. □

We are now ready to describe the basic ideas in our construction. We will construct a set X with the following properties:

Property 2.3.1. $\alpha n/2 \leqq |X| \leqq \alpha n$.

Property 2.3.2. For each i, $|(X+b_i)\backslash X| < |X|/s$.

Property 2.3.3. For each i such that $(a_i, n) \leqq s/\alpha$, $|a_iX\backslash X| < 2|X|/s$.

For any subset X of Z_n and $a \in Z_n$ we have $|aX\backslash X| \leqq |aX| \leqq |aZ_n| = n/(a, n)$. Thus Property 2.3.1 also implies the following additional property:

Property 2.3.4. If $(a_i, n) > s/\alpha$ then $|a_iX\backslash X| < 2|X|/s$.

Finally $|(a_iX+b_i)\backslash X| \leqq |((a_iX \cap X)+b_i)\backslash X| + |(a_iX\backslash X)+b_i| \leqq |(X+b_i)\backslash X| + |a_iX\backslash X|$, and hence the above properties imply the following property, as desired:

Property 2.3.5. For each i, $|f_i(X)\backslash X| < 3|X|/s$.

Let $P = \{a_i : (a_i, n) \leqq s/\alpha\}$, and let Q be the subset $\prod_{p \in P}\{1, p, \cdots, p^{s-1}\}$. For each p in P and $0 \leqq i \leqq s-1$, let $Q(p, i)$ be the subset $p^i\prod_{q \in P\backslash\{p\}}\{1, q, \cdots, q^{s-1}\}$. Thus Q is the set of elements of Z_n which can be written as a product of powers of elements of P in which the exponent of any element is at most $s-1$, and $Q(p, i)$ is the subset of elements of Q which can be so expressed with the exponent of p equal to i.

Next for each t with $\tau \leqq t \leqq n$ we define another subset $A(t)$ of Z_n by $A(t) = \{\sum_{z \in QB} a(z)z : a$ maps $QB \to \{0, 1, \cdots, t-1\}\}$, where $B = \{1, b_1, \cdots, b_k\}$. Now, finally, we define $X(t)$ as $X(t) = \bigcup_{q \in Q} q^{-1}A(t)$. We will show that $X(t)$ has Properties 2.3.2 and 2.3.3 for t in the range $\tau \leqq t \leqq n$. Moreover we will show that for some t in this range $X(t)$ also satisfies Property 2.3.1.

Before continuing with the proof we will attempt to provide some intuition as to why $X(t)$ has these properties. First of all, in order for any set X to have Property 2.3.2, it is clear that for each b_i it must be true that most of the elements of X can be arranged into long sequences of the form $x, x+b_i, x+2b_i, \cdots$, or in other words,

into long arithmetic progressions with period b_i. The set $A(t)$ is constructed so that for each q in Q, the set $q^{-1}A(t)$ (and hence also $X(t)$) can be arranged into arithmetic progressions with period b_i and of length at least t. This and its consequences are more formally presented in Lemmas 2.6 and 2.7.

Next let us consider why $X(t)$ should satisfy Property 2.3.3. The set Q is constructed so that $|Q(p, 0)|$ is small relative to $|Q|$ for each p in P. Moreover for any set A and $i>0$, if x is in $\bigcup_{q\in Q(p,i)} q^{-1}A$ then px is in $\bigcup_{q\in Q(p,i-1)} q^{-1}A$. Thus if $|q^{-1}A|\simeq|A|$ for each q in Q, one could hope that $(|pX\backslash X|/|X|)\simeq(|Q(p,0)|/|Q|)$. In general $|q^{-1}A|$ may be much smaller than $|A|$ ($q^{-1}A$ could be empty, for example), but one kind of set A which has $|q^{-1}A|\simeq|A|$ for every q is a long interval, i.e. $\{x, x+1, x+2, \cdots, x+j\}$ for sufficiently large j. This is expressed more precisely in Lemma 2.8. Examining the definition of $A(t)$ shows that $A(t)$ has been constructed so that it is the union of intervals of length at least t and so has the desired property. Since the sets $q^{-1}A(t)$ are not disjoint in general, the proof that $|pX\backslash X|$ is small is still quite complicated, and depends heavily on the fact that each set $q^{-1}A(t)$ is also the union of long intervals.

Finally we consider Property 2.3.1. It is obvious that $|X(t)|$ increases with t, and that for t large enough ($t=n$ for example) $X(t)$ is all of Z_n. What is harder to prove is that $|X(t)|$ increases slowly enough in the appropriate range so that there is some t with $\alpha n/2\leqq|X(t)|\leqq\alpha n$, and it is precisely for this reason that t is chosen to be so much larger than s.

We begin the proof by establishing some upper bounds on the size of our sets in terms of s, t and k.

LEMMA 2.4.
(i) $|Q|\leqq s^k$.
(ii) $|A(t)|\leqq t^{(k+1)sk}$
(iii) *For each* q in Q, $|q^{-1}A(t)|\leqq(s/\alpha)^{sk}|A(t)|$.
(iv) $|X(t)|\leqq s^k(s/\alpha)^{sk}t^{(k+1)sk}$.

Proof. (i) is obvious since $|P|\leqq k$, and (ii) follows directly from (i) since clearly $|A(t)|\leqq t^{|Q||B|}$. For the proof of (iii) note that for any subset Y of Z_n and any q in Z_n we have $|q^{-1}Y|\leqq(q,n)|Y|$. Moreover, it is easy to see that for any q in Q we have $(q,n)\leqq(s/\alpha)^{sk}$, which completes the proof of (iii). Finally (iv) follows in an obvious way from (i), (ii) and (iii). □

COROLLARY 2.5. $|X(\tau)|\leqq\alpha n$.

Proof. This follows immediately from inequalities (v) and (vi) of Lemma 2.2 and (iv) of Lemma 2.4. □

In order to prove that $X(t)$ has the properties that we desire we will need the following lemma describing the structure of $X(t)$ in terms of b-intervals. If $b\in Z_n$ and Y is a subset of Z_n, then we say Y is a b-interval of length m if the elements of Y are the elements of an arithmetic progression in Z_n of length m and of period b. Note that the actual cardinality of a b-interval will be less than its length if its length exceeds $n/(b,n)$, but if $m\leqq n$ then a 1-interval of length m is simply an interval of length m in the usual sense except that it is interpreted mod n. A b-block of a subset Y is a b-interval which is maximal with respect to containment in Y.

LEMMA 2.6. *For each q in Q, b in B, and x in $q^{-1}A(t)$ there is a b-interval Y of length t such that $z\in Y$ and $Y\subset q^{-1}A(t)$.*

Proof. Let a map $QB\to\{0,1,\cdots,t-1\}$ such that $qx=\sum_{z\in QB}a(z)z$. Then it is easy to check that the set $Y=\{x+jb: -a(qb)\leqq j\leqq t-1-a(qb)\}$ has the desired properties. □

COROLLARY 2.7. *For each b in B we have* $|(X(t)+b)\backslash X(t)|\leqq|X(t)|/t$.

Proof. Let $X_1, \cdots, X_d$ be the b-blocks of $X(t)$. By Lemma 2.6 each X_i is a b-interval of length at least t, and it is easy to see that this implies that for any i such that $|X_i| < t$ we must have $(X_i + b) \subset X_i$. Consequently $|(X(t)+b)\backslash X(t)| \leqq |\{i: |X_i| \geqq t\}| \leqq |X(t)|/t$ since the X_i are disjoint. □

In proving a similar result about $|pX(t)\backslash X(t)|$ for each $p \in P$, we will use the following observation, whose proof we omit since it is almost trivial.

LEMMA 2.8. *If Y is a 1-interval, then for any $r \in Z_n$ we have $|r^{-1}Y| \geqq |Y| - ((r, n) - 1)$.*

PROPOSITION 2.9. *If $p \in P$ then $|pX(t)\backslash X(t)| < 2|X(t)|/s$.*

Proof. This clearly holds for $s = 2$ so suppose $s \geqq 3$. For $0 \leqq i \leqq s-1$ let $D_i = \bigcup_{q \in Q(p,i)} q^{-1}A(t)$, and let $D = D_0 \backslash (\bigcup_{1 \leqq i \leqq s-1} D_i)$. Then it is easy to see that for $i \geqq 1$ we have $pD_i \subset D_{i-1}$, and hence $(pX(t)\backslash X(t)) \subset pD$. Thus it suffices to show that $|D| < 2|X(t)|/s$. Let $Y_1, \cdots, Y_d$ be the 1-blocks of D in increasing order with respect to size, and let $m = \max(0, \max\{i: |Y_i| < (s-2)t/(2s(s-1))\})$. We first show that $\sum_{1 \leqq i \leqq m} |Y_i| < (s-2)|X(t)|/(s(s-1))$. Since every 1-block of D_0 has length (and hence cardinality) at least t by Lemma 2.6, if $|Y_i| < (s-2)t/(2s(s-1))$ we must have that Y_i is adjacent to some 1-block of $(\bigcup_{1 \leqq j \leqq s-1} D_j)$ either on the right or on the left. Let us denote this 1-block as $b(Y_i)$. Notice that any particular 1-block of $(\bigcup_{1 \leqq j \leqq s-1} D_j)$ could be $b(Y_i)$ for at most two distinct i since it can border at most one of them on the right and at most one on the left. Thus $\sum_{1 \leqq i \leqq m} |b(Y_i)| \leqq 2|\bigcup_{1 \leqq j \leqq s-1} D_j| \leqq 2|X(t)|$. Finally, since Lemma 2.6 implies that $|b(Y_i)| \geqq t$ for each such i, we have $|Y_i| < (s-2)|b(Y_i)|/(2s(s-1))$, which completes this part of the proof.

It now suffices to prove that $\sum_{m < i \leqq d} |Y_i| < |X(t)|/(s-1)$, since $(s-2)/(s(s-1)) + 1/(s-1) = 2/s$. For convenience, if Z is a subset of Z_n and i is a nonnegative integer, we will use $p^{-i}Z$ to denote the subset $(p^i)^{-1}Z$. We first observe that if $0 \leqq i < j \leqq s-1$ then $p^{-i}D \cap p^{-j}D = \emptyset$, since if $x \in p^{-i}D \cap p^{-j}D$ then $p^i x \in D \cap D_{j-i}$, which contradicts the definition of D. Combining this with the fact that the Y_i are disjoint, it is easy to see that the sets $p^{-j}Y_i$ are disjoint, and hence $|X(t)| \geqq \sum_{m < i \leqq d} \sum_{0 \leqq j \leqq s-1} |p^{-j}Y_i|$. By Lemma 2.8 $|p^{-j}Y_i| \geqq |Y_i| - (p^j, n) + 1$, and since $(p, n) \leqq s/\alpha$, clearly $(p^j, n) \leqq (s/\alpha)^j$. Combining these observations, and recalling that $s/\alpha \geqq s \geqq 3$, we see that $|X(t)| > \sum_{m < i \leqq d} (s|Y_i| - (s/\alpha)^s)$. Moreover, since $t \geqq \tau$, Lemma 2.2(iii) implies $(s/\alpha)^s \leqq (s-2)t/(2s(s-1)) \leqq |Y_i|$, and hence $|X(t)| > \sum_{m < i \leqq d} (s-1)|Y_i|$, or equivalently $\sum_{m < i \leqq d} |Y_i| < |X(t)|/(s-1)$ as promised. □

The remainder of this section is devoted to showing that for $r = \max\{t: |X(t)| \leqq \alpha n\}$, we have $\alpha n/2 \leqq |X(r)| \leqq \alpha n$. Note that Corollary 2.5 guarantees that $r \geqq \tau$.

PROPOSITION 2.10. $|A(r+1)\backslash A(r)| \leqq \alpha n/(2s^k(s/\alpha)^{sk})$.

Proof. Let $C = A(r+1)\backslash A(r)$, and suppose $|C| > \alpha n/(2s^k(s/\alpha)^{sk})$. For each x in C we can choose a_x mapping $QB \to \{0, 1, \cdots, r\}$ such that $x = \sum_{z \in QB} a_x(z)z$. Also, for each such x we define a subset $T(x)$ of QB by $T(x) = \{z: a_x(z) = r\}$. Notice that $T(x) \neq \emptyset$ since otherwise we would have $x \in A(r)$. Finally for each nonempty subset Z of QB we define a subset $g(Z)$ of C as $g(Z) = \{x: T(x) = Z\}$. Now since $|C| > \alpha n/(2s^k(s/\alpha)^{sk})$, there must be some nonempty subset Z of QB with $|g(Z)| > \alpha n/(2s^k(s/\alpha)^{sk}2^{|QB|})$. As $r \geqq \tau$ and $|QB| \leqq (k+1)s^k$, by Lemma 2.2(iv) this implies $|g(Z)| > \alpha n/r$.

For each i with $1 \leqq i \leqq r$ let $y(i)$ be the element $\sum_{z \in Z} iz$. From the definition of $g(Z)$ it is easy to see that for each such i we have $(g(Z) - y(i)) \subset A(r)$. Moreover, we claim that if $1 \leqq i < j \leqq r$ we have $(g(Z) - y(i)) \cap (g(Z) - y(j)) = \emptyset$, since otherwise we would have $g(Z) \cap (g(Z) - y(j) + y(i)) = g(Z) \cap (g(Z) - y(j-i)) \neq \emptyset$; yet $g(Z) \cap (g(Z) - y(j-i)) \subset C \cap A(r) = \emptyset$. Thus $|A(r)| \geqq \sum_{1 \leqq i \leqq r} |g(Z)| = r|g(Z)| > \alpha n$, and

hence $|X(r)| > \alpha n$, which contradicts the definition of r. Consequently we must have $|C| \leqq \alpha n/(2s^k(s/\alpha)^{sk})$. □

COROLLARY 2.11. $|X(r)| > \alpha n/2$.

Proof. It suffices to show that $|X(r+1)\backslash X(r)| \leqq \alpha n/2$, since by the definition of r we have $|X(r+1)| > \alpha n$. By the same arguments used in the proof of Lemma 2.4(iii) we have $|q^{-1}(A(r+1)\backslash A(r))| \leqq (s/\alpha)^{sk}|(A(r+1)\backslash A(r)|$. Thus $|X(r+1)\backslash X(r)| \leqq |Q|(s/\alpha)^{sk}|A(r+1)\backslash A(r)| \leqq s^k(s/\alpha)^{sk}|A(r+1)\backslash A(r)| \leqq \alpha n/2$ by Proposition 2.10. □

If we take $X = X(r)$, combining Corollaries 2.7 and 2.11 with Proposition 2.9, we see that X satisfies Properties 2.3.1, 2.3.2 and 2.3.3, thus completing the proof of Theorem 2.1.

Remark 2.12. It is easy to see that by choosing s and τ in slightly different ways one can prove slightly different results. For example, by changing s to $\lfloor \alpha(\log \alpha n/\log\log \alpha n)^{1/(3k+2)} \rfloor$, one obtains the following result:

THEOREM 2.13. *For each real number α between 0 and 1 and each $n \geqq 0$ there exists a subset X of Z_n with $\alpha n/2 \leqq |X| \leqq \alpha n$, such that for $1 \leqq i \leqq k$ we have $|f_i(X)\backslash X| < 3|X|/\lfloor \alpha(\log \alpha n \log\log \alpha n)^{1/(3k+2)} \rfloor$.*

Notice that this avoids having to choose n sufficiently large at the expense of weakening the bound on $|f_i(X)\backslash X|$. Of course this theorem is trivially true for any subset $|X|$ when $s \leqq 3$, so really, when one takes Theorem 2.1 into consideration, this theorem is only interesting for n (approximately) in the range defined by the inequality $(3/\alpha)^{3k+2} \leqq \log \alpha n \leqq 2^{(1/\alpha)^{3k+2}}$.

Similarly our choosing $\alpha n/2$ and αn as the limits on the size of X were completely arbitrary. In fact if $0 < \beta < \alpha < 1$, there is a constant N depending on k, α and β, and a function $g(n, \alpha, \beta, k)$ going to infinity as n goes to infinity, such that for each $n \geqq N$ there exists a subset X of Z_n with $\beta n \leqq |X| \leqq \alpha n$ and $|f_i(X)\backslash X| \leqq 3|X|/g(n, \alpha, \beta, k)$.

3. Rational coefficients. The special problems, which occur in constructing non-expanding subsets for the case of rational coefficients, are basically caused by the way that the floor function $\lfloor x \rfloor$ interacts with the multiplying and taking inverses mod n. Our first goal in this section is to prove a result similar to Proposition 2.9. We wish to show that the subset of elements x of $X(t)$ such that $p^{-1}x$ is not contained in $X(t)$ is small relative to $|X(t)|$ for each p in P with $p \leqq s$. This result will be proved in Proposition 3.2, but first we prove a useful technical lemma.

For any subset Z of Z_n let $\beta(Z)$ denote the number of 1-blocks in Z.

LEMMA 3.1. *Let Z, V, W be subsets of Z_n, and let $p, t \in Z_n$ such that $0 < p < t < n$. Moreover, suppose that every 1-block in either V or W has length at least t, and that $pZ \subset V\backslash W$. Then there is a subset $h(Z)$ of $V\backslash W$ such that*

(i) $\beta(h(Z)) \leqq \beta(Z) + |h(Z)|/t$,

(ii) $|h(Z)| \geqq (|Z| - 2p\beta(Z))/(1 + (p-1)/t)$, *and*

(iii) $|h(Z)| \leqq |Z|$.

Proof. If $D = \{x, x+1, \cdots, y\}$ is a 1-block of Z, we will use $p\&D$ to denote the 1-block $\{px, px+1, \cdots, py\}$. Let $K = \bigcup\{p\&D : D$ is a 1-block of $Z\}$, and let $H = K \cap (V\backslash W)$. We first prove that in fact $H = K \cap V$. Clearly it suffices to show that for any 1-block D of Z we have $p\&D \cap (V\backslash W) = p\&D \cap V$. Suppose $z \in p\&D \cap V \cap W$. Since $pD \subset V\backslash W$, z cannot be in pD and hence for some adjacent pair $x, x+1$ in D, we have $px < z < p(x+1)$. This shows that the 1-block of W containing z has length at most $p-1$, which contradicts the assumption that every 1-block of W has length at least t.

We next prove that $|K\backslash V| \leqq (p-1)(|H|/t + \beta(K))$. Let Y be a 1-block of K. From the definition of K and the fact that $pZ \subset V$, it is easy to see that every 1-block

of $Y\backslash V$ has length at most $p-1$. Moreover, since every 1-block of V has length at least t it is easy to see that $\beta(Y\backslash V)$ is at most $|Y\cap V|/t+1$. Combining these we see that $|Y\backslash V|\leqq(p-1)(|Y\cap V|/t+1)$, which yields $|K\backslash V|\leqq(p-1)(|H|/t+\beta(K))$.

To complete the proof that H satisfies (ii), we will first show that $|K|\geqq|Z|-p\beta(K)$. Clearly every element of pZ is a multiple of p, and thus from the definition of K we see that every 1-block of K both begins and ends with a multiple of p. As at most one out of any (p,n) consecutive elements in 1-block can be a multiple of p (and hence an element of pZ), this shows that $|K|\geqq(p,n)(|pZ|-\beta(K))$. Obviously $|pZ|\geqq|Z|/(p,n)$, so $|K|\geqq|Z|-p\beta(K)$ as desired. Finally, we have $|H|=|K|-|K\backslash V|\geqq|Z|-p\beta(K)-(p-1)(\beta(K)+|H|/t)$, yielding $|H|\geqq(|Z|-2p\beta(Z))/(1+(p-1)/t)$ since obviously $\beta(K)\leqq\beta(Z)$.

Let K' be any subset of K with $\beta(K')\leqq\beta(K)$, and let $H'=K'\cap V$. Since every 1-block of V has length at least t, for each 1-block Y' of K' we must have $\beta(Y'\cap V)\leqq|Y'\cap V|/t+1$, and hence $\beta(H')\leqq\sum\{|Y'\cap V|t+1: Y'$ is a 1-block of $K'\}\leqq\beta(K')+|H'|/t\leqq\beta(Z)+|H'|/t$. If $|H|\leqq|Z|$ we may take $h(Z)$ to be H since the preceding remark shows that H satisfies (i). Otherwise take $h(Z)$ to be $K'\cap V$, where K' is a subset of K with $\beta(K')\leqq\beta(K)$ and $|K'\cap V|=|Z|$. To see that such a set K' must exist note that it is easy to construct a family $\{K(r): 1\leqq r\leqq|K|\}$ of nested subsets of K with $\beta(K(r))\leqq\beta(K)$ and $|K(r)|=r$. Now combining the facts that $|K(|K|)\cap V|>|Z|$ and $|K(r)\cap V|-|K(r-1)\cap V|\leqq 1$ for each $r>1$ shows that $|K(r)\cap V|=|Z|$ for some r. □

For each $p\in P$ and i with $0\leqq i\leqq s-1$ let $V(p,i)=\bigcup_{r\in Q(p,i)} r^{-1}A(t)$, and let $W(p,i)=\bigcup_{0\leqq j\leqq i} V(p,j)$. For convenience we also adopt the convention that $W(p,-1)=\varnothing$ for any p.

PROPOSITION 3.2. *For each p in P such that $p\leqq s$ we have*

$$|V(p,s-1)\backslash W(p,s-2)|\leqq\frac{2|X(t)|}{s}.$$

Proof. We assume $s\geqq 3$ since the proposition holds trivially for $s=2$. Let $\xi=(s-2)t/(2s(s-1))$, and let $Z(0)$ be the union of the 1-blocks of $V(p,s-1)\backslash W(p,s-2)$ which have length at least ξ. Then by the same argument as used in the proof of Proposition 2.9, we have $|(V(p,s-1)\backslash W(p,s-2))\backslash Z(0)|<(s-2)|X(t)|/(s(s-1))$, and hence it suffices to show that $|Z(0)|<|X(t)|/(s-1)$. Now observe that if $i\geqq 1$ then $p(V(p,i)\backslash W(p,i-1))\subset V(p,i-1)\backslash W(p,i-2)$. Moreover, by Lemma 2.6 every block in either $V(p,i)$ or $W(p,i-1)$ has length at least t. Thus by Lemma 3.1 we can recursively define $Z(i)=h(Z(i-1))$ such that $Z(i)\subset V(p,s-i-1)\backslash W(p,s-i-2)$, $\beta(Z(i))\leqq\beta(Z(i-1))+|Z(i)|/t$, and $\gamma(|Z(i)|-2p\beta(Z(i-1)))\leqq|Z(i)|\leqq|Z(i-1)|$ where $\gamma=1/(1+(p-1)/t)$. Since every 1-block in $Z(0)$ has length at least ξ, clearly $\beta(Z(0))\leqq|Z(0)|/\xi$. Also obviously $|Z(i)|/t\leqq|Z(0)|/\xi$, and hence by induction one can trivially show that $\beta(Z(i))\leqq 3^i|Z(0)|/\xi$. Using this, again by induction it is easy to show that $|Z(i)|\geqq\gamma^i|Z(0)|-3^ip|Z(0)|/\xi$. Since the sets $V(p,s-i-1)\backslash W(p,s-i-2)$ are disjoint for $0\leqq i\leqq s-1$, the sets $Z(i)$ are disjoint, and hence $|X(t)|\geqq\sum_{0\leqq i\leqq s-1}|Z(i)|\geqq|Z(0)|\sum_{0\leqq i\leqq s-1}(\gamma^i-3^ip/\xi)$. Now $\gamma^i=(1/(1+(p-1)/t))^i$, and it is easy to verify that $(1/(1+(p-1)/t))^i\geqq 1-i(p-1)/t$. This shows that $|X(t)|\geqq|Z(0)|(s-s^2(p-1)/t-3^sp/\xi)$. Finally it can easily be checked that $s^2(p-1)/t+3^sp/\xi<1$ since $t\geqq\tau$, $s\geqq 3$, $s\geqq p$ and $k\geqq 1$. □

Proposition 3.2 yields the following corollary which will be useful for proving the nonexpansion of shuffle-exchange graphs, as well as of $G(n,F)$ when the mappings in F have rational coefficients.

COROLLARY 3.3. *For each p in P such that $p \leqq s$ we have*

$$|\{x \in X(t): \{\lfloor x/p \rfloor + jn/(p,n): 0 \leqq j < (p,n)\} \backslash X(t) \neq \varnothing\}| < \left(\frac{2}{s} + \frac{p}{t}\right)|X(t)|.$$

Proof. Let W be the set of elements of $W(p, s-2)$ which are among the first p elements of their 1-block in $W(p, s-2)$. We claim that for each x in $W(p, s-2)\backslash W$ we have $\{\lfloor x/p \rfloor + jn/(p,n): 0 \leqq j < (p,n)\} \subset X(t)$. First note that since $x - p\lfloor x/p \rfloor \leqq p-1$ we have $p\lfloor x/p \rfloor \in W(p, s-2)$ and hence $p^{-1}(p\lfloor x/p \rfloor) \subset X(t)$. However, $p^{-1}(p\lfloor x/p \rfloor) = \{\lfloor x/p \rfloor + jn/(p,n): 0 \leqq j < (p,n)\}$. Thus $|\{x \in X(t): \{\lfloor x/p \rfloor + jn/(p,n): 0 \leqq j < (p,n)\} \backslash X(t) \neq \varnothing\}| \leqq |X(t) \backslash W(p, s-2)| + |W|$. Clearly $X(t) \backslash W(p, s-2) = V(p, s-1) \backslash W(p, s-2)$ so we have $|X(t) \backslash W(p, s-2)| < 2|X(t)|/s$ by Proposition 3.2. Moreover, since every 1-block of $W(p, s-2)$ has size at least t, we have $|W| \leqq p|W(p, s-2)|/t \leqq p|X(t)|/t$, which completes the proof. □

Let $X_1 = \{x \in X(t): \lfloor x/q \rfloor$ is not in $X(t)\}$, $X_2 = \{x \in X(t): px$ is not in $X(t)\}$, and $X_3 = \{x \in X(t): \{x, x+1, \cdots, x+p-1\} \backslash X(t) \neq \varnothing\}$. In the following corollary we will use $*$ to distinguish real multiplication from multiplication mod n. Thus for p and x in Z_n $p * x$ denotes the product of p and x regarded as real numbers, whereas px denotes the product mod n.

COROLLARY 3.4. *If $p, q \in P$ and $q \leqq s$ then* $|(\lfloor p * X(t)/q \rfloor \bmod n) \backslash X(t)| \leqq (2*(q+1)/s + q*(p*p+1)/t)|X(t)|$.

Proof. We first show that $|(\lfloor p * X(t)/q \rfloor \bmod n) \backslash X(t)| \leqq |X_1| + q*|X_2| + p*q*|X_3|$. Let $Y = \{x \in X(t): \{\lfloor x/q \rfloor\} \cup \{p\lfloor x/q \rfloor, p\lfloor x/q \rfloor + 1, \cdots, p\lfloor x/q \rfloor + p - 1\} \subset X(t)\}$. It is not hard to see that for any x we have $\lfloor p * x/q \rfloor \bmod n \in \{p\lfloor x/q \rfloor, p\lfloor x/q \rfloor + 1, \cdots, p\lfloor x/q \rfloor + p - 1\}$, and hence we see that $\lfloor p * Y/q \rfloor \bmod n \subset X(t)$. This shows that $|\lfloor p * X(t)/q \rfloor \backslash X(t)| \leqq |X(t) \backslash Y|$. Now clearly $|X(t) \backslash Y| \leqq |X_1| + |\{x: \lfloor x/q \rfloor \in X_2\}| + |\{x: p\lfloor x/q \rfloor \in X_3\}|$, from which it is easy to see that $X|(t) \backslash Y| \leqq |X_1| + q*|X_2| + p*q*|X_3|$.

The proof is completed by giving appropriate upper bounds for $|X_1|$, $|X_2|$ and $|X_3|$. From Corollary 3.3 we have $|X_1| < (2/s + q/t)|X(t)|$, and from the proof of Proposition 2.9 it is easy to see that $|X_2| < (2/s)|X(t)|$. Finally, since every 1-block in $X(t)$ has length at least t, one easily concludes that $|X_3| < (p/t)|X(t)|$. □

Combining this corollary with the results of the previous section yields the following theorem.

THEOREM 3.5. *Let F be the family $\{p_i x/q_i + b_i: 1 \leqq i \leqq k\}$ and let $0 < \alpha < 1$. Then there exists a constant N depending only on α and k such that for each $n \geqq N$ there exists a subset X of inputs of $G(n, F)$ with $\alpha n/2 < |X| \leqq \alpha n$ and $|\Gamma X| < (1 + \delta(\alpha, F, n))|X|$, where $\delta(\alpha, F, n)$ is the function $\sum_{1 \leqq i \leqq k} ((3+q_i)/s + (q_i(p_i^2+1))/\tau)$ where*

$$s = \lfloor (\log \alpha n/\log \log \alpha n)^{1/(3k+2)} \rfloor \quad \textit{and} \quad \tau = \lfloor (s/\alpha)^{(k+1)s^{k+2}} \rfloor.$$

If d is a divisor of n, the perfect d-shuffle rearranges the numbers 1 to n into the sequence

$$1, (n/d)+1, \cdots, ((d-1)n/d)+1,$$
$$2, (n/d)+2, \cdots, ((d-1)n/d)+2, \cdots,$$
$$(n/d), (n/d)+(n/d), \cdots, n$$

which corresponds to partitioning the numbers 1 to n into d segments of equal length and performing a perfect shuffle. Suppose D is a subset of the divisors of n. If a graph has inputs $x(i)$ and outputs $y(i)$ for $1 \leqq i \leqq n$, we say that it is a D-shuffle-exchange graph if $x(i)$ and $y(j)$ are adjacent whenever for some d in D the perfect d-shuffle

places i in the jth position, or j in the ith position. By this definition, the usual shuffle-exchange graph is simply a $\{2\}$-shuffle-exchange graph. It is not hard to see that the D-shuffle-exchange graph is a partial subgraph of $G(n, F(D))$ where $F(D)$ is the family $\{dx+b: d \in D, 0 \leqq b \leqq d-1\} \cup \{x/d + jn/d + 1: d \in D, 0 \leqq j \leqq d-1\}$. If we take $P = D$ and $B = \{1\}$, then it is not hard to see from Lemma 2.6 and Corollary 3.3 that, regarding $X(t)$ as a subset of inputs in $G(n, F(D))$, if $d \leqq s$ for each d in D we have $|\Gamma X(t)| < (1 + \sum_{d \in D} (4/s + 2d/\tau))|X(t)| < (1 + 5|D|/s)|X(t)|$ since $d \leqq s$ obviously implies $2d/\tau < 1/s$. This shows that as long as the divisors used are small enough relative to n, shuffle exchange graphs cannot be expanding graphs.

Acknowledgment. I would like to thank Nick Pippenger for many illuminating discussions.

REFERENCES

[1] H. ABELSON, *A note on time-space tradeoffs for computing continuous functions*, Inform. Process. Lett., 8 (1979), pp. 215–217.

[2] M. AJTAI, J. KOMLOS AND E. SZEMEREDI, *An $O(n \log n)$ sorting network*, in Proc. 15th Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1983.

[3] L. BASSALYGO AND M. PINSKER, *Complexity of an optimum nonblocking switching network without reconnections*, Problemy Peredachi Informatsii, 9, 1 (1973), pp. 84–87; Problems Inform. Transmission, 9, 1 (1974), pp. 64–66.

[4] R. C. BOSE AND R. J. NELSON, *A sorting problem*, J. Assoc. Comput. Mach., 9 (1962), pp. 282–296.

[5] D. CANTOR, *On nonblocking switching networks*, Networks, 1 (1971), pp. 367–377.

[6] F. R. K. CHUNG, *On concentrators, superconcentrators, and nonblocking networks*, Bell System Tech. J., 58 (1979), pp. 1765–1777.

[7] P. ERDOS, R. L. GRAHAM AND E. SZEMEREDI, *On sparse graphs with dense long paths*, Comput. Math. Appl., 1 (1975), pp. 365–369.

[8] O. GABBER AND Z. GALIL, *Explicit constructions of linear size superconcentrators*, Proc. 20th Annual Symposium on the Foundations of Computer Science, 1979, pp. 364–370.

[9] J. JA'JA', *Time-space tradeoffs for some algebraic problems*, Proc. 12th Annual ACM Symposium on the Theory of Computing, 1980, pp. 339–350.

[10] R. KARP AND N. PIPPENGER, *A time-randomness trade-off*, in preparation.

[11] M. KLAWE, *Nonexistence of one-dimensional expanding graphs*, Proc. 22nd Annual Symposium on the Foundations of Computer Science, Nashville, TN, 1981.

[12] T. LENGAUER AND R. TARJAN, *Asymptotically tight bounds on time-space trade-offs in a pebble game*, J. Assoc. Comput. Mach., 29 (1982), pp. 1087–1130.

[13] G. MARGULIS, *Explicit constructions of concentrators*, Problemy Peredachi Informatsii, 9(4) (1973), pp. 71–80; Problems Inform. Transmission, 10 (1975), pp. 325–332.

[14] G. MASSON AND B. JORDAN JR., *Generalized multi-stage connection networks*, Networks, 2 (1972), pp. 191–209.

[15] JU. P. OFMAN, *A universal automaton*, Trans. Moscow Math. Soc., 14 (1965), pp. 200–215.

[16] W. PAUL AND R. REISCHUK, *On Alternation* II—*a graph theoretic approach to determinism versus nondeterminism*, Acta Inform., 14 (1980), pp. 391–403.

[17] W. PAUL AND R. TARJAN, *Time-space trade-offs in a pebble game*, Acta Inform., 10 (1978), pp. 111–115.

[18] W. PAUL, R. TARJAN AND J. CELONI, *Space bounds for a game on graphs*, Math. Systems Theory, 10 (1979), pp. 239–251.

[19] M. PINSKER, *On the complexity of a concentrator*, in 7th International Teletraffic Conference, Stockholm, June 1973, pp. 318/1–318/4.

[20] N. PIPPENGER, *Superconcentrators*, SIAM J. Comput., 6 (1977), pp. 298–304.

[21] ———, *Generalized connectors*, SIAM J. Comput., 7 (1978), pp. 510–514.

[22] ———, *Pebbling with an auxiliary pushdown*, J. Comput. System Sci., 23 (1981), pp. 151–165.

[23] R. M. TANNER, *Explicit concentrators from generalized N-gons*, SIAM J. Alg. Discr. Meth., 5 (1984), to appear.

[24] M. TOMPA, *Time-space tradeoffs for computing functions, using connectivity properties of their circuits*, J. Comput. System Sci., 20 (1980), pp. 118–132.

[25] L. VALIANT, *Graph theoretic properties in computational complexity*, J. Comput. System Sci., 13 (1976), pp. 278–285.

SIAM J. COMPUT.
Vol. 13, No. 1, February 1984
© 1984 Society for Industrial and Applied Mathematics
0097-5397/84/1301-0014 $01.25/0

PUSHDOWN PERMUTER CHARACTERIZATION THEOREM*

PRAKASH V. RAMANAN†

Abstract. There is a well-known class of algorithms for permuting symbols which has been formally characterized by a device called a pushdown permuter. Two theorems attempting to characterize the type of permutations that can be achieved by a pushdown permuter have appeared in the literature. Counterexamples to these theorems have also appeared in the literature. This paper presents a correct statement of this theorem.

Key words. algorithm, permutation, stack, pushdown permuter

1. Introduction. There is a well-known algorithm which reads infix arithmetic expressions from left to right, one character at a time up to an end-marker, and, using a pushdown stack, produces from left to right, one character at a time, the suffix form of the expressions. This algorithm belongs to a well-known class of algorithms, characterized by their use of a single pushdown stack and a finite number of random access memory cells, to shuffle the characters about between the input string and the output string. Reingold [4] has developed a formal model for this class of algorithms, which he calls a pushdown permuter, and has attempted to characterize the type of permutations that can be achieved by a pushdown permuter. Carlson [1] has presented a counterexample to Reingold's characterization theorem. Shyamasundar [5] has also attempted to characterize this type of permutations. Ramanan [3] has presented a counterexample to Shyamasundar's characterization. In this paper we present a correct theorem characterizing the type of permutations that can be achieved by a pushdown permuter.

2. Pushdown permuter. Reingold defines a *pushdown permuter* (*p.d.p.*) to be a variant of a one-way, deterministic, finite state pushdown transducer whose finite input, output, and stack alphabets coincide. Informally, a p.d.p. can perform only the following kinds of steps: (a) It can read the input string one character at a time from left to right until it reaches an end-marker. (b) The character read from the input may be put directly into the output, which is also produced one character at a time from left to right, or it may be put on the top of a pushdown stack. (c) At any time, the only element accessible on the stack is the top element which can, if desired, be popped off the stack allowing access to the second element in the stack. The element popped off the stack can be thrown away, or it can be put into the output string. Once a symbol has been put in the output string it is forever after inaccessible and immutable.

A p.d.p. is nothing more than a control for a "switchyard" arrangement between the input string, the stack, and the output string. When the function of a p.d.p. is limited to just the permutation of the symbols from the input string, the capability of a p.d.p. to throw away symbols is not needed.

3. Pushdown permuter characterization. To simplify the notation we consider $12 \cdots n$ to be the input string and $p_1 p_2 \cdots p_n$ to be the output string. Knuth [2, § 2.2.1, ex. 5] has shown that if the p.d.p. can have access to nothing except the top

* Received by the editors January 22, 1982, and in revised form February 10, 1983. This research was supported in part by the Office of Naval Research under grant N00014-79-C-0775.

† Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801.

stack symbol, then $p_1p_2 \cdots p_n$ can be produced if and only if there is no subsequence[1] $p_ip_jp_k$ of $p_1p_2 \cdots p_n$ for which $p_i > p_k > p_j$.

Reingold's pushdown permuter characterization theorem attempts to generalize the result stated in the previous paragraph to the case in which some fixed number of symbols can be stored in a random access memory, in addition to the stack. His characterization is as follows:

"A p.d.p. with M memory cells can permute the input string $12 \cdots n$ to $p_1p_2 \cdots p_n$ if and only if there is no subsequence $xy_1 \cdots y_{M+1}z_1 \cdots z_{M+1}$ of $p_1p_2 \cdots p_n$ such that for all i and j, $x > z_i > y_j$."

Carlson has presented a counterexample to this characterization. We present the following characterization:

THEOREM. *A p.d.p. with M memory cells can permute the input string $12 \cdots n$ to $p_1p_2 \cdots p_n$ if and only if there is no subsequence $p_rp_{i_1}p_{i_2} \cdots p_{i_{M+1}}$ of $p_1p_2 \cdots p_n$ satisfying the following conditions:*

i) *p_r is the largest symbol preceding p_{i_1} in $p_1p_2 \cdots p_n$. Moreover $p_r > p_{i_t}$ for all t, $1 \leqq t \leqq M+1$.*

ii) *Let $w = \max(p_{i_1}, p_{i_2}, \cdots, p_{i_{M+1}})$. For any prefix $p_1p_2 \cdots p_m$, $r \leqq m \leqq i_{M+1}$, of $p_1p_2 \cdots p_n$, let $f_m = \max(p_1, p_2, \cdots, p_m)$. Then*

$$|\{p_j \mid j > m,\ w < p_j \leqq f_m\}| \geqq |\{i_t \mid i_t \leqq m\}|.$$

Proof. First we shall prove the *if* part of the theorem. For a permutation $p_1p_2 \cdots p_n$ of $12 \cdots n$ consider the p.d.p. which behaves as follows: At each input symbol e the p.d.p. places e into a vacant memory cell, if one exists; otherwise it examines e and the symbols in the M memory cells, and, of those $M+1$ symbols, it puts the one which appears right-most in $p_1p_2 \cdots p_n$ onto the stack. If at any point $p_1p_2 \cdots p_k$ has been put into the output, and p_{k+1} is in one of the memory cells, is at the top of the stack, or is the next symbol in the input string, then p_{k+1} is put into the output and if a memory cell is vacated, it is filled with the top symbol in the pushdown stack, which is popped up. If there are no more input symbols, then, since we kept the left-most occurring symbols in the memory cells, we put them into the output in the appropriate order, filling the vacated cells with symbols taken from the top of the pushdown stack. This process continues until all of the symbols have been put into the output, or, perhaps, until the p.d.p. gets stuck with all memory cells filled and the symbol which must be put next into the output inaccessible on the stack. Clearly, if the p.d.p. does not get stuck, it will produce $p_1p_2 \cdots p_n$. We must verify that if the p.d.p. gets stuck, then $p_1p_2 \cdots p_n$ has a subsequence of the form mentioned in the theorem.

Suppose at some point the p.d.p. gets stuck with all memory cells filled and the symbol u which must be put into the output next, inaccessible below the top of the stack. Consider the p.d.p. at the time the symbol u is put into the stack for the last time, never to come out again, and call the contents of the M memory cells at that time $y_1, y_2, \cdots, y_M$. Since the y_i stay in the memory cells while u is put into the stack, they must precede u in $p_1p_2 \cdots p_n$, and since the p.d.p. does not get stuck until trying to put u into the output, the y_i are all in the output at the time the p.d.p. does get stuck. Let x be the largest symbol in the output preceding all of the y_i at the time the p.d.p. gets stuck. Clearly, $x > y_i$ and $x > u$ because at the time u goes into the stack for the last time, the p.d.p. is "waiting" for some symbol still in the input, while u and the y_i have already been read.

---

[1] We define $p_{i_1}p_{i_2} \cdots p_{i_k}$ to be a subsequence of $p_1p_2 \cdots p_n$ provided that $1 \leqq i_1 < \cdots < i_k \leqq n$.

So far our proof is identical to that of Reingold's. At the time the symbol u is put into the stack for the last time, never to come out again, the largest symbol the p.d.p. has read from the input is max $(y_1, y_2, \cdots, y_M, u)$. Consider the p.d.p. at some later time t' before it gets stuck. Our assumption concerning the last time the symbol u is put into the stack implies that the number of symbols read from the input by the p.d.p. after u is put into the stack and until time t' be at least as large as the number of symbols put into the output by the p.d.p. during the same time period; otherwise, at some time during this period, the p.d.p. will pop up u from the stack and place it into a vacant memory cell. This argument holds true at any time, after u is put into the stack for the last time. Hence, we obtain the following condition: Let $p_1p_2 \cdots p_m$ be any prefix of $p_1p_2 \cdots p_n$ such that $x \in \{p_1, p_2, \cdots, p_m\}$, and $u \notin \{p_1, p_2, \cdots, p_m\}$. Let $w = \max(y_1, y_2, \cdots, y_M, u)$ and $f_m = \max(p_1, p_2, \cdots, p_m)$. Then

$$f_m - w \geqq |\{p_j | 1 \leqq j \leqq m, p_j > w\}| + |\{y_i | y_i = p_j \text{ for some } j \leqq m\}|.$$

Moreover, it is clear that if we take the prefix $p_1 p_2 \cdots p_k$ up to, but not including u, strict inequality should hold; otherwise, u will be accessible at the top of the stack. If we let $p_r = x, \{p_{i_1}, p_{i_2}, \cdots, p_{i_M}\} = \{y_1, y_2, \cdots, y_M\}$, and $p_{i_{M+1}} = u$, for some $i_1 < i_2 < \cdots < i_{M+1}$, we have proved the existence of a subsequence of the form mentioned in the theorem.

Now we shall prove the *only if* part of the theorem. Let $p_1p_2 \cdots p_n$ contain a subsequence of the form mentioned in the theorem. Since $p_r > p_{i_t}$ for all t, $1 \leqq t \leqq M+1$, all of the $M+1$ p_{i_t} must be read and stored before we read p_r and put it into the output. Since there are $M+1$ p_{i_t}, they cannot all be stored within the M memory cells. So at least one of the p_{i_t} must be on the stack at the time p_r is put into the output. Let p_{i_q} be the bottom-most p_{i_t} on the stack at this time. Condition ii) of the theorem for $r \leqq m < i_q$ guarantees that p_{i_q} can never be popped off the stack, and placed in a memory cell. When $m = i_q$, it guarantees that p_{i_q} is not at the top of the stack. Hence the p.d.p. gets stuck when it tries to put p_{i_q} into the output.

Acknowledgments. The author is grateful to Edward Reingold and M. S. Paterson for their valuable suggestions during the preparation of this paper.

REFERENCES

[1] C. R. CARLSON, *A counterexample to Reingold's pushdown permuter characterization theorem*, this Journal, 8 (1979), pp. 199–201.

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

[3] P. V. RAMANAN, *A counterexample to Shyamasundar's characterization of pushdown permuters*, Theoret. Comp. Sci., 23 (1983), pp. 103–105.

[4] E. M. REINGOLD, *Infix to prefix translation: The insufficiency of a pushdown stack*, this Journal, 1 (1972), pp. 350–353.

[5] R. L. SHYAMASUNDAR, *On a characterization of pushdown permuters*, Theoret. Comp. Sci., 17 (1982), pp. 333–341.

SIAM J. COMPUT.
Vol. 13, No. 1, February 1984

© 1984 Society for Industrial and Applied Mathematics
0097-5397/84/1301-0013 $01.25/0

TIGHTER BOUNDS FOR THE MULTIFIT PROCESSOR SCHEDULING ALGORITHM*

DONALD K. FRIESEN†

Abstract. This paper considers the problem of nonpreemptively scheduling n independent jobs on m identical, parallel processors with the object of minimizing the "makespan", or completion time for the entire set of jobs. Coffman, Garey, and Johnson [SIAM J. Comput., 7 (1978), pp. 1–17] described an algorithm MULTIFIT which has a considerably better worst case performance than the largest processing time first algorithm. In this paper we tighten the bounds obtained in that paper on the worst case behavior of this algorithm by giving an example showing that it may be as bad as 13/11 and proving that it can be no worse than 6/5.

Key words. binpacking, multiprocessor scheduling, approximation algorithms, worst case analysis, performance bounds

1. Introduction. A fundamental problem in deterministic scheduling theory is that of nonpreemptively scheduling independent tasks on a multiprocessor system to minimize the completion time for the entire set of tasks. Since this problem is known to be NP-complete [5], it is unlikely that an efficient algorithm will be found for obtaining the optimal schedule.

Much work has been done in studying approximation algorithms for various scheduling problems. For the problem discussed here, the "largest processing time first", or LPT schedule was analyzed by R. L. Graham [3]. The length of the LPT schedule on an M processor system was shown to be at most $4/3-1/(3M)$ times the length of the optimal schedule. More complicated algorithms were described in [4], which can be used to obtain results as close to optimal as desired, but their running time grows rapidly as the accuracy desired is increased, and they are exponential in M whereas Graham's was not.

More recently, in [1] an algorithm called MULTIFIT was presented, which is computationally comparable to the LPT algorithm. In this paper we propose to analyze somewhat more closely the algorithm MULTIFIT. In [1] it was shown that the length, MF (L), of the schedule produced by MULTIFIT for any list of tasks, L, and any number of processors satisfied

$$\mathrm{MF}\,(L) \leqq 1.22\mathrm{OPT}\,(L)+\frac{1}{2^k}$$

where OPT (L) is the length of the optimal schedule and k is the number of iterations used. The authors conjectured that the asymptotic bound was in fact (20/17) OPT (L), the worst case example known at the time. They also determined bounds which are tight for $\leqq 7$ processors.

In this paper we tighten the bound by showing that MF $(L) \leqq (1.2)$ OPT $(L)+1/2^k$ and by giving an example showing that there exist lists L for which MF $(L)=$ (13/11) OPT (L). (Note that $13/11=1.1818\ldots$ and $20/17=1.176\ldots.$)

2. Background. The scheduling problem we are examining may be formulated in the following way. There are $M \geqq 2$ identical processors and n independent tasks $T_1, T_2, \cdots, T_n$, each T_i having length $s(T_i)$. A schedule is an assignment of tasks to processors. The length of the schedule is the maximum of the sums of the lengths of

* Received by the editors February 18, 1980, and in revised form July 15, 1982.
† Department of Industrial Engineering, Texas A&M University, College Station, Texas 77843.

the tasks assigned to each processor. An optimal schedule is one for which the length is as small as possible.

In both the LPT schedule and in MULTIFIT, the tasks are first sorted into nonincreasing order of length. For the LPT schedule the tasks are assigned in decreasing order so that whenever a processor finishes a task, it is assigned to the longest task not yet assigned. Algorithm MULTIFIT is based on the "first fit decreasing" bin packing algorithm.

By regarding the processors as bins and the tasks T_i as items of size $s(T_i)$, the completion of a schedule by time t can be considered as the successful packing of the n items into M bins of size t. In the FFD algorithm for bin packing the bins are numbered from 1 to M and the items, pre-sorted into nonincreasing order of size, are packed sequentially, each going into the lowest numbered bin in which it will fit. By trying to pack the items using different bin sizes t, schedules of different length can be obtained. More precisely, we use an upper bound, UP, and a lower bound, LOW, on the length of the schedule we seek. A number of choices are available, for example, $\mathrm{LOW}=\sum_{x\in L} s(x)/M$ and $\mathrm{UP}=2\,\mathrm{LOW}$ as shown in [1]. Using a binary search technique, we can attempt an FFD packing initially using a bin size (schedule length) of $S=(\mathrm{UP}+\mathrm{LOW})/2$. Whenever we succeed, we decrease UP to S; and when we fail, we increase LOW to S.

We would like, then, to obtain an upper bound α such that $\mathrm{MF}\,(L)\leqq\alpha\ \mathrm{OPT}\,(L)$ for all lists L. Suppose we find α_0, the smallest number α for which the FFD algorithm is guaranteed to pack the list L in M bins of size α whenever there exists a packing into M bins of size 1. If we introduce the notation $\mathrm{FFD}\,(\alpha, L)$ to denote the number of bins of size α used by the FFD algorithm to pack the list of items L, and $\mathrm{OPT}\,(1, L)$ to denote the minimum number of bins of size 1 required to pack the list L, we can describe the goal as finding the smallest α for which $\mathrm{FFD}\,(\alpha, L)\leqq\mathrm{OPT}\,(1, L)$ for all lists L of items of size $\leqq 1$. (Clearly, the size of the largest item can be normalized to be $\leqq 1$, for any list by an appropriate choice of units.) Then [1, Thm. 3.1] guarantees that after k iterations of our binary search procedure, we will obtain a schedule length $\mathrm{MF}_k(L)$ satisfying

$$\mathrm{MF}_k\,(L)\leqq\left(\alpha_0+\frac{1}{2^k}\right)\mathrm{OPT}\,(L).$$

In [1], the inequality $\mathrm{FFD}\,(\alpha, L)\leqq\alpha\ \mathrm{OPT}\,(1, L)$ was shown for $\alpha=1.22$. We will prove in § 5 that it holds for $\alpha=1.2$ and in § 3, we give a list L for which $\mathrm{FFD}\,(\alpha, L)>\mathrm{OPT}\,(1, L)$ for all $\alpha<13/11$. Thus the worst case bound on the ratio of the length of the schedule is in the interval $[13/11, 6/5]$.

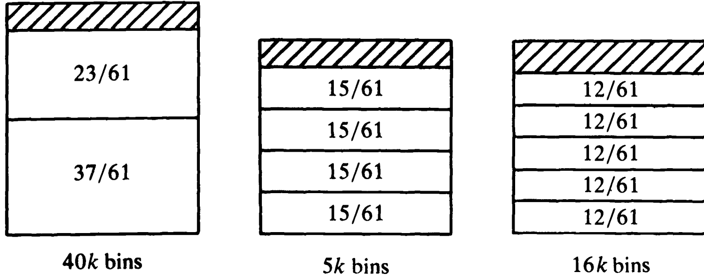
3. Examples. In [2], a parametric approach to bin packing was used to determine the effect of changing the bin size on various bin packing algorithms. Letting α denote the bin size and L an arbitrary list satisfying $s(b)\leqq\min(1,\alpha)$ for all $b\in L$, the $\limsup\,(\mathrm{FFD}\,(\alpha, L)/\mathrm{OPT}\,(1, L))$ was determined for all $\alpha\leqq 3$. Typical of these results is the following theorem ([2, Thm. 3.4.10]).

THEOREM 3.1. *For any list* L, $\mathrm{FFD}\,(\alpha, L)\leqq\mathrm{OPT}\,(1, L)+1$ *if* $\alpha>72/61$. *The techniques used to prove this result are similar to those in this paper and derive from* [1].

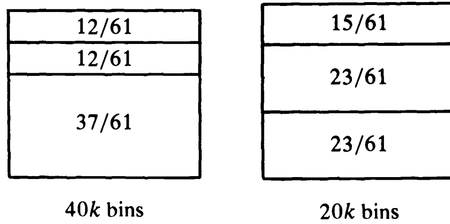
If we examine this result from the standpoint of scheduling with algorithm MULTIFIT, we can conclude that $\mathrm{MF}\,(L)\leqq(72/61)\,\mathrm{OPT}\,(L)$ if we can use one additional processor. Example 3.1 shows that 72/61 is indeed the smallest number α for which the statement is true. Unfortunately, as Example 3.2 shows, the additional processor may be required, at least if $\alpha<13/11$. Consequently the worst case bound for MULTIFIT is at least 13/11. We state this formally in the following theorem.

THEOREM 3.2. *For any* $M \geqq 13$, *there is a list* L *such that* $\text{FFD}(\alpha, L) > \text{OPT}(1, L) = M$ *if* $\alpha < 13/11$. *Equivalently, for* 13 *or more processors, the worst case bound for algorithm* MULTIFIT *is at least* 13/11.

Example 3.1. $L = \{a_1, a_2, \cdots, a_{40k}, b_1, b_2, \cdots, b_{40k}, c_1, c_2, \cdots, c_{20k}, d_1, d_2, \cdots, d_{80k}\}$, $s(a_i) = 37/61$, $s(b_i) = 23/61$, $s(c_i) = 15/61$, $s(d_i) = 12/61$ for all i $\text{FFD}(\alpha, L) = 61k$ for $\alpha \in [1, 72/61]$ since the packing of Fig. 3.1(a) will result. $\text{OPT}(1, L) = 60k$ since L can be packed as in Fig. 3.1(b).



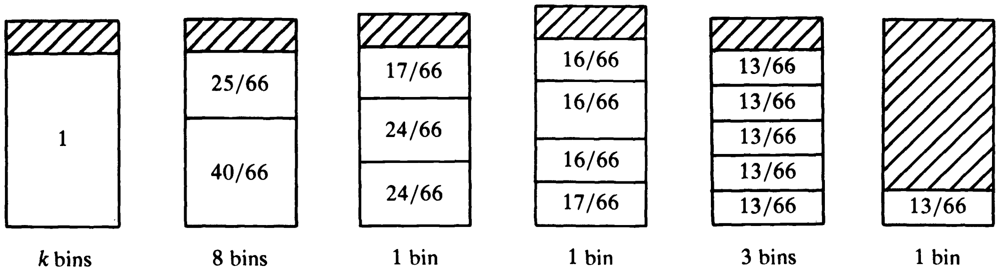
(a) FFD *packing of L for Example* 3.1.



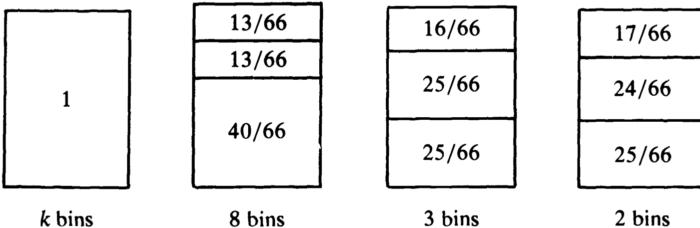
(b) OPT *packing of L for Example* 3.1.

FIG. 3.1

Example 3.2. L contains $k+39$ items, the FFD packing used $k+14$ bins of size $\alpha \in [1, 13/11)$, $\text{OPT}(1, L)$ is $k+13$. The sizes and packings are given in Fig. 3.2.



(a) FFD *packing of L for Example* 3.2.



(b) OPT *packing of L for Example* 3.2

FIG. 3.2

4. Notation and preliminary results. In § 5, we will prove that FFD $(6/5, L) \leqq$ OPT $(1, L)$ for all lists L. Since we will proceed by assuming that we have a minimal counterexample and derive a contradiction, we need to know some properties of a minimal counterexample. Essentially a counterexample is minimal if the number of bins is minimal and if, given the number of bins, the number of items is minimal. We begin by stating some definitions and results from [1] for reference.

A minimal counterexample L is a list satisfying

(i) FFD $(6/5, L) >$ OPT $(1, L) = M$;

(ii) For all lists L' satisfying (i), $|L'| \geqq |L|$;

(iii) For all lists L' satisfying (i), OPT $(1, L') \geqq M$.

From now on we assume that L is a minimal counterexample. If m is the smallest item in the list L, we have the following six lemmas, from [1], with trivial changes.

LEMMA 4.1. (a) FFD $(6/5L) =$ OPT $(1, L) + 1$.

(*b*) *All items of L but m are packed in the first* OPT $(1, L)$ *bins. We say that, for subsets X and Y of L, that X dominates Y if there is a one-to-one function* $f: Y \to X$ *such that* $s(t) \leqq s(f(y))$ *for all* $y \in Y$.

LEMMA 4.2. (cancellation lemma). *If B is a bin of the* FFD *packing and* B^* *is a bin of the* OPT *packing of L, then B cannot dominate* B^*.

LEMMA 4.3. *In the* OPT $(1, L)$ *packing, no bin* B^* *has fewer then* 3 *items.*

LEMMA 4.4. $s(m) = 1/5 + \Delta, \Delta > 0$.

LEMMA 4.5. *For any* $b \in L$, $s(b) \leqq 3/5 - 2\Delta$.

From Lemma 4.5, we see that the two largest items will fit in the first bin, and each bin of the FFD $(6/5, L)$ packing except the last will contain at least two items. We shall use a refinement of the classification system of [1] for describing the bins of the FFD $(6/5, L)$ packing and the items they contain.

A bin B of the FFD $(6/5, L)$ packing is called a k-bin if there were k items in the bin at the time the first item was assigned to a later bin. If a k-bin has more than k items at the end, it will be called a fallback k-bin. The items added after the first k are then called fallback items. If no other items are added, it is called a regular k-bin.

LEMMA 4.6. (a) *All k-bins precede all k'-bins for* $k < k'$ *in the* FFD $(6/5, L)$ *packing.*

(b) *All regular k-bins precede all fallback k-bins in the* FFD $(6/5, L)$ *packing.*

Using these lemmas we can deduce considerable information about L and the two packings. If $\Delta \geqq 1/5$, then $s(m) \geqq 1/5 + 1/15 = 4/15$ and, in the light of Lemmas 4.3 and 4.4, each bin in the OPT $(1, L)$ packing contains exactly 3 items. Since $s(b) \leqq 3/5 - 2\Delta$, if B is a bin of the FFD $(6/5, L)$ packing containing only two items, b_1 and b_2, $s(b_1) + s(b_2) \leqq 6/5 - 4\Delta$. Then $s(b_1) + s(b_2) + s(m) \leqq 6/5$ and m would fit in B. Thus each bin of the FFD $(6/5, L)$ packing contains at least three items, and so FFD $(6/5, L) \leqq$ OPT $(1, L)$. We assume, therefore, that $\Delta < 1/15$. If $\Delta > 1/25$, then no bin of the FFD $(6/5, L)$ packing can contain more than 4 items since $5(1/5 + \Delta) > 6/5$. Thus there can be no 5-bins in this case.

We introduce the following notation for the types of bins and items. A regular k-bin and its elements will be called *of type* X_k, except possibly for the last such bin. The items of a regular k-bin must average more than $(1-\Delta)/k$ in size since otherwise the last item m would fit. Hence all X_k items will be greater than $(1-\Delta)/k$ except possibly in the last regular k-bin. If this bin contains an item of size $\leqq (1-\Delta)/k$, then the bin and its items will be called exceptional and both the bin and its items will be called *of type* Z_k.

If the fallback item in a fallback k-bin is of size $> (1-\Delta)/(k+1)$, then the bin contains $k+1$ items each of size $> (1-\Delta)/(k+1)$ and we will classify the bin as type

X_{k+1}. Note that this includes all fallback 4-bins, and if $\Delta > 1/25$, all fallback 3-bins.

For $k = 2$ or 3, a fallback k-bin not covered by the preceding paragraph (and the nonfallback items in it) will be called of *type* Y_k, except possibly the last such bin. If a fallback k-bin contains a nonfallback item $\leqq (6/5)/(k+1)$, the bin and its items will be called exceptional of type V_k. Note that in any fallback k-bin, the sum of the sizes of the nonfallback items is $> k(6/5)/(k+1)$ so the average size must be at least $(6/5)/(k+1)$ and there can be at most one V_k bin for each k.

It is impossible to have two fallback items in a fallback k-bin since each would have size $> 1/5$. Thus the sum of the sizes of the items would be greater than $(k/(k+1))(6/5) + 2(1/5) \geqq 6/5$. The fallback item f in a Y_k or V_k bin is classified according to its size. If $s(f) > (1-\Delta)/4$, f is classified as type X_4, otherwise it is of type X_5.

Next we assign a weight $w(b)$ to each item b. Items of type X_k are assigned weight $(1-\Delta)/k$. Items of type Y_2 are assigned weight $(2/5)(1-\Delta)$ if the fallback item is of type X_5 and weight $(3/8)(1-\Delta)$ if the fallback item is of type X_4. Items of type Y_3 are assigned weight $(4/15)(1-\Delta)$.

Note that for all bins B described so far the weight function w satisfies

(i) $\sum_{b\in B} w(b) = 1-\Delta$,

(ii) $s(b) > w(b)$ for all $b \in B$.

We wish to assign weight to the Z_k and V_k items so that (i) and (ii) will be satisfied for exceptional bins as well. For a Z_k bin, $\sum_{b\in B} s(b) > 1-\Delta$. We define $w(b) = s(b) - (\sum_{b\in B} s(b) - (1-\Delta))/k$, that is $s(b) - w(b)$ is $1/k$ times the excess of the size of the bin over $1-\Delta$. For bins of type V_2 or V_3, we assign a weight to the fallback item f according to its type. If the V_k items in the bin are $b_1, \cdots, b_k$ we assign weight so that $s(b_i) - w(b_i)$ is the same for all i: $w(b_i) = s(b_i) - (\sum_{i=1}^{k} s(b_i) - (1-\Delta) + w(f))/k$. Since $\sum_{i=1}^{k} s(b_i) + w(f) > 1-\Delta$, $s(b_i) > w(b_i)$ in all cases and $\sum_{i=1}^{k} w(b_i) + w(f) = 1-\Delta$.

Two special cases must be noted. The weight of the last item, m, is $(1-\Delta)/4$ if $\Delta > 1/25$ and $1/5$ if $\Delta \leqq 1/25$. In the former case, m is of type X_4. In the latter, it is of no particular type and is treated separately. Also, if in a Y_2 or V_2 bin the regular items are large enough, it is convenient to modify their weights so that each receives a fair share of the excess weight. More precisely, if $s(b_1) + s(b_2) > (4/5)(1-\Delta) + 2\Delta$, then we classify the fallback item as type X_5 and we ensure that $s(b_i) - w(b_i) > \Delta$ for $i = 1, 2$. This can be done by weighting b_1 and b_2 as we did in the V_2 bin.

We summarize much of this information in Table 1, where the last column gives the conditions that Δ must satisfy if an item of the given type is to exist.

TABLE 1

Type	Minimum size	Weight	Restrictions on Δ
X_2	$(1-\Delta)/2$	$(1-\Delta)/2$	None
Y_2	$2/5$	$(3/8 \text{ or } 2/5)(1-\Delta)$	None
X_3	$(1-\Delta)/3$	$(1-\Delta)/3$	None
Y_3	$3/10$	$(4/15)(1-\Delta)$	$\Delta \leqq 1/25$
X_4	$(1-\Delta)/4$	$(1-\Delta)/4$	None
X_5	$1/5+\Delta$	$(1-\Delta)/5$	$\Delta \leqq 1/25$

For exceptional items in a bin Z_i, the average size is at least as great as the minimum size for items of type X_i and for V_i items, the average is at least the minimum

size for Y_i items. Similar statements hold for the average weight of the exceptional items. In the following lemma, we summarize some facts about the difference $s(x)-w(x)$.

LEMMA 4.7. *If x is an item of type V_3, Y_3 or X_5, or if $x=m$ and $\Delta\leqq 1/25$, then $s(x)-w(x)\geqq\Delta$. If x is an item of type V_2 or Y_2, $\Delta\leqq 1/25$, and the fallback item is of type X_4, then $s(x)-w(x)\geqq\Delta$.*

Proof. If x is of type X_5, then

$$s(x)-w(x)\geqq(1/5+\Delta)-(1-\Delta)/5=(6/5)\Delta.$$

If $x=m$, then $s(x)-w(x)=(1/5+\Delta)-1/5=\Delta$.

If x is of type Y_3, then $\Delta\leqq 1/25$ and so

$$s(x)-w(x)\geqq 3/10-(4/15)(1-\Delta)=1/30+4/15\Delta>\Delta.$$

If x is of type V_3, we again must have $\Delta\leqq 1/25$. Let B be the V_3 bin containing x and suppose $B=\{b_1,b_2,b_3,f\}$ with f the fallback item. Then

$$\sum_{i=1}^{3}(s(b_i)-w(b_i))>9/10-(4/5)(1-\Delta)>3\Delta.$$

Since $s(b_i)-w(b_i)$ is the same for all i, $s(x)-w(x)>\Delta$.

If x is of type Y_2, $\Delta\leqq 1/25$, and the fallback item in the bin containing x is of type X_4, then

$$s(x)-w(x)>2/5-(3/8)(1-\Delta)=1/40+(3/8)\Delta\geqq\Delta.$$

If x is of type V_2, $\Delta\leqq 1/25$, and the fallback item f in x's bin $B=\{b_1,b_2,f\}$ is of type X_4, then

$$\sum_{i=1}^{2}(s(b_i)-w(b_i))>4/5-(3/4)(1-\Delta)>2\Delta.$$

Again, $s(x)-w(x)>\Delta$ as in the V_3 case. □

For convenience of notation we extend our weight function from items to sets of items by defining

$$w(B)=\sum_{b\in B}w(b)$$

for $B\subseteq L$.

We complete this section by proving two additional lemmas needed several times in the proof of § 5. The first restricts the possibilities when a bin of the optimal packing contains three items. The second shows that Z_4 items ordinarily cannot be larger than X_4 items, even when the X_4 items are fallback items.

LEMMA 4.8. *Let B^* be a bin of the* OPT$(1,L)$ *packing with $|B^*|=3$, and $w(B^*)>1-\Delta$. Then B^* contains an item of type X_2 or Z_2.*

Proof. Suppose not. Let $B^*=\{b,c,d\}$ with $s(b)\geqq s(c)\geqq s(d)$. Suppose none of the items of B^* is in a fallback 2-bin; then the first bin B after the fallback 2-bins dominates B^* since all items of B^* are available and B contains the three largest available items. Thus at least one item of B^* is in a fallback 2-bin.

Let B be the first fallback 2-bin to contain an item from B^*, and let $B=\{x,y,f\}$ with f the fallback item. Since x and y are the largest items available, we must have $s(x)\geqq s(b)$, $s(y)\geqq s(c)$, and also $b\in B$ since $s(b)\geqq s(c)$. If $s(f)\geqq s(d)$, we would again have B^* dominated by B. Hence d does not fit in B, i.e.,

$$s(z)+s(b)+s(d)>6/5$$

where z is that one of x and y which is not b ($b \neq f$ since $s(b) \geqq s(d) > s(f)$) while

$$s(c)+s(b)+s(d) \leqq 1.$$

Subtracting yields

$$s(z) > s(c)+1/5.$$

We now claim that $s(z)+s(b) \leqq (4/5)(1-\Delta)+2\Delta$. If not then by the definition of w for Y_2 and V_2 bins in the special case where $s(b_1)+s(b) > (4/5)(1-\Delta)+2\Delta$, b must have a size that exceeds its weight by more than Δ; and hence, since $s(B^*) \leqq 1$, we have $w(B^*) < 1-\Delta$ contradicting our assumption.

Thus $s(x)+s(b) \leqq (4/5)(1-\Delta)+2$ and since $s(d) > 6/5-(4/5)(1-\Delta)-2\Delta = 2/5-(6/5)\Delta$. This implies that $s(z) > s(c)+1/5 \geqq s(d)+1/5 > 3/5-(6/5)\Delta$ contradicting Lemma 4.5.

LEMMA 4.9. *If b is an item of type Z_4 and c is an item of type X_4, then either $s(c) \geqq s(b)$ or $s(b) > (1-\Delta)/3$.*

Proof. Since all X_4 items are packed before the Z_4-bin, if $s(b) > s(c)$, it must be the case that c is a fallback item in a fallback 2-bin. If b would not fit and $s(b) \leqq (1-\Delta)/3$, then the nonfallback items, x and y, in this bin satisfy

$$s(x)+s(y) > 6/5-(1-\Delta)/3.$$

If $s(x)+s(y) > (4/5)(1-\Delta)+2\Delta$, then c would have been classified as type X_5 so we must have

$$(4/5)(1-\Delta)+2\Delta > 6/5-(1-\Delta)/3$$

and

$$(13/15)\Delta > 1/15 \quad \text{contradicting} \quad \Delta \leqq 1/15. \quad \square$$

5. Proof of the main result. The principal idea of the proof is to try to show that if B^* is a bin of the OPT $(1, L)$ packing, $w(B^*) \leqq 1-\Delta$. If this were true for all such B^*, we would be done since then

$$w(L) = \sum_{B^*} w(B^*) \leqq \text{OPT}(1, L)(1-\Delta),$$

while

$$w(L) = \sum_{B} w(B) = (\text{FFD}(6/5, L)-1)(1-\Delta)+w(m) > (\text{FFD}(6/5, L)-1)(1-\Delta)$$

where the summation over B is interpreted to mean the sum over the bins of the FFD $(6/5, L)$ packing and the summation over B^* means the sum over the bins of the OPT $(1, L)$ packing. Combining these inequalities, we get FFD $(6/5, L) \leqq$ OPT $(1, L)$ since both FFD $(6/5, L)$ and OPT $(1, L)$ are integers.

Unfortunately it is not true that $w(B^*) \leqq 1-\Delta$ for all B^*. Our proof will show, however, that the potential excess from those bins B^* of weight $> 1-\Delta$ is less than the weight of m. First we show that only bins B^* containing exceptional items can have $w(B^*) > 1-\Delta$. This is enough to enable us to prove the result if $\Delta > 1/25$ (Lemma 5.2). In Lemmas 5.3, 5.4, and 5.5 we examine more closely how the exceptional items can be packed in preparation for the proof of the main result. Roughly the idea is to show that the excess attributable to each exceptional item is at most $\Delta/2$ (except for possibly one bin).

LEMMA 5.1. *If B^* is any bin of the* OPT $(1, L)$ *packing containing no exceptional items, then $w(B^*) \leqq 1-\Delta$.*

Proof. Suppose B^* is a bin of the OPT$(1, L)$ packing which contradicts the lemma. If b is an item of type X_5 or Y_3 or if $b = m$ and $\Delta \leqq 1/25$ then $s(b) - w(b) > \Delta$ and since $s(x) \geqq w(x)$ for all items $x \in L$, we must have $w(B^*) < s(B^*) - \Delta \leqq 1 - \Delta$ if B^* contains an item of type X_5 or Y_3. We assume then that no such item is in B^*.

Since $5s(n) > 1$, B^* must contain no more than 4 items; thus by Lemma 4.3, $|B^*| = 3$ or 4.

Case 1. $|B^*| = 3$.

By Lemma 4.8, if $w(B^*) > 1 - \Delta$, B^* must contain an X_2 item. There can be no other item of size $> (1-\Delta)/3$ else $s(B^*) > (1-\Delta)/2 + (1-\Delta)/3 + (1/5 + \Delta) > 1$. The only possibility is for the other two items to be of type X_4 and $w(B^*) = (1-\Delta)/2 + 2(1-\Delta)/4 = 1 - \Delta$.

Case 2. $|B^*| = 4$.

Now B^* cannot contain an item of type X_2 or Y_2 since then $s(B^*) > 1$. Thus B^* can only contain items of types X_3 and X_4. If all are of type X_4, $w(B^*) = 1 - \Delta$ while if even one is of type X_3, $s(B^*) > (1-\Delta)/3 + 3(1-\Delta)/4 = 13/12 - (13/12)\Delta > 1$. □

Next we show that for a Δ large enough there is no difficulty.

LEMMA 5.2. *If* $\Delta > 1/25$, *then* FFD $(6/5, L) \leqq$ OPT $(1, L)$.

Proof. We assume that L is a minimal counter-example and complete the proof by contradiction.

By our earlier discussion and by Lemma 5.1, we need only show that the excess weight (over $1-\Delta$) in bins containing exceptional items is bounded by $w(m) = (1-\Delta)/4$. We shall in fact show that the excess is bounded by 3Δ which is less than $(1-\Delta)/4$ if $1/25 < \Delta \leqq 1/15$.

For $\Delta > 1/25$, the only exceptional items are of type Z_2, V_2 and Z_3. Let x be either a V_2 item or a Z_3 item of size $> (1-\Delta)/3$. (Note that both V_2 items must exceed $(1-\Delta)/3$ since if the smaller V_2 item were $\leqq (1-\Delta)/3$ then the larger V_2 item is $> 1 - \Delta - (1-\Delta)/3 = 2/3 - 2\Delta/3 > 3/5 - 2\Delta$ contradicting Lemma 4.5.) Let $x \in B^*$ in the OPT $(1, L)$ packing. $|B^*| < 4$ since otherwise

$$S(B^*) > (1-\Delta)/3 + 3(1/5 + \Delta) > 1.$$

But if $|B^*| = 3$, by Lemma 4.8, either $w(B^*) \leqq 1 - \Delta$ or B^* contains an X_2 or Z_2 item. Since

$$(1-\Delta)/2 + (1-\Delta)/3 + 1/5 + \Delta > 1,$$

there cannot be an X_2 item in B^* and we conclude that the only bins of weight $> 1 - \Delta$ are those containing Z_2 items or Z_3 items of size $\leqq (1-\Delta)/3$. Since at least one Z_3 item must have size exceeding $(1-\Delta)/3$, this means there are at most four exceptional bins to contend with—two containing Z_2 items and two containing the Z_3 items of size $\leqq (1-\Delta)/3$.

Let B_1^* and B_2^* be the bins containing the two Z_2 items. Then $|B_1^* \cup B_2^*| = 6$ since there can be no 2-bins and $|B_1^* \cup B_2^*| \geqq 7$ would imply that

$$s(B_1^* \cup B_2^*) > 1 - \Delta + 5(1/5 + \Delta) > 2.$$

Also $B_1^* \cup B_2^*$ cannot contain an item of size $> (1-\Delta)/3$ other than the Z_2 items else

$$2 \geqq s(B_1^* \cup B_2^*) > 1 - \Delta + (1-\Delta)/3 + 3(1/5 + \Delta) = 29/15 + (5/3)\Delta,$$

contradicting $\Delta > 1/25$. If a Z_3 item of size $\leqq (1-\Delta)/3$ is in $B_1^* \cup B_2^*$ then there are at most three bins B^* with $W(B^*) > 1 - \Delta$. If $B_1^* \cup B_2^*$ contains a Y_3 item, then one of the bins (B_1^* or B_2^*) must have weight $\leqq 1 - \Delta$ by Lemma 4.7, and again there are at most three bins B^* with $w(B^*) > 1 - \Delta$. If $B_1^* \cup B_2^*$ contains no Z_3 items and no

Y_3 items, the four items that are not of type Z_2 must be of type X_4 and

$$W(B_1^*)+W(B_2^*)=1-\Delta+4(1-\Delta)/4=2(1-\Delta).$$

In this case there are most two bins which can produce an excess above $1-\Delta$ per bin. In all cases then, the total excess is at most 3Δ. Hence

$$\sum_{b\in L} w(b)=\sum_{B^*\in \text{OPT}(1,2)} (B^*)<(1-\Delta)\text{ OPT}(1,2)+3\Delta$$

and

$$\sum_{b\in L} (b)=\sum_{B\in \text{FFD}(6/5,L)} w(B)=(1-\Delta)(\text{FFD}(6/5,L)-1)+w(m).$$

Combining these yields

$$\text{FFD}(6/5,L)<\text{OPT}(1,L)+(3\Delta-w(m))/(1-\Delta)+1<\text{OPT}(1,L)+1,$$

since $3\Delta<w(m)=(1-\Delta)/4$ for $\Delta<1/15$. Since FFD $(6/5,L)$ and OPT $(1,L)$ are integers, the lemma follows. □

We still have to consider the case where $\Delta\leqq 1/25$. In the next lemma we show that the only exceptional items we need to consider are of type Z_2, Z_3 and Z_4 and that the only other items that can be in a bin B^* satisfying $w(B^*)>1-\Delta$ are of types X_2, X_3, X_4.

LEMMA 5.3. *If B^* is any bin of the* OPT $(1,L)$ *packing satisfying* $w(B^*)>1-\Delta$, *then B^* contains no Y_2, V_2, Y_3, V_3 or X_5 items, and does not contain m.*

Proof. For an item, x, of type Y_3, V_3 or X_5, and for m we have $s(x)-w(x)\geqq\Delta$ by Lemma 4.7. Since $s(y)\geqq w(y)$ for all items, if $x\in B^*$, $w(B^*)<s(B^*)-\Delta\leqq 1-\Delta$.

Suppose $x\in B^*$ and x is of type Y_2 or V_2.

Case 1. $|B^*|=4$.

In this case $s(x)\leqq 1-3(1/5+\Delta)=2/5-3\Delta$. Certainly x cannot be of type Y_2 or be the larger V_2 item, since such items must be larger than $2/5$. If x were the smaller V_2 item, letting y be the larger V_2 item, we would have

$$s(y)+s(x)>4/5+3\Delta,$$

since the next smallest item after x did not fit. Hence by the way weights are assigned to V_2 items, letting f be the fallback item,

$$\begin{aligned} s(x)-w(x)&\geqq[(s(x)+s/y)-(1-\Delta)+w(f)]/2\\ &\geqq[4/5+3\Delta-(1-\Delta)+(1-\Delta)/5]/2>\Delta.\end{aligned}$$

Thus $w(B^*)\leqq s(B^*)-\Delta\leqq 1-\Delta$.

Case 2. $|B^*|=3$.

By Lemma 4.8, B^* must contain an X_2 or Z_2 item, c. Let y be the other Y_2 or V_2 item packed with x in the FFD $(6/5,L)$ packing. Then we must have $s(c)\geqq s(y)$. Hence $s(c)+s(x)>s(y)+s(x)>4/5$ and $s(B^*)$ would be $>4/5+1/5+\Delta>1$ which is impossible. □

Thus there are now at most 9 exceptional items and hence at most 9 bins B^* satisfying $w(B^*)>1-\Delta$. For convenience of notation we define $XS(B^*)=w(B^*)-(1-\Delta)$. We need to show that for the bins $B_1^*, B_2^*, \cdots, B_j^*$, satisfying $XS(B_i^*)>0$ we have

$$\sum_{i=1}^{j} XS(B_i^*)<w(m)=1/5.$$

In the next lemma we bound $XS(B_i^*)$ for some of these B_i^* if B_i^* contains only one exceptional item.

LEMMA 5.4. *If B^* is a bin containing exactly one exceptional item b and $XS(B^*) \geqq \Delta/2$, then the type of b cannot be Z_4 and if it is Z_3, then $s(b) < (1-\Delta)/3$.*

Proof. Suppose B^* is a bin of the optimal packing with $w(B^*) > 1-\Delta/2$, and that B^* contains a single exceptional item b.

Then B^* cannot contain an X_5 item so if $s(b) \geqq (1-\Delta)/3$ we must have $|B^*| < 4$. (Otherwise $s(B^*) \geqq (1-\Delta)/3 + 3(1-\Delta)/4 > 1$.) But if $|B^*| = 3$, by Lemma 4.8 B^* must contain an X_2 item and then $s(B^*) > (1-\Delta)/2 + (1-\Delta)/3 + (1-\Delta)/4 > 1$. Thus we must have $s(b) < (1-\Delta)/3$. If b is of type Z_3, we are done. Suppose b is of type Z_4.

If $|B^*| = 3$, B^* must contain an X_2 item by Lemma 4.8. The remaining item, c, can only be of type X_4. By Lemma 4.8, $s(c) \geqq s(b)$. Now $w(B^*) = (1-\Delta)/2 + (1-\Delta)/4 + w(b)$. Also $1 \geqq s(B^*) > (1-\Delta)/2 + s(c) + s(b) \geqq (1-\Delta)/2 + 2s(b)$. Therefore $s(b) < 1/4 + \Delta/4$, and hence $w(B^*) < 3(1-\Delta)/4 + 1/4 + \Delta/4 = 1 - \Delta/2$ contradicting the hypothesis. Thus b cannot be of type Z_4.

If $|B^*| = 4$ there can be no item of size $> (1-\Delta)/3$ so all other items are of type X_4. By Lemma 4.9 each is at least as big as b and $w(b) < 2(b) \leqq 1/4$. Thus $w(B^*) < 3(1-\Delta)/4 + 1/4 = 1 - 3\Delta/4$, again contradicting the hypothesis on B^*. □

Next we consider what may happen to the Z_2-items.

LEMMA 5.5. *If B_1^* and B_2^* are the bins of the* OPT $(1, L)$ *packing which contain the larger and smaller Z_2 items respectively, and $XS(B_1^*) + XS(B_2^*) \geqq \Delta$, then either $B_1^* \cup B_2^*$ contains at least four exceptional items or one of the Z_3 items with size $< (1-\Delta)/3$.*

Proof. Suppose the lemma is false. Then there is no Z_2-item in $B_1^* \cup B_2^*$ whose size is $< (1-\Delta)/3$, and by Lemma 5.3, $B_1^* \cup B_2^*$ contains only items of type X_2, X_3, X_4, Z_2, Z_3, and Z_4.

If there is any item of size $\geq (1-\Delta)/3$ in $B_1^* \cup B_2^*$ (other than the two Z_2 items), there must be at least two items of size $< (1-\Delta)/4$, since otherwise

$$2 \geqq S(B_1^*) + S(B_2^*) > 1 - \Delta + (1-\Delta)/3 + 2(1-\Delta)/4 + 1/5 + \Delta = 61/30 - (5/6)\Delta \geqq 2.$$

These two small items cannot be of type X_2, X_3, or X_4 and hence must be exceptional, yielding a total of four exceptional items and a contradiction. From this argument we see that there can be no items of size $> (1-\Delta)/3$.

We must have $|B_1^* \cup B_2^*| = 6$ since otherwise $s(B_1^* \cup B_2^*) > 1 - \Delta + 5(1/5 + \Delta) > 2$. The remaining four items must be of type X_4 or Z_4. If there are two Z_4 items, there are four exceptional items while if there are none, $w(B_1^*) + w(B_2^*) = 1 - \Delta + 4(1-\Delta)/4 = 2(1-\Delta)$. Thus there must be exactly one Z_4 item b. By Lemma 4.9, b is no bigger than the X_4 items and hence

$$2 \geqq s(B_1^* \cup B_2^*) > 1 - \Delta + 4s(b).$$

Hence $s(b) < 1/4 + \Delta/4$ and, since $w(b) < s(b)$, $w(B_1^*) + w(B_2^*) < 1 - \Delta + 3(1-\Delta)/4 + 1/4 + \Delta/4$ from which we infer that $XS(B_1^*) + XS(B_2^*) < \Delta/2$, again a contradiction.

We now need to gather the pieces together to complete the proof of the main result.

THEOREM 5.1. FFD $(6/5, L) \leqq$ OPT $(1, L)$.

Proof. As in the lemmas we are assuming that L is a minimal counterexample. From Lemma 5.2 we may assume that $\Delta \leqq 1/25$. From Lemmas 5.1 and 5.2 we know that the only bins B^* for which $XS(B^*) > 0$ are those containing Z_2, Z_3 and Z_4 items. Also the only nonexceptional items in such a bin are of types X_2, X_3 or X_4.

By the definition of type Z_3, there are either one or two Z_3 items of size $(1-\Delta)/3$ or less.

Suppose first that there is only one Z_3 item, b, with $s(b) \leqq (1-\Delta)/3$. Using the notation of Lemma 5.5, if $b \in B_1^* \cup B_2^*$, then all other bins containing Z_3 or Z_4 items either contain two or more exceptional items or have an excess $> \Delta/2$. Thus for the bins B_i^* with $XS(B_i^*) > 0$,

$$\sum_{i=1}^{j} XS(B_i^*) < 2\Delta + 6(\Delta/2) = 5\Delta \leqq 1/5 = w(m)$$

since $\Delta \leqq 1/25$. Since the total excess is less than $w(m)$, the theorem follows as in the proof of Lemma 5.2.

If $b \notin B_1^* \cup B_2^*$, then either $XS(B_1^*) + XS(B_2^*) \leqq \Delta$ or $B_1^* \cup B_2^*$ contains at least four exceptional items. In the former case,

$$\sum_{i=1}^{j} XS(B_i^*) < XS(B_1^*) + XS(B_2^*) + \Delta + 6(\Delta/2) \leqq 5\Delta \leqq w(m),$$

while in the latter case,

$$\sum_{i=1}^{j} XS(B_i^*) < XS(B_1^*) + XS(B_2^*) + \Delta + 4(\Delta/2) \leqq 5\Delta \leqq w(m).$$

Thus in all cases the theorem follows if there is only one Z_3 item of size $\leqq (1-\Delta)/3$.

Now suppose that there are two Z_3 items, b and c, of size $\leqq (1-\Delta)/3$ and let the third Z_3 item be z. Suppose that one of b and c, say b, is in $B_1^* \cup B_2^*$. If there are four exceptional items in $B_1^* \cup B_2^*$, then by the argument used above

$$\sum_{i=1}^{j} XS(B_i^*) < XS(B_1^*) + XS(B_2^*) + \Delta + 4(\Delta/2) \leqq 5\Delta \leqq w(m).$$

Similarly if $B_1^* \cup B_2^*$ contains b and an X_5 item or m

$$\sum_{i=1}^{j} XS(B_i^*) < XS(B_1^*) + XS(B_2^*) + \Delta + 5(\Delta/2) < (9/2)\Delta < w(m).$$

Hence all items in $B_1^* \cup B_2^*$ other than b and the two Z_2 items are of size $> (1-\Delta)/4$ and $s(b) < 2 - (1-\Delta) - 3(1-\Delta)/4 = 1/4 + (7/4)\Delta$. Then $s(z) > 1 - \Delta - s(b) - s(c) > 5/12 - (29/12)\Delta$. Since $(1-\Delta)/2 + 5/12 - (29/12)\Delta + 1/5 + \Delta > 1$, z cannot be in a bin with an X_2 item. Since $B_1^* \cup B_2^*$, by the above reasoning, contains only one item, b, of type Z_3, z cannot be in the same bin with a Z_2 item. Thus, by Lemma 4.8, z must be in a bin B^* with three other items. None of these can be $> (1-\Delta)/4$ in size since $5/12 - (29/12)\Delta + (1-\Delta)/4 + 2(1/5 + \Delta) > 1$ when $\Delta \leqq 1/25$. But then either B^* contains an X_5 item (or m), or B^* contains three other exceptional items. Either of these can be shown by the previous reasoning to result in $\sum_{i=1}^{j} XS(B_i^*) \leqq w(m)$. We conclude that $B_1^* \cup B_2^*$ can contain no Z_3 items of size $\leqq (1-\Delta)/3$.

Suppose now that $b \in B^*$ with $XS(B^*) > 0$ and that B^* contains no other exceptional items. We begin showing that $s(b) < 1/4 + (3/4)\Delta$.

If $|B^*| = 3$, by Lemma 4.8, B^* contains an X_2 item. The remaining item must be of size $> (1-\Delta)/4$ because items smaller than this are either exceptional or of type X_5 (or m). Thus $s(b) < 1 - (1-\Delta)/2 - (1-\Delta)/4 = 1/4 + (3/4)\Delta$. If $|B^*| = 4$, the remaining items must all be of size $> (1-\Delta)/4$ and again $s(b) < 1 - 3(1-\Delta)/4 = 1/4 + (3/4)\Delta$.

But now

$$s(z) > (1-\Delta) - (1-\Delta)/3 - (1/4 + (3/4)\Delta) = 5/12 - (17/12)\Delta,$$

and we can show that this is impossible in the following way. Let B^* be the optimal

bin containing z. If $|B^*|=3$, then by Lemma 4.8, B^* must be B_1^* or B_2^* since B^* cannot contain an X_2 item and still have $s(B^*)\leqq 1$.

But then

$$2\geqq s(B_1^*\cup B_2^*)>1-\Delta+5/12-(17/12)\Delta+3(1/5+\Delta)=121/60+(7/12)\Delta>1.$$

If $|B^*|=4$, $s(B^*)>5/12-(17/12)\Delta+3(1/5+\Delta)>1$. We conclude that b and c each belong to optimal bins containing other exceptional items or with excess $\leqq 0$.

We can summarize our results when b and c are of size $<(1-\Delta)/3$ by saying that $B_1^*\cup B_2^*$ either contains four exceptional items or $XS(B_1^*)+XS(B_2^*)\leqq\Delta$ and that for each other bin B_i^* with $XS(B_i^*)>0$ either B_i^* contains two exceptional items or $XS(B_i^*)\leqq\Delta/2$. Combining these we see that

$$\sum_{i=1}^{j} XS(B_i^*)\leqq(9/2)\Delta<w(m)$$

and the theorem follows. □

REFERENCES

[1] E. G. COFFMAN, JR., M. R. GAREY AND D. S. JOHNSON, *An application of bin-packing to multiprocessor scheduling*, this Journal, 7 (1978) pp. 1–17.

[2] D. K. FRIESEN, *Sensitivity analysis for heuristic algorithms*, Ph.D. thesis, Dept. Computer Science, Univ. Illinois, Urbana-Champaign, 1978.

[3] R. L. GRAHAM, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[4] S. SAHNI, *Algorithms for scheduling independent tasks*, J. Assoc. Comput. Mach., 23 (1976), pp. 116–127.

[5] J. D. ULLMAN, *Complexity of sequencing problems*, Computer and Job/Shop Scheduling Theory, E. G. Coffman, ed., John Wiley, New York, 1976, Chap. 4.

SIAM J. COMPUT.
Vol. 13, No. 1, February 1984

© 1984 Society for Industrial and Applied Mathematics

0097-5397/84/1301-0014 $01.25/0

ON THE COMPLEXITY OF SOME COMMON GEOMETRIC LOCATION PROBLEMS*

NIMROD MEGIDDO† AND KENNETH J. SUPOWIT‡

Abstract. Given n demand points in the plane, the p-center problem is to find p supply points (anywhere in the plane) so as to minimize the maximum distance from a demand point to its respective nearest supply point. The p-median problem is to minimize the sum of distances from demand points to their respective nearest supply points. We prove that the p-center and the p-median problems relative to both the Euclidean and the rectilinear metrics are NP-hard. In fact, we prove that it is NP-hard even to approximate the p-center problems sufficiently closely. The reductions are from 3-satisfiability.

Key words. computational geometry, facility location, p-center problem, p-median problem, NP-hardness, approximation of NP-hard problems

1. Introduction. The goal of the present paper is to prove the NP-hardness of the following common problems in geometric location theory:

P1. *Euclidean p-center problem*: Given a set $X=\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ of points in the plane, find a set $S=\{(z_1, t_1), (z_2, t_2), \cdots, (z_p, t_p)\}$ of p points so as to *minimize*

$$\max_{1\leqq i\leqq n} \min_{1\leqq j\leqq p} \{(x_i-z_j)^2+(y_i-t_j)^2\}.$$

Intuitively, we wish to minimize the radius R such that the points (x_i, y_i) $(i=1, 2, \cdots, n)$ can be enclosed by p circles of radius R.

P2. *Rectilinear p-center problem*: Following the notation of Problem 1, we wish to minimize

$$\max_{1\leqq i\leqq n} \min_{1\leqq i\leqq p} \{|x_i-z_j|+|y_i-t_j|\}.$$

In other words, we wish to minimize the number A such that all the points (x_i, y_i) $(i=1, 2, \cdots, n)$ can be enclosed within p squares of area A, the edges of each square forming angles of 45° with the axes.

P3. *Euclidean p-median problem*: Following the notation of Problem 1, we wish to minimize

$$\sum_{i=1}^{n} \min_{1\leqq j\leqq p} \{\sqrt{(x_i-z_j)^2+(y_i-t_j)^2}\}.$$

P4. *Rectilinear p-median problem*: Here we wish to minimize

$$\sum_{i=1}^{n} \min_{1\leqq j\leqq p} \{|x_i-z_j|+|y_i-t_j|\}.$$

We will prove that in fact it is NP-hard even to approximate P1 to within about 15% and P2 to within 50%.

We note that the analogous problems on the real line (instead of the plane) are both "easy": the p-center problem on the real line is solvable in $O(n \log n)$ time [10], [2], and the p-median problem on the real line is solvable in $O(n^2p)$ time [11].

* Received by the editors May 21, 1982, and in final form January 14, 1983.

† Department of Statistics, Tel Aviv University, Tel Aviv, Israel. The research of this author was partially supported by the National Science Foundation under grant ECS8121741 at Northwestern University, Evanston, Illinois 60201.

‡ Hewlett-Packard Laboratories, Palo Alto, California 94304.

The 1-median problem is also known as the Weber problem. No known algorithm finds the exact coordinates of the median (see [12] for a discussion of this difficulty).

The graphic counterparts of the p-center and the p-median problems are easily shown to be NP-hard [6], [7], using a reduction from Minimum Dominating Set. It is usually more complicated to prove NP-hardness of a geometric problem than of its graphic counterpart (see [4] and [14]).

Shamos [16] conjectures that P1 is NP-hard. Papadimitriou [15] proved NP-hardness of a different Euclidean p-median problem, namely, that in which the points (z_j, t_j) $(j = 1, 2, \cdots, p)$ must be selected from the set X. He mentioned the NP-hardness of both our Problems P1 and P3 as open. Previous versions of the proofs in the present paper were given in [18] and [8].

2. An overview of the proofs. In each of the proofs we establish a reduction from 3-satisfiability [5]. Formally, given a boolean expression

$$E = E_1 \wedge E_2 \wedge \cdots \wedge E_m,$$

where $E_j = x_j \vee y_j \vee z_j$ $(\{x_j, y_j, z_j\} \subseteq \{u_1, \bar{u}_1, u_2, \bar{u}_2, \cdots, u_q, \bar{u}_q\})$, the 3-satisfiability problem is to decide whether there exists a set $S \subseteq \{u_1, \bar{u}_1, u_2, \bar{u}_2, \cdots, u_q, \bar{u}_q\}$ such that

$$S \cap \{x_j, y_j, z_j\} \neq \varnothing \qquad (j = 1, 2, \cdots, m),$$

and

$$|S \cap \{u_i, \bar{u}_i\}| = 1 \qquad (i = 1, 2, \cdots, q).$$

The reduction from 3-satisfiability to a geometric problem will be established as follows. Each variable u_i $(i = 1, 2, \cdots, q)$ will be represented by a "circuit" of objects (e.g. circles, squares, points) in the plane. There will be essentially two different ways to partition the objects of the circuit so that the solution of the location problem is close to optimal. These two different partitions correspond to the choice of truth value for u_i. The clauses E_j $(j = 1, 2, \cdots, m)$ are represented by "clause configurations" which determine how the different circuits meet each other. A clause configuration relates the property that a clause is satisfied to the property that a partition is efficient from the point of view of the location problem.

Circuits must cross each other, without interfering with each other's properties; this requires that we design the "junctions" carefully. A schematic view of the circuits and their relations to the clause configurations is shown in Fig. 1. The details for each of the four problems are given in the succeeding sections.

3. The Euclidean p-center problem. We shall establish the NP-hardness of the Euclidean p-center problem by proving the following problem to be NP-hard:

Circle Covering. Given n unit circles in the plane and an integer $p > 0$, decide whether there exist p points such that each circle contains at least one point (we say that a circle *contains* a point if the point lies on, or in the interior of, the circle).

We now reduce 3-satisfiability to Circle Covering. In the reduction each variable u_i will be represented by a circuit of circles (see Fig. 2) $C^i = \{C^i_0, C^i_1, \cdots, C^i_{r_i}\}$, where $C^i_0 = C^i_{r_i}$, r_i is even and $C^i_k \cap C^i_l \neq \varnothing$ if and only if $|k - l| \leqq 1 \pmod{r_i}$. We say that a set of points Z *covers* a set of circles C if each circle in C contains at least one point in Z. Thus, at least $r_i/2$ points are required to cover C^i. There are essentially two different ways to cover all the circles, namely, either all the points belong to $C^i_k \cap C^i_{k+1}$ for $k = 0, 2, 4, \cdots$, or all belong to $C^i_k \cap C^i_{k+1}$ for $k = 1, 3, 5, \cdots$. In the former case, corresponding to the assignment of "true" to u_i, the points are called *true points*; in

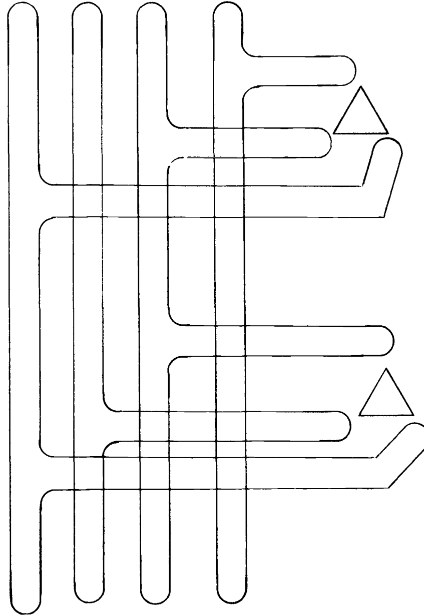


FIG. 1. *A schematic view of the reductions.*

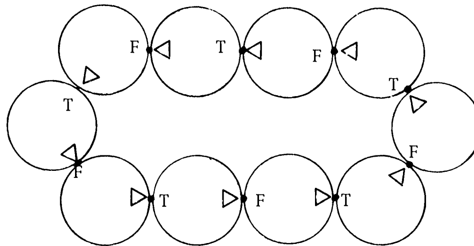


FIG. 2. *A circuit in the reduction for circle covering.*

the latter case, corresponding to the assignment of "false" to u_i, the points are called *false points*. We note that circuits may have to cross each other; we specify later how the "junctions" are designed.

Each clause E_j is represented in the reduction by a configuration of four circles as shown in Fig. 3. Specifically, there is one central circle that intersects the intersection of every two other circles of the configuration. However, the intersection of all four circles is empty. These properties imply that two points are both necessary and sufficient to cover all four circles, namely, one point to cover the central circle and two other circles, and another point to cover the remaining circle. Denote the central circle by D^j and the other three by D^j_x, D^j_y and D^j_z, corresponding to the literals x_j, y_j and z_j, respectively. The circle D^j_x, for example, intersects precisely two circles C^i_k, C^i_{k+1}, where i is such that $x_j \in \{u_i, \bar{u}_i\}$. Moreover, $D^j_x \cap C^i_k \cap C^i_{k+1} \neq \emptyset$. If $x_j = u_i$ then k is even; otherwise $(x_j = \bar{u}_i)$ k is odd. Thus, if the assignment of a truth value to u_i implies that some point in $C^i_k \cap C^i_{k+1}$ is selected, then this point may be selected so as to belong to D^j_x as well. It thus follows that if the overall truth assignment satisfies E_j, then at least one of the circles D^j_x, D^j_y, D^j_z can be covered by a true or a false point. The circle D^j can never be covered by such a point; thus we need precisely one more point per satisfied clause to guarantee that all the corresponding clause configurations are covered.

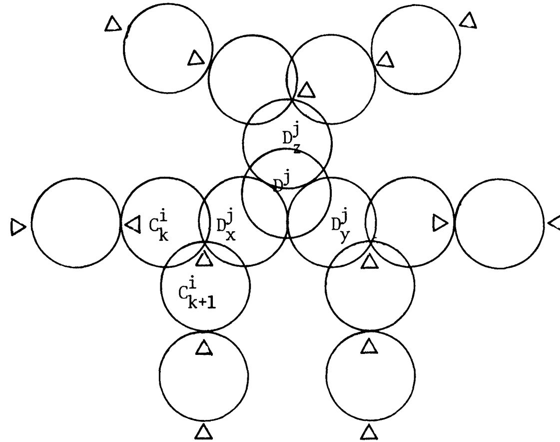


FIG. 3. *A clause configuration in the reduction for circle covering.*

We now discuss junctions. In each junction, a vertical segment of one circuit crosses a horizontal segment of another; the exact structure is shown in Fig. 4. Formally, a junction common to the circuits corresponding to u_i and to u_j has the following characteristics: Suppose that the circuits meet at a circle C_k^i (of the circuit corresponding to u_i) which is identical with a circle C_l^j (of the circuit corresponding to u_j). We insist that both k and l be odd numbers. This ensures that the segments of circuits between consecutive junctions have equal numbers of true points and of false points.

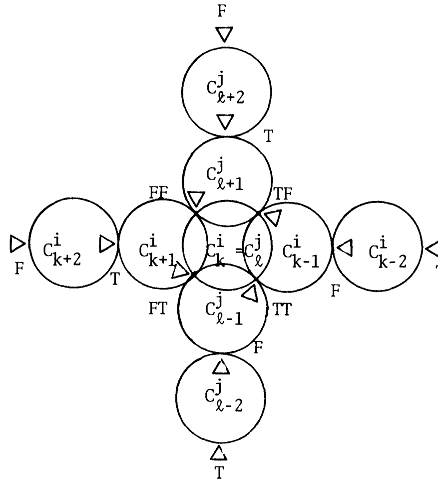


FIG. 4. *A junction in the reduction for circle covering.*

Furthermore, the junction is designed so that the central circle $C_k^i = C_l^j$ intersects the following *nonempty* sets: $C_{k-1}^i \cap C_{l-1}^j$, $C_{k-1}^i \cap C_{l+1}^j$, $C_{k+1}^i \cap C_{l-1}^j$ and $C_{k+1}^i \cap C_{l+1}^j$. Note that the requirement $C_{k-1}^i \cap C_{k+1}^i = C_{l-1}^j \cap C_{l+1}^j = \varnothing$ is satisfied. These facts imply that one point may be saved at each junction. Specifically, a point of the intersection $C_{k+1}^i \cap C_k^i \cap C_{l-1}^j$, for example, is both a false point for the circuit of u_i and a true point for the circuit of u_j. Thus if we assign false to u_i and true to u_j then the points marked with arrows in Fig. 4 constitute a cover for the two circuits that complies with this truth assignment. We denote the number of junctions by J.

Letting

$$p=\sum_{i=1}^{q}\frac{r_i}{2}+m-J,$$

we claim that E is satisfiable if and only if there exists a set Z of p points covering our entire structure. First, assume that E is satisfied by a truth assignment τ. For $i=1,2,\cdots,q$, if $\tau(u_i)=\text{true}$, then include in Z the true points of the circuit for u_i; otherwise include in Z the false points of the circuit for u_i. Since one point is saved per junction, Z so far contains $\sum_{i=1}^{q} r_i/2-J$ points. Since each clause is satisfied, include in Z only one more point per clause in order to cover the central circle as well as the at most two more circles of the clause configuration not covered by a true or a false point. Thus Z contains p points and covers the complete structure.

To prove the converse, let Z be a set of p points that covers the entire structure. We will construct a truth value τ satisfying E. To that end, we will count the number of points available and conclude that the points selected for covering each circuit are either all true points or all false points. Consider a *segment* of a circuit between two consecutive junctions, i.e. a maximal set of circles of the form $\{C^i_{k+2}, C^i_{k+3},\cdots, C^i_{l-2}\}$, where k and l are odd and for each s, $k+2\leqq s\leqq l-2$, C^i_s is not involved in any junction. It follows that the length of each segment is odd, and that segments are pairwise disjoint. Furthermore, there are $2J$ of them, since each junction touches four segments and each segment touches two junctions. The total length of the segments is equal to

$$(\sum r_i-J)-5J=\sum r_i-6J.$$

Since there are $2J$ segments, each of odd length, it follows that

$$\frac{1}{2}(\sum r_i-6J-2J)+2J=\sum\frac{r_i}{2}-2J$$

points are required to cover all of them. It is easy to verify that within each segment, except for at most one point, either all the selected points are true, or all are false (provided that all of the segments are covered with no more than $\sum r_i/2-2J$ points). We are therefore left with only

$$p-\left(\sum\frac{r_i}{2}-2J\right)=m+J$$

more points with which to cover the rest of the circles (i.e. the junctions and the clause configurations), possibly with the aid of the previous points. However, the central circle of each junction and the central circle of each clause configuration are not covered by any point that covers a circle of a segment; hence we need to allocate *precisely one* more point for each junction and for each clause configuration. However, this implies that the aid from the previous points must be organized carefully.

Consider first the junctions. Each junction consists of five circles. The point which covers the central circle can cover at most two more circles; thus two circles per junction need to be covered with the aid of points covering the segments. However, each segment may assist in covering at most one circle of the junction; hence each segment must assist by covering precisely one such circle. Moreover, the assistance at each junction must come from segments of two distinct circuits, since the point that covers the central circle of the junction also covers two more circles belonging to distinct circuits. It finally follows that at each junction the two segments of the

same circuit involved must be covered with the same truth-value type of cover. This implies that the points selected for the cover induce an assignment τ of truth value to the literals.

Now consider any clause configuration. The point that covers the central circle cannot cover all three other circles; hence at least one of the other three is covered with the aid of points covering the segments. The particular way of constructing the clause configuration ensures that its corresponding clause is satisfied by τ. This completes the proof that 3-satisfiability is reducible to Circle Covering. It is easy to verify that the reduction is polynomial.

An interesting consequence of this reduction applies to the NP-hardness of finding an approximate solution to the p-center problem. Suppose that instead of unit circles we draw circles of radius $R \geqq 1$ centered at the centers of the circles used in the reduction. We claim that as long as $R < 2/\sqrt{3}$, the same intersection relations hold among the circles; that is, the minimum number of points required to cover all the circles is independent of R when $1 \leqq R < 2/\sqrt{3}$ (if $R = 2/\sqrt{3}$ then the intersection of all the four circles in a clause configuration is nonempty and hence p points suffice even if E is not satisfiable). It follows that an approximate solution to the p-center problem less than $2/\sqrt{3}$ times the optimal is necessarily an optimal solution. This implies that if $\mathrm{P} \neq \mathrm{NP}$ then no polynomial-time algorithm for the p-center problem always gives a solution less than $2/\sqrt{3} \approx 1.15$ times the optimal; in other words, it is NP-hard to approximate the p-center problem with a relative error of less than about 15%.

4. The rectilinear p-center problem. We now prove the NP-hardness of the rectilinear p-center problem, which is more surprising that the NP-hardness of the Euclidean problem because of the following facts:

(i) The rectilinear 1-center problem is trivially solvable in linear time [3], while the Euclidean problem requires much more sophisticated tools [1], [13], [17], [9].

(ii) The rectilinear problem seems to decompose into two one-dimensional problems. This is true in the case $p = 1$, but turns out to be false in general, in view of our NP-hardness result.

As in the Euclidean case, we will consider a "covering" problem. In the present case, instead of circles, we deal with squares that are all identically oriented; without loss of generality we may assume that their boundaries are parallel to the axes.

Square Covering. Given n unit squares in the plane, each of whose boundaries are parallel to the axes, and an integer $p > 0$, decide whether there exist p points such that each square contains at least one point.

As an aside, we note that our squares have the property of interval intersection, namely, a set of such squares has a nonempty intersection if and only if every two squares in the set intersect. We now define the concept of a square-intersection graph, which extends the notion of an interval-intersection graph. Specifically, an undirected graph is a *square-intersection graph* if there is a one-to-one correspondence between the vertices of the graph and a set of squares in the plane (whose boundaries are parallel to the axes) such that two squares intersect if and only if their corresponding vertices are linked with an edge. Obviously, two such squares intersect if and only if their intervals of projection on the axes intersect. However, the problem of minimum cover by cliques is easily seen to be polynomial on an interval-intersection graph and, as follows from our results, NP-hard on a square-intersection graph.

The proof of NP-hardness for square covering is an adaptation of that for circle covering. First, variable u_i is represented by a circuit $\{S_0^i, S_1^i, \cdots, S_{r_i}^i\}$, of squares as

is shown in Fig. 5; r_i is even. Two squares of a circuit intersect if and only if they are adjacent. A clause configuration is shown in Fig. 6. The intersection of two squares may have positive measure. Each clause E_j is represented by a *single* square S_j that

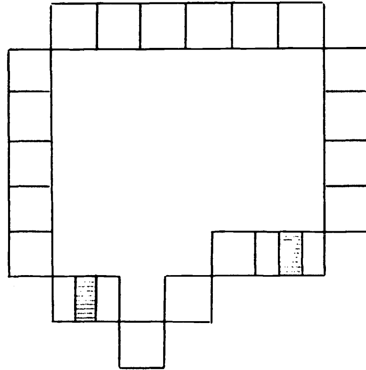


FIG. 5. *A circuit in the reduction for square covering.*

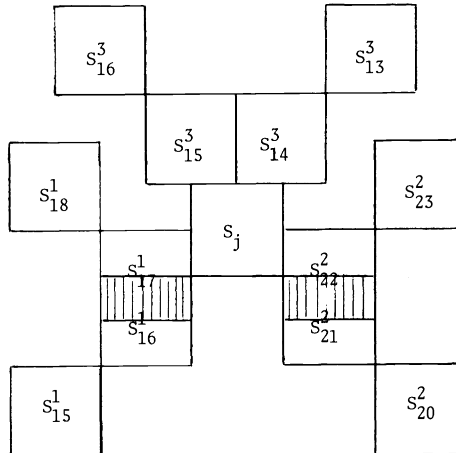


FIG. 6. *A clause configuration in the reduction for square covering.*

touches the three circuits involved (this differs from the proof for circle covering, in which each clause was represented by *four* extra circles; see Fig. 3). A junction is illustrated in Fig. 7. In a junction we simply coalesce a square of one circuit with a square of the other circuit. The coalesced squares must each be odd indexed in its respective circuit so that there will be an *odd* number of squares between consecutive junctions of a circuit. The four corners of the junction square correspond to the four combinations of possible truth values for the corresponding variables. Thus, in order to cover all the circuits, we need $p=\sum r_i/2-J$ points, where J is the number of junctions. If E is satisfiable then p points suffice for covering the clause configurations as well as the circuits if the type of cover in each circuit is chosen appropriately. Conversely, suppose that p points suffice to cover the entire structure. Then consider segments of circuits between consecutive junctions: $\{S^i_{k+2}, S^i_{k+3}, \cdots, S^i_{l-2}\}$, where S^i_k and S^i_l are consecutive junction squares in the circuit for u_i. As in the Euclidean case, we need $\sum r_i/2-2J$ points in order to cover all these segments. We are therefore left with only J more points with which to cover the junctions and the clause configurations. By arguments analogous to those used in the Euclidean case, it follows that the cover with p points induces truth values that satisfy E.

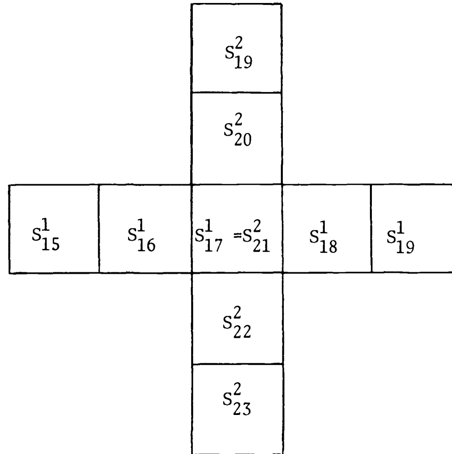


FIG. 7. *A junction in the reduction for square covering.*

An example of the entire structure of squares in the reduction is illustrated in Fig. 8.

As in the Euclidean case, the problem of finding an approximate solution is NP-hard. More precisely, the intersection relations among squares remain the same even if we enlarge their sizes by any factor less than $\frac{3}{2}$. Indeed, if the two unit squares of a circuit which touch a square representing a clause have 50% overlap (see Fig. 6), then by inflating each square by a factor of $\frac{3}{2}$ we obtain nonempty intersections of squares belonging to distinct circuits. This implies that, assuming $P \neq NP$, no polynomial-time algorithm for the rectilinear problem always gives solutions less than $\frac{3}{2}$ times the optimal.

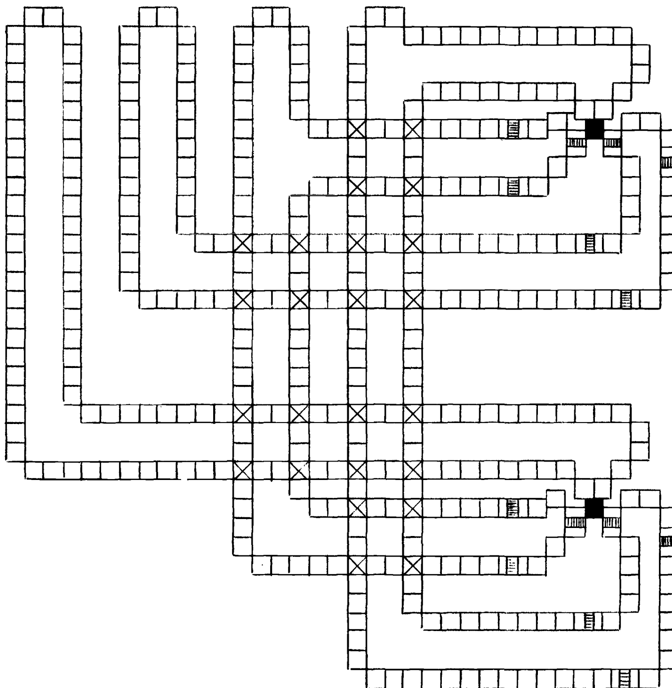


FIG. 8. *The complete structure for a sample square-covering reduction.*

5. The rectilinear p-median problem. We reduce 3-satisfiability to the rectilinear p-median problem. Here clause E_j is represented by a single point P_j^*, while variable u_i is represented by a circuit C^i of points $\{P_0^i, P_1^i, \cdots, P_{r_i}^i\}$, such that $P_0^i = P_{r_i}^i$, and $r_i \equiv 0 \pmod 3$. Denoting the rectilinear distance by d, we also require that if $k \equiv 0 \pmod 3$ then $d(P_k^i, P_{k+1}^i) = 1$, and that otherwise $d(P_k^i, P_{k+1}^i) = b \gg 1$. Moreover, if $|k - l| > 1 \pmod{r_i}$ then $d(P_k^i, P_l^i) > b$. An example of a circuit is shown in Fig. 9.

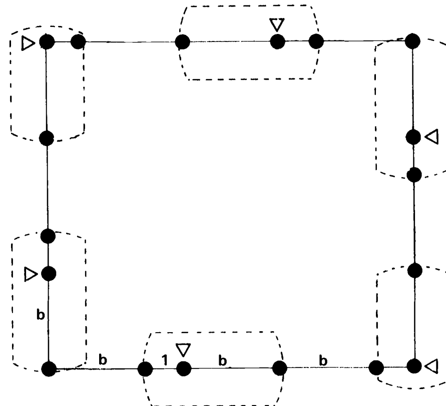


FIG. 9. *A circuit in the reduction for rectilinear p-median. The triangles indicate points P_k^i such that $k \equiv 0 \pmod 3$; hence the dotted lines indicate a true partition.*

Consider the p-median problem on the circuit C^i with $p = r_i/3$. The circuit may be partitioned into triples of points of the form (x, y, z), where $d(x, y) = 1$, $d(y, z) = b$ and $d(x, z) = b + 1$. We claim that the optimal solution of the p-median problem on C^i is obtained by partitioning into such triples and allocating one point per triple. Formally, let $f(S)$ denote the minimum of the 1-median problem on a subset S of a circuit C^i. The p-median problem on C^i may be rephrased as: Partition C^i into p sets $S_1, S_2, \cdots, S_p$ so as to minimize $\sum f(S_j)$. Now consider the function g, where

$$g(1) = 0, \qquad g(2) = 1, \qquad g(k) = (k-2)b + 1 \quad \text{for } k \geqq 3.$$

It can be verified that for every subset S of C^i, $f(S) \geqq g(|S|)$. Considering the problem of minimizing $\sum g(s_j)$ subject to $\sum s_j = 3p$, we observe that the optimal solution is $s_j = 3$, $j = 1, 2, \cdots, p$. Thus, the value $p(b+1)$ is a lower bound on the minimum of $\sum f(S_j)$. Note that there are two different partitions that yield the same value of $p(b+1)$. These are the partitions into triples $P_{k-1}^i, P_k^i, P_{k+1}^i$ where either $k \equiv 0 \pmod 3$ for every k (this is called the *true partition*) or $k \equiv 1 \pmod 3$ for every k (this is called the *false partition*). The solution points coincide with the middle points P_k^i of the triples. Note that every other selection of solution points does not achieve the lower bound of $(b+1)r_i/3$ on a circuit.

We now discuss the clause configurations. For each clause $E_j = x_j \vee y_j \vee z_j$, we allocate one point P_j^* that is situated at a distance of b from three points, one from each circuit related to the clause E_j (see Fig. 10). The point of a circuit that is nearest to P_j^* is chosen according to the relation of the corresponding literal to E_j. For example, if $x_j = u_i$, then the point P_k^i of the circuit of u_i that is nearest to P_j^* is such that $k \equiv 0 \pmod 3$; if $x_j = \bar{u}_i$ then $k \equiv 1 \pmod 3$. The second nearest point is at a distance of $b + 1$ from P_j^* and each other point is at a distance of at least $2b$. Suppose that S is a set of points that contains precisely one "clause point" P_j^* and 0 or more circuit points (we have not yet introduced junctions). Consider the minimum sum of distances $f(S)$ in the 1-median problem on the set S. We claim that $f(S) \geqq h(|S|)$, where

$$h(1) = 0, \quad h(2) = b, \quad h(3) = b + 1, \quad h(k) = (k-2)(b+1) - 1 \quad \text{for } k \geqq 4.$$

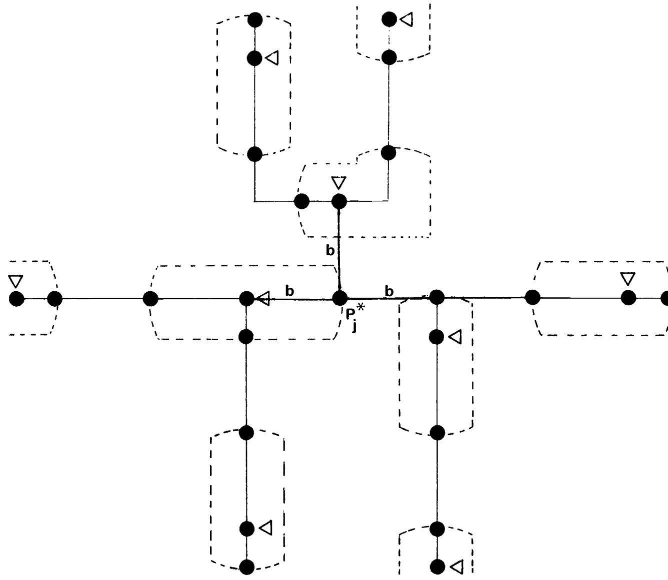


FIG. 10. *A clause configuration in the reduction for rectilinear p-median.*

In particular, the minimum $f(S)$ when $|S|=4$ is attained at sets of the form $S=\{P^i_{l-1}, P^i_l, P^i_{l+1}, P^*_j\}$, where P^i_l is closest to P^*_j, in which case $f(S)=2b+1$ (the solution point coincides with P^i_l). Our strategy is to enforce that if the truth values are chosen so that E_j is satisfied, then there is a solution point at a distance of b from P^*_j.

We now describe the junctions. As in the previous problems, a junction is established when a vertical piece of one circuit meets a horizontal piece of another. A junction of the circuits C^i and C^j is shown in Fig. 11. The junction occurs at a unit

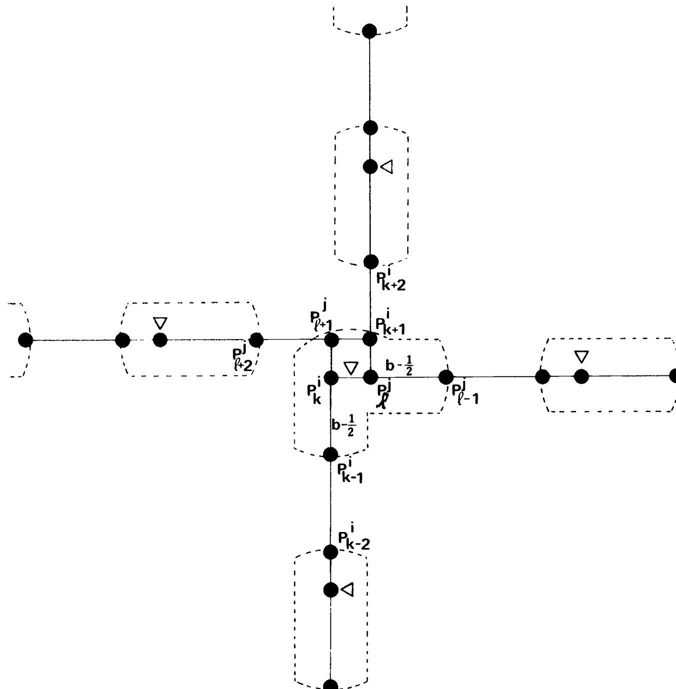


FIG. 11. *A junction in the reduction for rectilinear p-median.*

square whose vertices are the points $P^i_k, P^j_l, P^i_{k+1}, P^j_{l+1}$ such that $k, l \equiv 0 \pmod 3$. The points $P^i_{k-1}, P^j_{l-1}, P^i_{k+2}, P^j_{l+2}$ are each at a distance of $b-\frac{1}{2}$ from some vertex of this square. Note that the sum of rectilinear distances from the vertices of a unit square to any point on the square is equal to 4. Suppose that S is a set of points of our structure that contains at least one junction point. Then $f(S) \geqq e(|S|)$, where

$$e(1)=0, \quad e(2)=1, \quad e(3)=2, \quad e(4)=4, \quad e(5)=b+3.5, \quad e(6)=2b+4,$$
$$e(7)=3b+4.5, \quad e(8)=4b+6.$$

Consider now the p-median problem on the entire structure we have defined, where $p=\sum r_i/3-J$. Note first that there is a set of p solution points such that each of the points P^i_k with $k\equiv 2 \pmod 3$ and each clause point P^*_j has a solution point at a distance not greater than $b+1$ from it, while each other point has a solution point at a distance not greater than 1 from it. Thus, if b is sufficiently large, then an optimal solution for the p-median problem must yield a total distance less than $(\sum r_i/3+m)b+A$, where A is some constant independent of b. By considering the segments between consecutive junctions (as in the previous problems), we find that we must allocate precisely one solution point per junction.

More formally, let

$$T=(b+1)\sum\frac{r_i}{3}+mb+2J.$$

First, assume that E is satisfied by a truth assignment τ; we will construct a solution to our p-median problem of value T. τ induces a set of solution points at locations on the circuits and on one edge per junction square such that every point P^i_k ($k\equiv 2 \pmod 3$) has a solution point at a distance of b from it (see Fig. 11). The same is true for the clause points P^*_j. We thus manage to have a total distance of

$$\left(\sum\frac{r_i}{3}+m\right)b+4J+\left(\sum\frac{r_i}{3}-2J\right)1=(b+1)\sum\frac{r_i}{3}+mb+2J=T.$$

To show the converse, assume that there is a set Z of p points such that

$$\sum_{i=1}^{n}\min_{(z_1,z_2)\in Z}\{|x_i-z_1|+|y_i-z_2|\}=T.$$

Note that the p-median problem amounts to partitioning our set of $n=\sum r_i+m$ points into $p=\sum r_i/3-J$ sets $S_1, S_2, \cdots, S_p$ and then solving a 1-median problem of each S_j. We know that the partition is into sets of the following types:

(i) m sets each containing precisely one clause point and no junction points. If S is of this type then $f(S)\geqq h(|S|)$.

(ii) J sets containing four junction points (one whole junction) and no clause points. If S is of this type then $f(S)\geqq e(|S|)$.

(iii) $p-m-J$ sets containing neither junction points nor clause points. These sets satisfy $f(S)\geqq g(|S|)$.

Now consider the optimization problem of finding a vector $s=(s_1, s_2, \cdots, s_p)$ so as to minimize

$$\sum_{j=1}^{m}h(s_j)+\sum_{j=m+1}^{m+J}e(s_j)+\sum_{j=m+J+1}^{p}g(s_j)$$

subject to $\sum s_j = n$. Consider Table 1, showing "marginal costs". It follows from Table 1 that by letting

$$s_j = \begin{cases} 4, & 1 \leqq j \leqq m, \\ 6, & m+1 \leqq j \leqq m+J, \\ 3, & m+J+1 \leqq j \leqq p, \end{cases}$$

TABLE 1

k	2	3	4	5	6	7
$h(k)-h(k-1)$	b	1	b	$b+1$	$b+1$	$b+1$
$e(k)-e(k-1)$	1	1	2	$b-\frac{1}{2}$	$b+\frac{1}{2}$	$b+\frac{1}{2}$
$g(k)-g(k-1)$	1	b	$b+1$	$b+1$	$b+1$	$b+1$

we obtain an optimal solution to the optimization problem. The value of this solution is

$$\begin{aligned} h(4)m + e(6)J + g(3)(p-m-J) &= m(2b+1) + J(2b+4) + (p-m-J)(b+1) \\ &= \left(\sum \frac{r_i}{3}\right)(b+1) + mb + 2J = T, \end{aligned}$$

which is, of course, a lower bound on the solution of the p-median problem. We have already seen that it is realizable if E is satisfiable. Hence the solution induced by the truth values is optimal. Moreover, if this bound is realizable then necessarily every set of four points of type (i) must have a total distance of $2b+1$, and this is possible if and only if each clause point has a solution point that reflects the fact that the clause is satisfied. The characteristic that a solution with total distance of T must induce truth values is established by considering the segments of circuits between junctions as in the previous problems. In summary, E is satisfiable if and only if the optimal solution of the p-median problem is T.

6. The Euclidean p-median problem. The proof that the Euclidean p-median problem is NP-hard is very similar to that for the rectilinear case. Note that if the angles of the polygon corresponding to a circuit are greater than or equal to 120°, then solution points around the polygon coincide with circuit points (see Fig. 12). A clause configuration is shown in Fig. 13. We note that the 1-median problem on a set of three points of a circuit P^i_k, P^i_{k+1}, P^i_{k+2} (where $k \equiv 0 \pmod 3$ or $k \equiv 1 \pmod 3$) has an optimal value of $b+1$. Moreover, the function g of the preceding section is

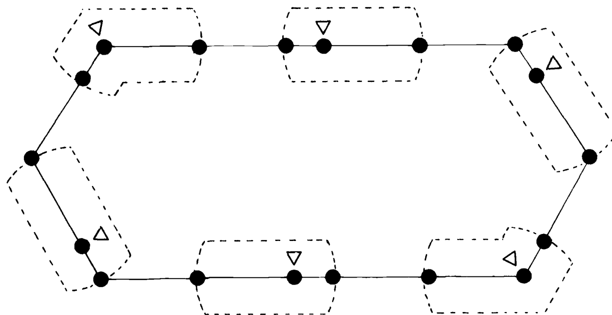


FIG. 12. *A circuit in the reduction for Euclidean p-median. The triangles indicate points P^i_k such that $k \equiv 0 \pmod 3$; hence the dotted lines indicate a true partition.*

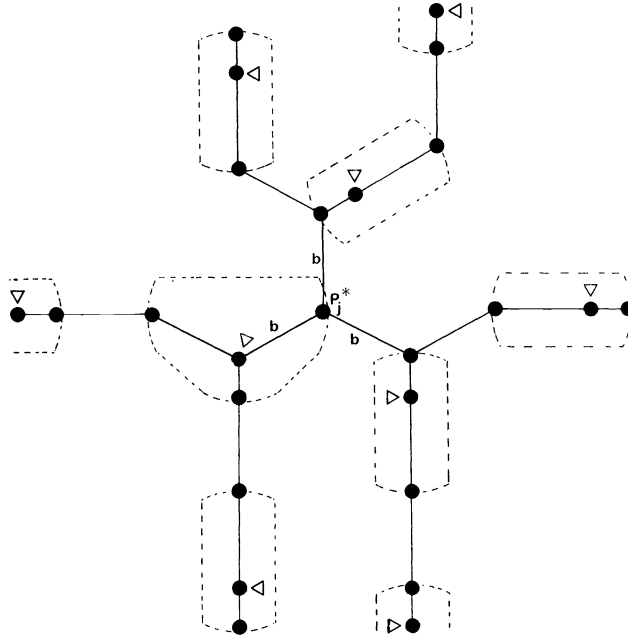


FIG. 13. *A clause configuration in the reduction for Euclidean p-median.*

valid for lower bounding in the present section. The same is true for the function h, i.e. lower bounding for a set S that contains a clause point (there are sets of four points for which $f(S)=h(4)=2b+1$; see Fig. 13).

The situation with junctions (shown in Fig. 14) is a little more delicate in the present section. Points P^i_k, P^i_{k+1}, P^j_l and P^j_{l+1} form a unit square. The distance between a point P^i_{k-1} ($k\equiv 0 \pmod 3$) and a corner of the square equals b. We now need to revise the definition of the lower-bounding function $e(k)$ of the preceding section. We define $e(k)$ to be the minimum of the optimal solutions of 1-median problems on sets S such that $|S|=k$ and S contains four junction points. Then $e(4)=2\sqrt{2}$.

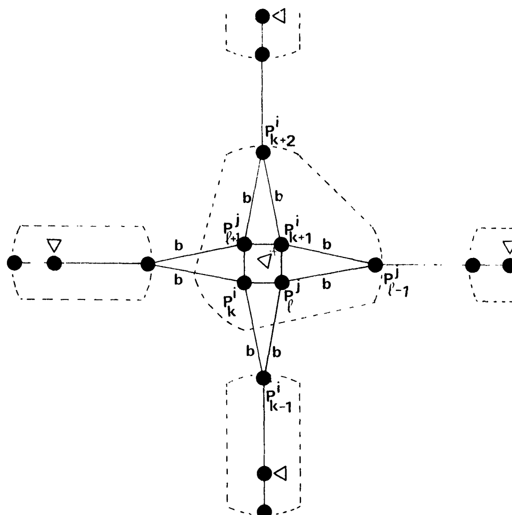


FIG. 14. *A junction in the reduction for Euclidean p-median.*

The problems corresponding to $e(5)$, $e(6)$ and $e(7)$ are shown in Fig. 15(a, b, c). It follows that, for b sufficiently large,

$$b+2\sqrt{2}\leqq e(5)\leqq b+2\sqrt{2}+0.5.$$

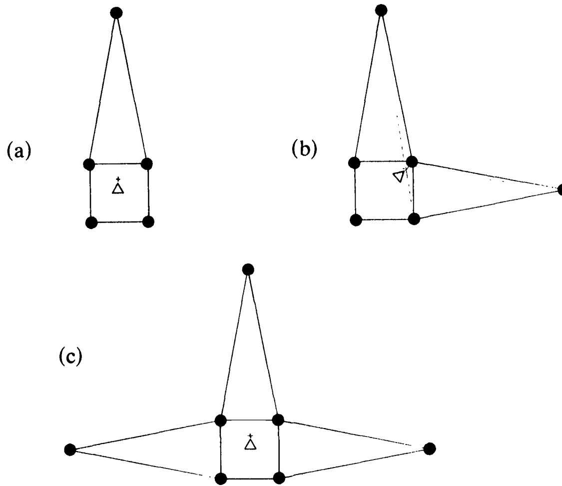


FIG. 15. *Covering points near a junction for Euclidean p-median.*

Furthermore, as $b\to\infty$,

$$e(b)\to 2b+\sqrt{2}+2$$

and

$$e(7)\approx e(5)+2b+1.$$

These facts are sufficient for deducing that an optimal solution to the optimization problem of minimizing

$$\sum_{j=1}^{m} h(s_j)+\sum_{j=m+1}^{m+J} e(s_j)+\sum_{j=m+J+1}^{p} g(s_j)$$

(subject to $\sum_{j=1}^{p} s_j=n$) is the same as in the preceding section, i.e., $s_j=4$ $(j=1,2,\cdots,m)$, $s_j=6$ $(j=m+1,m+2,\cdots,m+J)$, $s_j=3$ $(j=m+J+1,m+J+2,\cdots,p)$. The value of the optimal solution is asymptotically equal to

$$m(2b+1)+J(2b+\sqrt{2}+2)+(p-m-J)(b+1)=\left(\sum\frac{r_i}{3}\right)(b+1)+mb+J\sqrt{2}.$$

The optimal value is realizable in the p-median problem if and only if E is satisfiable.

REFERENCES

[1] J. ELZINGA AND D. W. HEARN, *Geometrical solutions for some minimax location problems*, Transportation Sci., 6 (1972), pp. 379–394.

[2] G. N. FREDERICKSON AND D. B. JOHNSON, *Optimal algorithms for generating quantile information in $X+Y$ and matrices with sorted columns*, in Proc. 13th Annual Conference on Information Science and Systems, Johns Hopkins Univ., Baltimore, MD, 1979, pp. 47–52.

[3] R. L. FRANCIS AND J. A. WHITE, *Facility Layout and Location*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[4] M. R. GAREY, R. L. GRAHAM AND D. S. JOHNSON, *Some* NP-*complete geometric problems*, in Proc. 8th ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1976, pp. 10–22.

[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] O. KARIV AND S. L. HAKIMI, *An algorithmic approach to network location problems, Part* I: *The p-centers*, SIAM J. Appl. Math., 37 (1979), pp. 513–538.

[7] ———, *An algorithmic approach to network location problems, Part* II: *p-medians*, SIAM J. Appl. Math., 37 (1979), pp. 539–560.

[8] N. MEGIDDO, *On some planar location problems*, Dept. of Statistics, Tel Aviv Univ., Tel Aviv, October 1981.

[9] ———, *Linear-time algorithms for linear programming in* R^3 *and related problems*, this Journal, 12 (1983), pp. 759–776.

[10] N. MEGIDDO, A. TAMIR, E. ZEMEL AND R. CHANDRASEKARAN, *An* $O(n \log^2 n)$ *algorithm for the k-th longest path in a tree, with applications to location problems*, this Journal, 10 (1981), pp. 328–337.

[11] N. MEGIDDO, E. ZEMEL AND S. L. HAKIMI, *The maximum coverage location problem*, SIAM J. Alg. Discr. Meth., 4 (1983), pp. 253–261.

[12] Z. A. MELZAK, *A Companion to Concrete Mathematics*, Wiley Interscience, New York, 1973.

[13] K. P. K. NAIR AND R. CHANDRASEKARAN, *Optimal location of a single service center of certain types*, Naval Res. Logist. Quart., 18 (1971), pp. 503–510.

[14] C. H. PAPADIMITRIOU, *The Euclidean traveling salesman problem is* NP-*complete*, Theoret. Comput. Sci., 4 (1977), pp. 237–244.

[15] ———, *Worst-case and probabilistic analysis of a geometric location problem*, this Journal, 10 (1981), pp. 542–557.

[16] M. I. SHAMOS, *Computational Geometry*, Doctoral dissertation, Dept. of Computer Science, Yale Univ., New Haven, CT, 1978.

[17] M. I. SHAMOS AND D. HOEY, *Closest-point problems*, in Proc. 16th Annual IEEE Symposium on the Foundations of Computer Science, Institute of Electrical and Electronics Engineers, New York, 1975, pp. 151–162.

[18] K. J. SUPOWIT, *Topics in computational geometry*, Doctoral dissertation, Dept. of Computer Science, Univ. of Illinois, Urbana–Champaign, IL, April 1981.

SIAM J. COMPUT.
Vol. 13, No. 1, February 1984

© 1984 Society for Industrial and Applied Mathematics
0097-5397/84/1301-0015 $01.25/0

MINIMAL-COST BROTHER TREES*

THOMAS OTTMANN†, D. STOTT PARKER‡, ARNOLD L. ROSENBERG†,
HANS W. SIX†, AND DERICK WOOD⊥

Abstract. We investigate three cost measures for the recently introduced brother search trees. In particular we characterize node visit optimal, comparison-cost optimal and space-cost optimal 1-2 brother trees and present linear-time algorithms to construct optimal 1-2 brother trees for each cost measure. Furthermore we also consider, briefly, these cost measures for brother leaf search trees.

Key words. brother trees, node visit cost, comparison cost, space cost, optimal cost

1. Introduction. In many data processing situations we are given a large set of keys as an initial configuration. Then the set is dynamically altered by inserting new keys and deleting unwanted keys. Furthermore, member operations and other queries which do not alter the set of keys are also posed. Queries of this latter type may far exceed the others. Data structures for which an arbitrary sequence of member, insert, and delete operations can be carried out efficiently are usually called dictionaries; see Aho, Hopcroft and Ullman [1974]. It is well known that dictionaries can be implemented in such a way that all three dictionary operations can be performed in time $O(\log n)$. Various balanced tree schemes are known which may be used for this task. Among them are the AVL trees of Adelson-Velskii and Landis [1962], the 2-3 trees of Hopcroft (see Aho, Hopcroft and Ullman [1974]), the brother (leaf-search) trees of Ottmann and Six [1976], and Ottmann, Six and Wood [1978] and the 1-2 brother trees of Ottmann and Wood [1981]. The insertion procedure for a balanced-tree scheme can also be used to handle the initialization phase in the above mentioned data processing situation: By iteratively inserting the N keys of the given initial set, beginning with the empty tree, we obtain an initial tree in time $O(N \log N)$. However, this iterative insertion method does not utilize the often valid assumption that the initial set of N keys is given in lexicographic order. Therefore, a natural question arises, namely, how to construct efficiently a balanced tree which is optimal in some sense, when given a set of keys in lexicographic order. This problem has been solved for the class of 2-3 trees by Miller, Pippenger, Rosenberg and Snyder [1979] and by Rosenberg and Snyder [1978]. The present paper addresses this question for 1-2 brother trees and brother leaf-search trees. We characterize those trees which are optimal with respect to three different cost measures, the expected number of node-visits per

* Received by the editors January 30, 1979, and in final revised form March 17, 1983. This research was partially supported by Natural Sciences and Engineering Research Council of Canada grants A-5692 and A-7700, a grant from the Deutsche Forschungsgemeinschaft (DFG) and National Science Foundation grants IST 80-12419 and MCS 81-16522. This paper was typeset using software developed at the University of Waterloo. Final copy was produced on Autologics APS-µ5 typesetter.

† Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, D-7500 Karlsruhe, West Germany.

‡ Computer Science Department, University of California, Los Angeles, California 90024.

§ Computer Science Department, Duke University, Durham, North Carolina 27706.

⊥ Data Structuring Group, Computer Science Department, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada.

access, the expected number of key comparisons per access, and the space cost. Linear-time algorithms to construct optimal trees are also designed.

2. Brother trees, 1-2 brother trees and their cost. A *brother tree* is a rooted, oriented tree each of whose nonleaf nodes has either one or two sons. Each unary node must have a binary brother. All root-to-leaf paths have the same length.

There are two basic ways of representing a set of keys as the (values of the) nodes of a tree. Either the keys are stored at internal nodes and the leaves are not used, or the keys are stored at the leaves while the internal nodes contain separating or routing values to direct queries to the correct leaf. Which is the appropriate form of representation depends on the particular application. If, for example, a sequence of insert, delete, and min operations has to be performed (that is if we want to implement a priority queue by using brother trees) storing the keys at the leaves and assigning to each internal node the minimum value stored in the subtree of that node is appropriate.

The two ways of storing sets of keys lead to the class of 1-2 brother trees on the one hand and to the class of brother (leaf-search) trees on the other: In a 1-2 *brother tree* a binary node has one key and both unary nodes and leaves have no keys. All keys resident in a binary node's left subtree are strictly less than the key resident at the node; all keys resident in a binary node's right subtree are strictly greater than the key resident at the node.

In a *brother leaf-search tree* the leaves contain the keys ordered from left to right in increasing order. The internal nodes contain separating or routing information which enables us to retrieve the keys stored at the leaves. Various assignments of such separators, that is routing schemes, (cf. Kwong and Wood [1980]), are possible: we may, for example, assign to each internal node the maximum value of its sons, or we may assign to each internal node the rightmost key in the left subtree of that node.

We simply speak of *brother trees* if we are only interested in structural properties and wish to disregard the method of representing keys.

We will make frequent use of some basic notions of trees. The *depth* of a node p in a tree is its distance from the root, that is, the number of edges on the path from the root to p. The *height* of a node p is the largest distance from p to a leaf in the subtree of the tree with root p. The *height of a tree* is the height of its root.

The root of a tree is said to be at level 0; the sons of a node at level l are said to be at level $l+1$.

Given a binary tree T of height h, with leaves on level h only. The *profile* $\pi(T)$ is the integer sequence $\pi(T)=\nu_0,\dots,\nu_h$ where ν_i is the number of nodes at level i in T. The *detailed profile* $\Delta(T)$ is the sequence of pairs

$$\Delta(T) = <\omega_0,\beta_0>,\dots,<\omega_h,\beta_h>$$

where each ω_j (resp., β_j) denotes the number of unary (resp., binary) nodes at level j in T. Here we are setting $\beta_h=\nu_h=N+1$, and $\omega_h=0$ by convention, where $N+1$ is the number of leaves of the tree.

From these definitions we immediately obtain:

$$\nu_0 = 1, \tag{2.1}$$

$$\nu_h = \beta_h = 1 + \sum_{i=0}^{h-1} \beta_i = N+1, \text{ the number of leaves of } T, \tag{2.2}$$

$$\nu_i = \omega_i + \beta_i, \quad 0 \leq i \leq h, \tag{2.3}$$

$$\nu_{i+1} = \omega_i + 2\beta_i, \quad 0 \leq i < h. \tag{2.4}$$

PROPOSITION. *Let* $\Delta = <\omega_0, \beta_0>, \ldots, <\omega_h, \beta_h>$ *be the detailed profile of a binary tree of height h with leaves on level h only. Then* Δ *is the detailed profile of a brother tree if and only if*

$$\beta_0 = 1, \tag{2.5}$$

$$\beta_i \geq \omega_{i+1} \quad (0 \leq i < h). \tag{2.6}$$

Proof. It is clear that the detailed profile of a brother tree fulfills conditions (2.5) and (2.6). It remains to show that there exists a brother tree having detailed profile Δ, where Δ is the detailed profile of a binary tree of height h with leaves on level h only which fulfills the conditions (2.5) and (2.6). Using (2.4) we obtain for all i, $0 \leq i < h$:

$$\beta_{i+1} + \omega_{i+1} = 2\beta_i + \omega_i \geq 2 \cdot \omega_{i+1} + \omega_i, \quad \text{by (2.6).}$$

Thus

$$\beta_{i+1} \geq \omega_{i+1} + \omega_i \geq \omega_{i+1}.$$

From (2.5) we obtain $\beta_0 \geq \omega_0$. Summarizing we have

$$\beta_i \geq \omega_i, \quad 0 \leq i \leq h. \tag{2.7}$$

Thus starting with level h we can now associate to every unary node a binary one on the same level. Each of these pairs can be provided with a binary father on the next higher level because of condition (2.6). Thus we ultimately obtain a brother tree having detailed profile Δ. □

In analogy with the related cost measures for 2,3-trees, (see Miller, Pippenger, Rosenberg and Snyder [1979], and Rosenberg and Snyder [1978]) we define the node-visit cost NVCOST, the comparison-cost COMPCOST, and the space-cost SPACECOST of 1-2 brother trees: Let T be a 1-2 brother tree of height h with $\pi(T) = \nu_0, \ldots, \nu_h$ and $\Delta(T) = <\omega_0, \beta_0>, \ldots, <\omega_h, \beta_h>$.

$$\begin{aligned} \text{NVCOST}(T) &= \sum_{i=0}^{h-1} (i+1) \cdot \beta_i \\ &= h \cdot \nu_h - \sum_{i=0}^{h-1} \nu_i \quad \text{(by (2.3), (2.4)).} \end{aligned} \tag{2.8}$$

This definition implies that 1-2 brother trees having the same (detailed) profile have the same NVCOST.

The comparison-cost is (the number of keys times) the average number of key-comparisons needed to access a key in T. Since no key-comparison is necessary to access the only son of a unary node, we define

$$\mathrm{COMPCOST}(T) = \sum_{p \text{ binary}} \mathrm{bindist}(p) \tag{2.9}$$

where bindist(p) denotes the number of binary nodes on the path from the root to p. Finally, let

$$\begin{aligned}\mathrm{SPACECOST}(T) &= \sum_{i=0}^{h-1} \nu_i \\ &= \text{number of internal nodes of } T.\end{aligned} \tag{2.10}$$

Observe that for each 1-2 brother tree T we have:

$$\begin{aligned}\mathrm{NVCOST}(T) + \mathrm{SPACECOST}(T) &= h \cdot \nu_h \\ &= \mathrm{height}(T) \cdot \text{number of leaves of } T.\end{aligned}$$

A tree is called *optimal* with respect to a certain cost measure if it has the minimum cost among all trees with the same number of keys.

In §3, 4 and 5 we will characterize the 1-2 brother trees which are optimal with respect to the three different cost measures introduced above. We will use the following abbreviations throughout:

NVO for *node-visit optimal*,
CCO for *comparison-cost optimal*, and
SCO for *space-cost optimal*.

Since 1-2 brother trees are "expanded" height-balanced or AVL trees (see Ottmann, Six and Wood [1979]), it should be clear that the placement of the unary nodes plays a central role in determining optimality. We will show that to a certain extent there is a duality between NVO and SCO trees, in that their only differentiating feature is the number of unary nodes. On the one hand an NVO tree has as many unary nodes as possible, so they must necessarily appear close to the leaf level of the tree. On the other hand a SCO tree has as few unary nodes as possible, so they must appear close to the root. "NVO" is to 1-2 brother trees what "bushy" is to 2-3 trees, while "SCO" is similar to the notion of "scrawny" for 2-3 trees (see Miller, Pippenger, Rosenberg and Snyder [1979]). This analogy is not complete, since the "scrawny" 1-2 brother trees or Fibonacci trees of Ottmann and Wood [1981] are not SCO. However, as we shall demonstrate "SCO" is "scrawny" under the constraint that the trees' height be minimal. We will also show that CCO trees are characterized as those 1-2 brother trees, which have at most one unary node on each root-to-leaf path.

In §6 we compare the three cost measures for 1-2 brother trees.

For brother leaf-search trees the above-introduced cost measures have to be modified slightly in order to take account of the fact that keys are stored only at the leaves while internal nodes contain separators. In §7 we briefly discuss appropriate

modifications and their difficulties, and address the question of characterizing optimal brother leaf-search trees. Finally in §8 we mention some open problems, and give a brief history of this paper.

We conclude this section with an example:

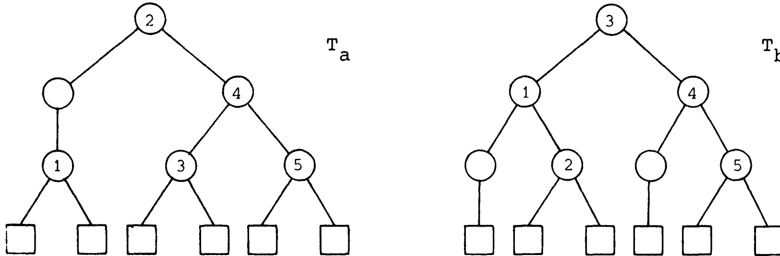


FIG. 1

The trees T_a and T_b of Fig. 1 are both 1-2 brother trees representing the set of keys $\{1,...,5\}$. Their detailed profiles are:

$$\Delta(T_a) = <0,1>, <1,1>, <0,3>, <0,6>,$$
$$\Delta(T_b) = <0,1>, <0,2>, <2,2>, <0,6>.$$

Their costs are:

	T_a	T_b
NVCOST	*12*	*11*
COMPCOST	*11*	*11*
SPACECOST	6	7

In fact, T_a is SCO but not NVO, and T_b is NVO but not SCO. Both trees are CCO. This example already shows that node-visit cost optimality and space cost optimality are independent of each other.

3. Node-visit-optimal 1-2 brother trees. We characterize NVO 1-2 brother trees and design a linear-time algorithm to construct them. The first result gives a necessary condition for 1-2 brother trees to be NVO, namely, height-minimality.

LEMMA 3.1. *Let T and T' be 1-2 brother trees both having the same number of leaves (that is containing the same number of keys). Let T(respectively, T') be of height h (respectively, h'). If $h < h'$ then* NVCOST(T) < NVCOST(T') .

Proof. Let $\pi(T) = \nu_0,...,\nu_h$, $\pi(T') = \nu'_0,...,\nu'_{h'}$, where $\nu_h = \nu'_{h'}$, $h < h'$. Then by (2.8)

$$\begin{aligned}\mathrm{NVCOST}(T') - \mathrm{NVCOST}(T) &= h'\nu'_{h'} - \sum_{i=0}^{h'-1} \nu'_i - (h\cdot\nu_h - \sum_{i=0}^{h-1}\nu_i)\\ &= \nu'_{h'}(h'-h) - \sum_{i=0}^{h'-1}\nu'_i + \sum_{i=0}^{h-1}\nu_i, \quad \text{since } \nu_h = \nu'_{h'}\\ &\geq \nu'_{h'} + \sum_{i=0}^{h-1}\nu_i - \sum_{i=0}^{h'-1}\nu'_i, \quad \text{since } h' > h.\end{aligned}$$

By (2.2) both trees have the same number of binary nodes. This implies

$$\sum_{i=0}^{h-1}\nu_i - \sum_{i=0}^{h'-1}\nu'_i \;\geq\; \sum_{i=0}^{h-1}\beta_i - \sum_{i=0}^{h'-1}\nu'_i \;=\; -\sum_{i=0}^{h'-1}\omega'_i.$$

Furthermore, the number of unary nodes of a 1-2 brother tree is always strictly less than the number of binary nodes in the tree, since each unary node must have a binary father and brother. This implies

$$-\sum_{i=0}^{h'-1}\omega'_i \;>\; -\sum_{i=0}^{h'-1}\beta'_i \;=\; -(\nu'_{h'}-1).$$

Therefore we have $\mathrm{NVCOST}(T') - \mathrm{NVCOST}(T) \geq 1$. □

The trees of Fig. 1a, b illustrate that height minimality is not sufficient for NV-optimality of a 1-2 brother tree.

There does not appear to be a 1-2 brother tree correspondent to the 2, 3-tree notion of "dense profile" as in Miller, Pippenger, Rosenberg and Snyder [1979], with which to characterize NVO 1-2 brother trees. We demonstrate that the number of nodes at each level of a NVO tree is not completely specified by the number of nodes on the level below. Thus one cannot hope for a simple modification of the notion of denseness for 2-3 trees, namely

$$\nu_l = \min(3^l, \lfloor \nu_{l+1}/2 \rfloor) \text{ for } 1 \leq l \leq h-1 ,$$

to something like

$$\nu_l = \min(2^l, \lfloor 2/3\,\nu_{l+1} \rfloor),\ 1 \leq l \leq h-1,$$

for 1-2 brother trees. We argue as follows.

We consider for example 1-2 brother trees with $N+1 = 18\cdot 2^{h-5}$ leaves of height h. The following two detailed profiles are both profiles of equally costly 1-2 brother trees:

$$\begin{aligned}\Delta &= <0,2^0>,\ldots,<0,2^{h-4}>,<2^{h-5},3\cdot 2^{h-5}>,<3\cdot 2^{h-5},4\cdot 2^{h-5}>,\\ &\quad <4\cdot 2^{h-5},7\cdot 2^{h-5}>,<0,N+1>,\\ \Delta' &= <0,2^0>,\ldots,<0,2^{h-4}>,<2^{h-4},2^{h-4}>,<0,3\cdot 2^{h-4}>,\\ &\quad <3\cdot 2^{h-4},3\cdot 2^{h-4}>,<0,N+1>.\end{aligned}$$

Lemma 3.2 given below will show that both are profiles of NVO 1-2 brother trees. However, the numbers of nodes ν_{h-1} and ν'_{h-1} on level $h-1$ are different:

$$\nu_{h-1} = 4\cdot 2^{h-5} + 7\cdot 2^{h-5} = \frac{11}{18}(N+1),$$

$$\nu'_{h-1} = 3\cdot 2^{h-4} + 3\cdot 2^{h-4} = \frac{12}{18}(N+1).$$

This shows that for NVO 1-2 brother trees the number of nodes on level $h-1$ is not uniquely determined by the number of nodes $\nu_h = N+1$ on level h. Similar arguments carry over to level $h-2$ and to other numbers of leaves.

In order to characterize profiles of NVO 1-2 brother trees each three adjacent levels have to be considered:

LEMMA 3.2. *Let* $\Delta = <\omega_0, \beta_0>, \ldots, <\omega_h, \beta_h>$ *be the detailed profile of a NVO 1-2 brother tree of height* h. *Then for each* k, $3 \le k \le h$, *at least one of the following conditions must hold:*

$$\omega_{k-2} = 0, \tag{3.1}$$

$$\beta_{k-2} = \omega_{k-1}, \tag{3.2}$$

$$\beta_{k-1} = \omega_k. \tag{3.3}$$

Proof. Assume the contrary, that is there exists a NVO 1-2 brother tree of height h and a k, $3 \le k \le h$, such that none of the conditions (3.1), (3.2), (3.3) is met. We show that there exists another 1-2 brother tree representing the same number of keys with the detailed profile

$$\Delta' = <\omega_0, \beta_0>, \ldots, <\omega_{k-2}-1, \beta_{k-2}+1>, <\omega_{k-1}+2, \beta_{k-1}-1>, <\omega_k, \beta_k>, \ldots, <\omega_h, \beta_h>$$

which has better NV-cost. First note that $\omega_{k-2} - 1 \ge 0$ because (3.1) does not hold by assumption. Furthermore, $\beta_{k-1} \ge \omega_k \ge 0$, because (2.6) holds for the given detailed profile Δ. Thus the assumption $\beta_{k-1} \ne \omega_k$ implies $\beta_{k-1} > \omega_k$, hence $\beta_{k-1} - 1 \ge 0$. Therefore we know that all numbers occurring in Δ' are nonnegative. It remains to show only that Δ' obeys the constraints (2.4), (2.5), (2.6). In order to show (2.4) it suffices to prove

$$2\beta_{k-3} + \omega_{k-3} = (\beta_{k-2}+1) + (\omega_{k-2}-1),$$

$$2(\beta_{k-2}+1) + (\omega_{k-2}-1) = (\beta_{k-1}-1) + (\omega_{k-1}+2),$$

and

$$2(\beta_{k-1}-1) + (\omega_{k-1}+2) = \beta_k + \omega_k.$$

These equations are obtained trivially from the fact that condition (2.4) must hold for the given detailed profile Δ. Similarly, condition (2.5) must hold for Δ' because it holds for Δ.

Finally, proving (2.6) for Δ' reduces to showing the inequality $\beta_{k-2}+1 \ge \omega_{k-1}+2$. It can be obtained from the assumption $\beta_{k-2} \ne \omega_{k-1}$ and from the fact that (2.6) holds for Δ. □

COROLLARY 3.3. *Let* $\Delta = <\omega_0, \beta_0>, \ldots, <\omega_h, \beta_h>$ *be the detailed profile of a NVO 1-2 brother tree. If there exists an* l, $0 < l < h$, *such that* $\beta_l > \omega_{l+1}$ *and* $\beta_{l-1} > \omega_l$ *then for all* l', $0 \le l' \le l-1$, $\omega_{l'} = 0$ *must hold.*

Proof. Let us assume that $\beta_l > \omega_{l+1}$ and $\beta_{l-1} > \omega_l$ for an l, $0 < l < h$.

We will show that $\omega_{l'}=0$ must hold for all l', $0 \leq l' \leq l-1$, by induction on l'. First the application of Lemma 3.2 for $k=l+1$ yields $\omega_{l-1}=0$ or $\beta_{l-1}=\omega_l$ or $\beta_l=\omega_{l+1}$. Hence, our assumptions imply that $\omega_{l-1}=0$ must hold. Next assume that $\omega_k=0$ for all k, $l' \leq k \leq l-1$. Using condition (2.6) this in particular implies that $\beta_{l'-1}>\omega_{l'}=0$ and $\beta_{l'}>\omega_{l'+1}=0$ must hold. Again applying Lemma 3.2 for $k=l'-1$ yields $\omega_{l'-1}=0$. □

This corollary in particular implies that a NVO 1-2 brother tree must be complete binary up to level l (that is $\omega_0=\omega_1=\cdots=\omega_l=0$), if $\omega_l=\omega_{l-1}=0$.

PROPOSITION 3.4. *Let T be a 1-2 brother tree with $N+1$ leaves and detailed profile $\Delta=<\omega_0,\beta_0>,\ldots,<\omega_h,\beta_h>$. If $\beta_{h-2}=\omega_{h-1}$ then $N+1=3\beta_{h-2}+2\omega_{h-2}$.*

Proof. We have $N+1=2\beta_{h-1}+\omega_{h-1}=2\beta_{h-1}+\beta_{h-2}$. However

$$2\beta_{h-2}+\omega_{h-2} = \beta_{h-1}+\omega_{h-1} = \beta_{h-1}+\beta_{h-2},$$

so

$$\beta_{h-2}+\omega_{h-2} = \beta_{h-1}.$$

Thus

$$N+1 = 2(\beta_{h-2}+\omega_{h-2})+\beta_{h-2} = 3\beta_{h-2}+2\omega_{h-2}. \quad \square$$

The next theorem constitutes one part of our result characterizing NVO 1-2 brother trees.

THEOREM 3.5. *Let T be a 1-2 brother tree with N keys and minimal possible height $h=\lceil \log_2(N+1) \rceil$, where $3\cdot 2^{h-2} \leq N+1 \leq 2^h$. Then T is NVO if and only if its detailed profile $\Delta=<\omega_0,\beta_0>,\ldots,<\omega_h,\beta_h>$ satisfies*

(i) $\beta_k=2^k$, $\omega_k=0$ *for* $0 \leq k \leq h-2$,

(ii) $\beta_{h-1}=(N+1)-2^{h-1}$, *and* $\omega_{h-1}=2^h-(N+1)$.

Proof. Because NVCOST is a function of the (detailed) profile of 1-2 brother trees it suffices to show the "only if" part. Let T be a NVO 1-2 brother tree with detailed profile Δ. We claim first that $\omega_{h-2}=0$. If not, Lemma 3.2 tells us that $\beta_{h-2}=\omega_{h-1}$ (since otherwise $\beta_{h-1}=\omega_h=0$, which is impossible).

By Proposition 3.4 we obtain

$$N+1 = 3\beta_{h-2}+2\omega_{h-2} \leq 3\cdot(2^{h-2}-\omega_{h-2})+2\omega_{h-2} < 3\cdot 2^{h-2}$$

a contradiction.

We claim second that $\omega_{h-3}=0$. If not we again find by Lemma 3.2 that $\beta_{h-2}=\omega_{h-1}$, since we now know that $\omega_{h-2}=0$. But then,

$$\begin{aligned} N+1 &= 3\beta_{h-2} \\ &= 2(2\beta_{h-3}+\omega_{h-3}) \\ &\leq 3(2(2^{h-3}-\omega_{h-3})+\omega_{h-3}) \\ &< 3\cdot 2^{h-2}, \quad \text{also a contradiction.} \end{aligned}$$

From $\omega_{h-2}=0$ and $\omega_{h-3}=0$ we obtain $\beta_{h-3}>\omega_{h-2}$ and $\beta_{h-4}>\omega_{h-3}$. Now

Corollary 3.3 tells us that $\omega_l = 0$ for all $l \leq h-4$. The theorem is thus complete except for β_{h-1} and ω_{h-1}. Since $\beta_{h-2} = 2^{h-2}$ we find

$$\beta_{h-1} + \omega_{h-1} = 2\beta_{h-2} = 2^{h-1}$$

and of course

$$2\beta_{h-1} + \omega_{h-1} = N+1.$$

These two equations form a linear system in β_{h-1} and ω_{h-1} which when solved gives the results stated in the theorem. □

Before we prove the second half of our result characterizing NVO 1-2 brother trees we show the following lemma:

LEMMA 3.6. *Let T be a 1-2 brother tree with N keys and minimal possible height $h = \lceil \log_2(N+1) \rceil \geq 5$ where $2^{h-1} < N+1 < 3\cdot 2^{h-2}$. If T is NVO then T is a complete binary tree up to level $h-4$.*

Proof. Let $\Delta = \langle\omega_0, \beta_0\rangle, \ldots, \langle\omega_h, \beta_h\rangle$ be the detailed profile of a NVO 1-2 brother tree T with height h, where h obeys the assumptions of the lemma.

CLAIM 1. $\omega_{h-4} = 0$.

If not Lemma 3.2 tells us that $\beta_{h-4} = \omega_{h-3}$ or $\beta_{h-3} = \omega_{h-2}$.

Case 1. $\beta_{h-4} = \omega_{h-3}$ and $\beta_{h-3} \neq \omega_{h-2}$ (that is $\beta_{h-3} > \omega_{h-2}$).

We will show that there exists a 1-2 brother tree T' representing the same number of keys with detailed profile

$$\Delta' = \langle\omega_0, \beta_0\rangle, \ldots, \langle\omega_{h-5}, \beta_{h-5}\rangle, \langle\omega_{h-4}-1, \beta_{h-4}+1\rangle, \langle\omega_{h-3}+1, \beta_{h-3}\rangle,$$
$$\langle\omega_{h-2}+3, \beta_{h-3}-2\rangle, \langle\omega_{h-1}-2, \beta_{h-2}+1\rangle, \langle\omega_h, \beta_h\rangle$$

which has better NVCOST. Application of Lemma 3.2 for $k = h-1$ and the assumptions $\omega_{h-3} = \beta_{h-4} \neq 0$ and $\beta_{h-3} \neq \omega_{h-2}$ yield $\beta_{h-2} = \omega_{h-1}$. This equation leads to $N+1 = 3\beta_{h-2} + 2\omega_{h-2}$ by Proposition 3.4.

Because $\beta_{h-2} \geq \omega_{h-2}$ the assumptions $\beta_{h-2} \leq 3$ and $h \geq 5$ lead to $N+1 < 17$, a contradiction. Thus we know that all numbers occurring in Δ' are nonnegative. It is easy to show that Δ' is the profile of a binary tree. Thus it remains to prove that Δ' is the detailed profile of a 1-2 brother tree. We need to show only that condition (2.6) holds for Δ'. The only nontrivial case is the proof of $\beta_{h-3} \geq \omega_{h-3}+3$.

First we have

$$\beta_{h-3} + \omega_{h-3} \leq 2^{h-3} - \omega_{h-4},$$

so

$$\beta_{h-2} + \omega_{h-2} = 2\beta_{h-3} + \omega_{h-3} \leq 2^{h-3} - \omega_{h-4} + \beta_{h-3}.$$

From Proposition 3.4 we infer

$$\beta_{h-2} = \frac{1}{3}(N+1) - \frac{2}{3}\omega_{h-2}.$$

Thus we obtain

$$\frac{1}{3}(N+1) - \frac{2}{3}\omega_{h-2} + \omega_{h-2} \leq 2^{h-3} - \omega_{h-4} + \beta_{h-3}.$$

This is equivalent to

$$\beta_{h-3} \geq \frac{1}{3}\omega_{h-2} + \omega_{h-4} + \frac{1}{3}(N+1) - 2^{h-3}$$
$$> \frac{1}{3}\omega_{h-2} + \omega_{h-4} + \frac{1}{3}2^{h-1} - 2^{h-3},$$

so

$$\beta_{h-3} > \frac{1}{3}\omega_{h-2} + \omega_{h-4} + \frac{1}{3}2^{h-3}. \tag{3.4}$$

On the other hand we have

$$\beta_{h-3} + \omega_{h-3} = 2\beta_{h-4} + \omega_{h-4}$$
$$= \omega_{h-3} + \beta_{h-4} + \omega_{h-4};$$

by the first assumption of Case 1. Hence

$$\beta_{h-3} = \beta_{h-4} + \omega_{h-4} \leq 2^{h-4}.$$

Now using the second assumption of Case 1 we obtain $\omega_{h-2} < 2^{h-4}$. Then

$$2\omega_{h-2} < 2^{h-3},$$
$$3\omega_{h-2} < 2^{h-3} + \omega_{h-2},$$
$$\omega_{h-2} < \frac{1}{3}2^{h-3} + \frac{1}{3}\omega_{h-2},$$
$$\frac{1}{3}2^{h-3} + \frac{1}{3}\omega_{h-2} + \omega_{h-4} > \omega_{h-2} + \omega_{h-4}.$$

From the last inequality and (3.4) we obtain

$$\beta_{h-3} \geq \omega_{h-2} + \omega_{h-4} + 2 \geq \omega_{h-2} + 3.$$

This completes Case 1.

Case 2. $\beta_{h-3} = \omega_{h-2}$

We will show that there exists a 1-2 brother tree T' representing the same number of keys with detailed profile

$$\Delta' = <\omega_0, \beta_0>, \ldots, <\omega_{h-5}, \beta_{h-5}>, <\omega_{h-4}-1, \beta_{h-4}+1>,$$
$$<\omega_{h-3}+3, \beta_{h-3}-2>, <\omega_{h-2}-3, \beta_{h-2}+2>, <\omega_{h-1}+2, \beta_{h-1}-1>, <\omega_h, \beta_h>$$

which has better NVCOST.

Clearly, $\beta_{h-1} > 0$ and thus $\beta_{h-1} \neq \omega_h$ ($\omega_h = 0$ by convention). Furthermore, the assumption $\beta_{h-3} = \omega_{h-2}$ implies $\omega_{h-2} \neq 0$. Thus, application of Lemma 3.2 for $k = h$ yields $\beta_{h-2} = \omega_{h-1}$. From this equation we obtain by Proposition 3.4

$$N+1 = 3\beta_{h-2} + 2\omega_{h-2}.$$

In order to show that all numbers occurring in Δ' are nonnegative it remains to show that $\beta_{h-3} = \omega_{h-2} \geq 3$. We know

$$2\beta_{h-3} + \omega_{h-3} = \beta_{h-2} + \omega_{h-2} = \beta_{h-2} + \beta_{h-3},$$

so

$$\beta_{h-2} = \beta_{h-3} + \omega_{h-3}.$$

Again using Proposition 3.4 the last equation and the assumption of Case 2 imply:

$$\begin{aligned} N+1 &= 3\beta_{h-2} + 2\omega_{h-2} \\ &= 3(\beta_{h-3} + \omega_{h-3}) + 2\beta_{h-3} \\ &= 5\beta_{h-3} + 3\omega_{h-3}. \end{aligned} \tag{3.5}$$

On the other hand

$$\beta_{h-3} + \omega_{h-3} \leq 2^{h-3} - \omega_{h-4},$$

so

$$\omega_{h-3} \leq 2^{h-3} - \omega_{h-4} - \beta_{h-3}. \tag{3.6}$$

From (3.5) and (3.6) we obtain

$$N+1 \leq 2\beta_{h-3} + 2^{h-1} - 2^{h-3} - 3\omega_{h-4}.$$

Using $2^{h-1} < N+1$ we obtain

$$2^{h-4} + \frac{3}{2}\omega_{h-4} < \beta_{h-3}. \tag{3.7}$$

Since we have assumed $\omega_{h-4} > 0$ we know that $\beta_{h-3} \geq 3$.

It is easy to show that Δ' is the detailed profile of a binary tree. Thus it remains to prove that Δ' is the detailed profile of a 1-2 brother tree. For this we need to show only that condition (2.6) holds for Δ'. The only nontrivial case is the proof of $\beta_{h-4} + 1 \geq \omega_{h-3} + 3$.

Clearly, $\beta_{h-4} + \omega_{h-4} \leq 2^{h-4}$. Using (3.7) we obtain

$$\beta_{h-4} + \frac{5}{2}\omega_{h-4} < \beta_{h-3}.$$

On the other hand

$$2\beta_{h-4} + \omega_{h-4} = \beta_{h-3} + \omega_{h-3},$$

so

$$2\beta_{h-4} + \omega_{h-4} > \beta_{h-4} + \frac{5}{2}\omega_{h-4} + \omega_{h-3}.$$

Therefore

$$\beta_{h-4} > \omega_{h-3} + \frac{3}{2}\omega_{h-4}.$$

Since $\omega_{h-4} > 0$ this implies $\beta_{h-4} \geq \omega_{h-3} + 2$, completing the proof of Case 2 and of our first claim $\omega_{h-4} = 0$.

CLAIM 2. $\omega_{h-5} = 0$.

If not, $h > 5$ must hold and Lemma 3.2 tells us that $\beta_{h-4} = \omega_{h-3}$ (because $\beta_{h-5} > \omega_{h-4} = 0$, by our first claim).

Again applying Lemma 3.2 for $k = h-1$ we have

$$\omega_{h-3} = 0 \quad \text{or} \quad \beta_{h-3} = \omega_{h-2} \quad \text{or} \quad \beta_{h-2} = \omega_{h-1}. \tag{3.8}$$

Since $\omega_{h-4} = 0$ (by Claim 1) and $\omega_{h-5} > 0$ (by assumption), Corollary 3.3 tells us that $\omega_{h-3} > 0$.

We now argue that $\beta_{h-2} = \omega_{h-1}$ is true. If not, (3.8) tells us that $\beta_{h-3} = \omega_{h-2}$. Again applying Lemma 3.2 for $k = h$ we can infer from $\beta_{h-3} = \omega_{h-2}$ that $\beta_{h-2} = \omega_{h-1}$ - a contradiction.

We will now show that there exists a 1-2 brother tree T' representing the same number of keys with detailed profile

$$\Delta' = <\omega_0,\beta_0>,\ldots,<\omega_{h-6},\beta_{h-6}>,<\omega_{h-5}-1,\beta_{h-5}+1>,<\omega_{h-4},\beta_{h-4}+1>,$$
$$<\omega_{h-3}+1,\beta_{h-3}+1>,<\omega_{h-2}+9,\beta_{h-2}-6>,<\omega_{h-1}-6,\beta_{h-1}+3>,<\omega_h,\beta_h>$$

which has better NVCOST. First we conclude from $\beta_{h-2}=\omega_{h-1}$ that

$$3\beta_{h-2}+2\omega_{h-2}=N+1, \quad \text{by Proposition 3.4}$$
$$> 2^{h-1}, \quad \text{by assumption,}$$

so

$$5\beta_{h-2}>2^{h-1}, \quad \text{by (2.7)}$$
$$\beta_{h-2}>\frac{1}{5}2^{h-1}\geq 6, \quad \text{because} \quad h>5.$$

This shows that all numbers occurring in the detailed profile Δ' are nonnegative. In order to show that Δ' is the detailed profile of a 1-2 brother tree, it is sufficient to prove $\beta_{h-3}+1\geq\omega_{h-2}+9$. We briefly recall what we know about the detailed profile Δ': $\omega_{h-5}>0$, $\omega_{h-4}=0$, $\beta_{h-4}=\omega_{h-3}$, $\beta_{h-2}=\omega_{h-1}$. As consequence we obtain furthermore: $\beta_{h-3}=\omega_{h-3}=\beta_{h-4}$, since $2\beta_{h-4}+\omega_{h-4}=\beta_{h-3}+\omega_{h-3}$. In order to get an upper bound for ω_{h-2}, the number of unary nodes on level $h-2$, we first show that there must be considerably more binary than unary nodes on level $h-2$:

$$\beta_{h-2}+\omega_{h-2} = 2\beta_{h-3}+\omega_{h-3} = 3\beta_{h-4}$$
$$= 2\beta_{h-4}+\beta_{h-3}$$
$$\geq 2\beta_{h-4}+\omega_{h-2}, \quad \text{by (2.6)}$$

so, $\beta_{h-2}\geq 2\beta_{h-4}$ and $\omega_{h-2}\leq\beta_{h-4}$. Let $\beta_{h-2}=2\omega_{h-2}+x$, where $x\geq 0$. Then

$$N+1 = 3\beta_{h-2}+2\omega_{h-2} = 8\omega_{h-2}+3x$$

that is $x=\frac{1}{3}(N+1)-\frac{8}{3}\omega_{h-2}$. Now

$$\beta_{h-2}+\omega_{h-2} = 2\omega_{h-2}+x+\omega_{h-2}$$
$$= \frac{1}{3}\omega_{h-2}+\frac{1}{3}(N+1) = 3\beta_{h-4};$$

hence

$$\omega_{h-2}+2^{h-1} < 9\beta_{h-4}$$
$$\leq 8(2^{h-4}-\omega_{h-5})+\beta_{h-4}$$
$$= \beta_{h-4}+2^{h-1}-8\omega_{h-5}.$$

Since $\beta_{h-4}=\beta_{h-3}$ we obtain $\beta_{h-3}>\omega_{h-2}+8\omega_{h-5}\geq\omega_{h-2}+8$. This concludes the proof of our second claim.

Application of Corollary 3.3 yields the result stated in the lemma. □

We are now ready for the second half of our result characterizing NVO 1-2 brother trees.

THEOREM 3.7. *Let T be a* 1-2 *brother tree with N keys and minimal possible*

height $h = \lceil \log_2(N+1) \rceil$, *where* $2^{h-1} < N+1 < 3 \cdot 2^{h-2}$. *Then T is NVO if and only if its detailed profile* $\Delta = <\omega_0, \beta_0>, \ldots, <\omega_h, \beta_h>$ *satisfies*

(i) $\beta_k = 2^k$, $\omega_k = 0$ *for* $0 \le k \le h-4$ *and*

(ii) $\beta_{h-2} = \omega_{h-1}$.

Proof. Let T be a NVO 1-2 brother tree with N keys and height $h = \lceil \log_2(N+1) \rceil$, where $2^{h-1} < N+1 < 3 \cdot 2^{h-2}$. Clearly, (i) holds because of Lemma 3.6. Application of Lemma 3.2 for $k = h$ yields $\omega_{h-2} = 0$ or $\beta_{h-2} = \omega_{h-1}$ or $\beta_{h-1} = \omega_h$. The last alternative is false, because $\omega_h = 0$ by definition.

We show that $\beta_{h-2} = \omega_{h-1}$. If not, then $\omega_{h-2} = 0$ must hold. Application of Lemma 3.2 for $k = h-1$ yields $\omega_{h-3} = 0$ or $\beta_{h-3} = \omega_{h-2}$ or $\beta_{h-2} = \omega_{h-1}$. Clearly, the first alternative must hold, because $\beta_{h-2} \ne \omega_{h-1}$ by assumption and $\beta_{h-3} \ne \omega_{h-2} = 0$.

Thus we know that T must be complete binary up to level $h-2$. This implies $N+1 \ge 3 \cdot 2^{h-2}$ - a contradiction. This completes the proof of (ii).

In order to show the "only if" part assume that T is a 1-2 brother tree with minimal possible height h whose detailed profile Δ fulfills (i) and (ii). The equation $\beta_{h-2} = \omega_{h-1}$ implies (cf. Proposition 3.4) $(N+1) = 3\beta_{h-2} + 2\omega_{h-2}$ and $\beta_{h-1} = \beta_{h-2} + \omega_{h-2}$. Hence, we infer:

$$\begin{aligned}(\omega_{h-2} + \beta_{h-2}) + (\omega_{h-1} + \beta_{h-1}) &= (\omega_{h-2} + \beta_{h-2}) + (\beta_{h-2} + \beta_{h-2} + \omega_{h-2}) \\ &= 3\beta_{h-2} + 2\omega_{h-2} \\ &= N+1.\end{aligned}$$

Now we use the fact that T's NVCOST depends only on T's profile, that is on the numbers $\nu_k = \omega_k + \beta_k$, for $k = 0, \ldots, h$. Thus, we obtain from (2.8):

$$\begin{aligned}\text{NVCOST}(T) &= h \cdot \nu_h - \sum_{k=0}^{h-1} \nu_k \\ &= h(N+1) - \sum_{k=0}^{h-4} 2^k - [(\omega_{h-3} + \beta_{h-3}) + (\omega_{h-2} + \beta_{h-2}) + (\omega_{h-1} + \beta_{h-1})] \\ &= h(N+1) - (2^{h-3} - 1) - (2^{h-3} + N + 1) \\ &= (h-1)(N+1) - 2^{h-2} + 1.\end{aligned}$$

This shows that the NVCOST of a 1-2 brother tree T with minimal possible height h whose detailed profile fulfills (i) and (ii) depends only on the number of leaves $N+1$. Now the "if" part of the theorem tells us that T must be NVO. □

3.1 An algorithm for constructing NVO 1-2 brother trees. Our characterization of NVO 1-2 brother trees directly leads to a linear-time algorithm for constructing such a tree T for a given sorted list of N keys. It should be clear that it suffices to construct a detailed profile of a NVO 1-2 brother tree. The construction in linear time of the skeleton of a brother tree from the given detailed profile is straightforward. The skeleton is then filled with the sorted list of N keys by traversing T in inorder and depositing the next key whenever a binary node is visited.

Procedure NVO-Profile

Input: Natural number N (of keys to be stored)

Output: The detailed profile of a NVO 1-2 brother tree with N binary nodes and $N+1$ leaves.

begin
  $h := \lceil \log_2(N+1) \rceil$;
  if $N+1 \geq 3 \cdot 2^{h-2}$
  then $\Delta = \langle 0,2^0 \rangle, \ldots, \langle 0,2^{h-2} \rangle, \langle 2^h-(N+1),(N+1)-2^{h-1} \rangle, \langle 0,N+1 \rangle$
  else $\{2^{h-1} < N+1 < 3 \cdot 2^{h-2}\}$
  begin
    determine i, $0 \leq i \leq 3$, such that
    $(8+i) \cdot 2^{h-4} \leq (N+1) < (9+i)2^{h-4}$;
    $x := (9+i) \cdot 2^{h-4} - (N+1)$;
    $y := -(8+i) \cdot 2^{h-4}$;
    {note that$(N+1) = (8+i) \cdot x + (9+i) \cdot y$}
    case i **of**
    $0{:}\Delta := \langle 0,2^0 \rangle, \ldots, \langle 0,2^{h-4} \rangle, \langle 2^{h-4},2^{h-4} \rangle,$
        $\langle x,2x+3y \rangle, \langle 2x+3y,3x+3y \rangle, \langle 0,N+1 \rangle$;
    $1{:}\Delta := \langle 0,2^0 \rangle, \ldots, \langle 0,2^{h-4} \rangle, \langle x,x+2y \rangle,$
        $\langle 2y,3x+2y \rangle, \langle 3x+2y,3x+4y \rangle, \langle 0,N+1 \rangle$;
    $2{:}\Delta := \langle 0,2^0 \rangle, \ldots, \langle 0,2^{h-4} \rangle, \langle 0,2^{h-3} \rangle,$
        $\langle 2x+y,2x+3y \rangle, \langle 2x+3y,4x+4y \rangle, \langle 0,N+1 \rangle$;
    $3{:}\Delta := \langle 0,2^0 \rangle, \ldots, \langle 0,2^{h-4} \rangle, \langle 0,2^{h-3} \rangle,$
        $\langle x,3x+4y \rangle, \langle 3x+4y,4x+4y \rangle, \langle 0,N+1 \rangle$
    end
  end
end.

It is easy to check that the above algorithm generates optimal profiles of 1-2 brother trees. These profiles are not necessarily unique.

4. Space optimal 1-2 brother trees. In this section we characterize those 1-2 brother trees which have minimum storage requirement among all 1-2 brother trees with the same number of keys (or, equivalently, of leaves). It turns out that the trees with the minimal number of nodes must have also minimal height, but not conversely. All 1-2 brother trees with the same number of keys (or leaves) have the same number of binary nodes. This means that a 1-2 brother tree has minimal SPACECOST if and only if it has the minimal number of unary nodes.

Our first observation concerning SCO 1-2 brother trees is that the total number of nodes strictly increases with the number of stored keys.

LEMMA 4.1. *Let T_N and T_{N+1} be two SCO 1-2 brother trees with N and $N+1$ leaves respectively, (or, equivalently $N-1$ and N keys, respectively). Then the total number of nodes in T_N is strictly less than the total number of nodes in T_{N+1}.*

Proof. Consider the SCO 1-2 brother tree T_{N+1}. Remove one of its keys (and leaves) by performing the *delete* procedure of Ottmann and Wood [1981]. This procedure restructures the tree in order to retain the brother tree structure. Inspection of this procedure shows that the total number of nodes (in T_{N+1}) is decreased. Let T_N' denote the resulting 1-2 brother tree with N leaves. Then we have:

$$\text{SPACECOST}(T_N) \leq \text{SPACECOST}(T_N') < \text{SPACECOST}(T_{N+1})$$

since we have removed (at least) one node from T_{N+1}. □

Let a 1-2 brother tree T_{N+1} with $N+1$ leaves, $N \geq 0$, be constructed with the following profile:

$$\nu_h = N+1 \ , \ \nu_i = \left\lceil \nu_{i+1}/2 \right\rceil , 0 \leq i < h \text{, where } h = \left\lceil \log_2(N+1) \right\rceil . \quad (4.1)$$

We first prove that T_{N+1} is SCO.

THEOREM 4.2. *For all* $N \geq 0$, T_{N+1} *constructed with profile given by* (4.1) *is SCO.*

Proof. Clearly T_1 is SCO. Complete the proof by induction on the number of leaves of T_{N+1}. Assume T_{k+1} is SCO for all k, $0 \leq k < N$, for some $N \geq 0$. Consider T_{N+1}. If T_{N+1} is not SCO, then there exists a T'_{N+1} which is SCO, whose profile is different from (4.1). Let $\nu'_0,...,\nu'_{h'}$ denote the profile of T'_{N+1}, where $h' \geq h$. Compare ν_h and $\nu'_{h'}, \nu_{h-1}$ and $\nu'_{h'-1}, \cdots$. Let $j \geq 1$ be the least integer such that $\nu_{h-j} \neq \nu'_{h'-j}$.

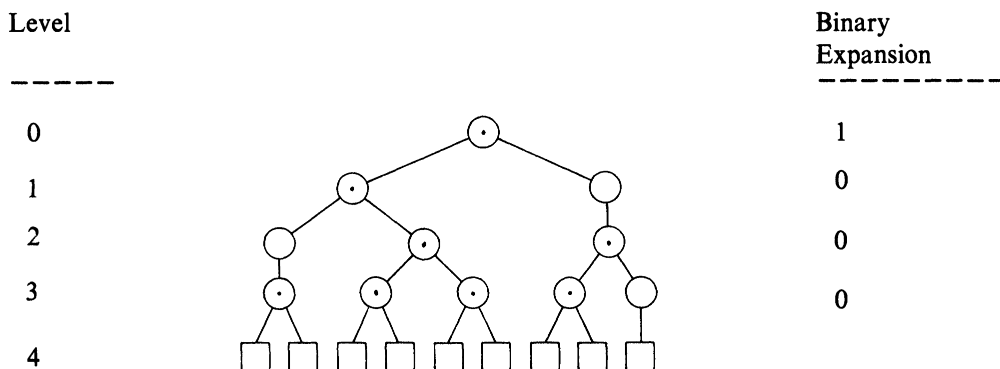
If $j > 1$ then we can replace the prefix of T'_{N+1} of height $h' - j + 1$ with the SCO tree with ν_{h-j+1} leaves, by our inductive assumption. If $j = 1$ then $\nu'_{h'-1} > \nu_{h-1}$, in which case SPACECOST($T_{\nu'_{h'-1}}$) > SPACECOST($T_{\nu_{h-1}}$), by Lemma 4.1, contradicting the optimality of T'_{N+1}. Both cases lead to the conclusion that T_{N+1} is SCO. Hence the result. □

The following corollary follows easily from Theorem 4.2:

COROLLARY 4.3. *A 1-2 brother tree is SCO if and only if it has at most one unary node on each level (that is, if and only if its profile is given by* (4.1)).

This also implies that a SCO 1-2 brother tree must be height-minimal. The converse is not true, as can be seen from Fig. 1. Furthermore, SCO profiles are unique for any value of N, although this is not true for NVO profiles. Finally, we mention that we can read off from the binary expansion of the number N on which levels the unary nodes (if any) occur in SCO 1-2 brother trees. *A* value of 0 corresponds to a unary level.

Example. An 8-key SCO 1-2 brother tree must have exactly one unary node on levels 1, 2, 3. For $8 = (1000)_2$. Thus, the tree may be as follows:



As in §3 we simply generate the detailed profile of an SCO tree, from which a corresponding skeletal brother tree T is easily generated. The N keys are sorted and then T is filled in with an inorder traversal. In §5 we give an explicit SCO tree construction algorithm.

Procedure SCO Profile

Input: Natural number N (of keys to be stored).

Output: The detailed profile of an SCO 1-2 brother tree with N binary nodes and $N+1$ leaves.

begin $h := \lceil \log_2(N+1) \rceil$
$\omega_h := 0 ; \beta_h := N+1 ; \nu_h := N+1$
for $i := h-1$ **downto** 0 **do**
begin $\beta_i := \nu_{i+1}$ **div** 2;
$\omega_i := \nu_{i+1}$ **mod** 2 ;
$\nu_i := \omega_i + \beta_i$
end;
Let $\Delta = <\omega_0, \beta_0>, \ldots, <\omega_h, \beta_h>$
end.

5. Comparison cost optimal 1-2 brother trees. We will characterize structurally those 1-2 brother trees that have minimal comparison cost among all 1-2 brother trees with a given number of leaves (or, equivalently, of keys). Our characterization uses the close correspondence between brother trees and AVL trees (see Ottmann, Six and Wood [1979]). Consider a brother tree T. Let contract(T) denote the tree obtained by replacing each unary node p in T by its only son σp. Then the resulting tree is height-balanced, that is for each node p, the balance factor of p, that is the height difference between the left and right subtree of p, is $+1$, 0 , or -1 . NVCOST and COMPCOST are defined for AVL trees in analogy to the related cost measures of 1-2 brother trees. Clearly, the two cost measures coincide for AVL trees. Further, if T is a 1-2 brother tree then COMPCOST(T) = NVCOST(contract(T)). Hence, characterizing the CCO 1-2 brother trees amounts to characterizing the NVO AVL trees.

LEMMA 5.1. *An AVL tree T is NVO (or, equivalently, CCO) if and only if all leaves of T occur on (at most) two adjacent levels.*

Proof. Clearly, an AVL tree is NVO if and only if its internal path length is minimal. Knuth [1973, §2.3.4.5] has shown that a binary tree has minimal internal path length if and only if all its leaves occur on at most two adjacent levels. □

Let T be a 1-2 brother tree. Then all leaves of contract(T) occur on (at most) two adjacent levels if and only if every root to leaf path of T contains at most one unary node. This yields the desired characterization of CC0 1-2 brother trees:

THEOREM 5.2. *A* 1-2 *brother tree T is CCO if and only if every root-to-leaf path of T contains at most one unary node.*

Proof. The stated condition is clearly sufficient by Lemma 5.1 and the preceding remark. Now consider a 1-2 brother tree T which contains a path with more than one unary node on it. Then contract(T) is an AVL tree in which not all leaves appear on two adjacent levels; hence, contract(T) is not NVO and, therefore, T cannot be CCO. □

NVCOST and COMPCOST are independent cost measures. For example, the 17-leaf NVO tree must have two unary nodes on some root-to-leaf path, hence cannot be CCO. On the other hand, the 6-leaf tree T_a of Fig. 1 is CCO but not NVO. In particular, COMPCOST is not a function of a tree's detailed profile. Fig. 1 further shows that COMPCOST optimality does not imply SPACECOST optimality. However, each SCO 1-2 brother tree can be transformed to a (profile equivalent SCO and)

CCO tree. Rather than providng this directly we give an algorithm which constructs an SCO 1-2 brother tree which is simultaneously CCO, using the binary expansion of N.

Procedure SCO and CCO 1-2 Brother Tree.

Input: Natural number N (of keys to be stored).

Output: A 1-2 brother tree with N binary nodes which is simultaneously SCO and CCO.

```
begin h := ⌈log₂(N+1)⌉;
   if N=0 then return else
   if N=1 then return else
   begin Create a binary node p, say;
      if the coefficient of 2^(h-2) in the binary expansion
         of N is 0 then
      begin { insert unary node }
         a) Attach a unary node q as the right son of p
            and attach a complete binary tree of height
            h-2 as q's only subtree.
         b) Attach a subtree of N-2^(h-2) binary nodes,
            as p's left subtree, constructed recursively.
      end else { both sons are binary }
            Attach a subtree of (N-1) div 2 binary nodes
            as p's left subtree, and one of N-1-(N-1) div 2
            binary nodes as p's right subtree; both constructed
            recursively.
   end
end.
```

6. A comparison of the cost-measures. The two cost measures NVCOST and SPACECOST are distinct in the sense that there are N-key 1-2 brother trees which are NVO but not SCO and 1-2 brother trees which are SCO but not NVO. However, the cost measures are not totally unrelated. We already know that NVO and SCO 1-2 brother trees both must have the minimal height $h = \lceil \log_2(N+1) \rceil$, where N is number of keys. Moreover NVO and SCO 1-2 brother trees are in one sense dual to each other, since, for a given height, in NVO trees the number of nodes is maximized while in SCO trees the number of nodes is minimized. Thus, for $N = 2^h - 1$ the complete binary tree of height h is both NVO and SCO. In general, the SPACECOST of the NVO 1-2 brother tree exceeds the SPACECOST of the SCO tree with the same number of keys; and the NVCOST of the SCO tree exceeds the NVCOST of the NVO tree. But, by how much may they differ?

Recall that for each 1-2 brother tree T of height h with N keys the following holds:

$$\mathrm{NVCOST}(T) + \mathrm{SPACECOST}(T) = h \cdot (N+1) \quad . \tag{6.1}$$

Let $T_{\mathrm{SCO}}(N)$ denote the SCO 1-2 brother tree with N keys and height $h = \lceil \log_2(N+1) \rceil$, and let $T_{\mathrm{NVO}}(N)$ denote the corresponding NVO tree.

Summing up the numbers of nodes in profiles for $T_{\mathrm{NVO}}(N)$ given in Theorems 3.5 and 3.7, we find

$$\text{SPACECOST}(T_{\text{NVO}}(N)) = \begin{cases} N+2^{h-2} & \text{if } 2^{h-1}<N+1\le 3\cdot 2^{h-2}, \\ 2^h-1 & \text{if } 3\cdot 2^{h-2}<N+1\le 2^h. \end{cases} \tag{6.2}$$

From (6.1) we may then infer

$$\text{NVCOST}(T_{\text{NVO}}(N)) = \begin{cases} h\cdot(N+1)-N-2^{h-2} & \text{if } 2^{h-1}<N+1\le 3\cdot 2^{h-2}, \\ h\cdot(N+1)-2^h+1 & \text{if } 3\cdot 2^{h-2}<N+1\le 2^h. \end{cases} \tag{6.3}$$

Corollary 4.3 and (6.1) also give us the results:

$$N \le \text{SPACECOST}(T_{\text{SCO}}(N)) \le N+(h-1), \tag{6.4}$$

$$(h-1)N+1 \le \text{NVCOST}(T_{\text{SCO}}(N)) \le (h-1)N+h. \tag{6.5}$$

Combining (6.2) with (6.4) we easily find

$$1 \le \frac{\text{SPACECOST}(T_{\text{NVO}}(N))}{\text{SPACECOST}(T_{\text{SCO}}(N))} < \frac{3}{2}.$$

Similarly using (6.3) and (6.5) we find that, for all N ,

$$\frac{\text{NVCOST}(T_{\text{SCO}}(N))}{\text{NVCOST}(T_{\text{NVO}}(N))} \le \frac{(h-1)N+h}{(h-1)N+h-2^{h-2}}$$

so for $h \ge 3$ we obtain, since $N \ge 2^{h-1}$,

$$1 \le \frac{\text{NVCOST}(T_{\text{SCO}}(N))}{\text{NVCOST}(T_{\text{NVO}}(N))} < \frac{4}{3}.$$

In fact this bound can also be shown to hold for $h=1$ or 2. Thus we have shown that, for any value of N, the SPACECOSTs and NVCOSTs of optimal trees are within a small constant factor of each other.

7. Optimal brother leaf search trees. Recall that a brother leaf search has all the keys at the leaf level, while the internal binary nodes contain separating or routing information. Hence to access a key stored in a brother leaf search tree, each node on the path from the root to the appropriate leaf must be visited. Therefore, the node-visit cost of a brother leaf search tree T can be defined by

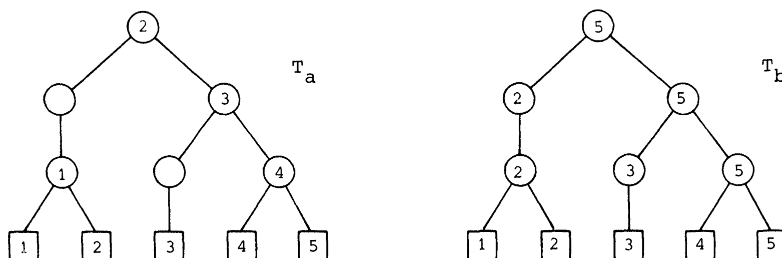
$$\text{NVCOST}(T) = (\text{height}(T)+1)\cdot(\text{ number of leaves of } T\text{ }).$$

This implies that an N-key brother leaf search tree T, that is an N leaf tree, is NVO if and only if T has the minimal possible height $h = \lceil \log_2 N \rceil$.

This definition of the NVCOST and the resulting trivial characterization of NVO brother leaf search trees implicitly presupposes that separators are assigned to internal nodes in such a way that a search for a key in the tree only visits nodes which lie on the root to leaf path. This presupposition does not hold if we assign to each internal node the maximum value of its sons, as in Ottmann and Six [1976], since this implies the left sons of nodes on the search path are also visited. But if we assign to each (internal) binary node p the maximum value in the left subtree of p as in Mehlhorn

[1977], then the presupposition does indeed hold. This indicates that the definition of NVCOST for brother leaf search trees is *dependent* on the routing scheme used, cf. Kwong and Wood [1980].

The routing scheme is also crucial to the definition of COMPCOST for a brother leaf search tree. This is because COMPCOST depends on the underlying search procedure. Consider, for example, the following two sample trees T_a and T_b, where T_a has separators which are the maximum value in the left subtree, and T_b has separators which are the maximum value in the right or only subtree. Note that by the results of §3, 4, and 5 T_a and T_b are NVO, SCO, and CCO brother trees.



In order to retrieve a key x stored in the tree T_a we may perform $search_a(root(T_a), x)$ where $search_a$ is the following procedure:

Procedure $search_a(p,x)$

Case 1 [p is a leaf]

if $value(p) = x$ **then return** (p) **else** nil;

Case 2 [p is unary, that is p has a single son σp]

$search_a(\sigma p, x)$

Case 3 [p is binary, that is p has a left son λp and a right son ρp]

if $x \leq value(p)$ **then** $search_a(\lambda p, x)$ **else**
$search_a(\rho p, x)$.

In order to access each key stored in T_a exactly once via $search_a$, a total number of 17 comparisons is required. The routing scheme for T_b suggests retrieving a key x by performing $search_b(root(T_b), x)$, where $search_b$ is obtained from $search_a$ by replacing Case 3 (where p is binary) by the following:

compare x and value(λp). (This comparison has one of three different possible outcomes)

Case 3.1 [$x = value(\lambda p)$]

if λp is a leaf **then return** (λp) **else** $search_b(\lambda p, x)$;

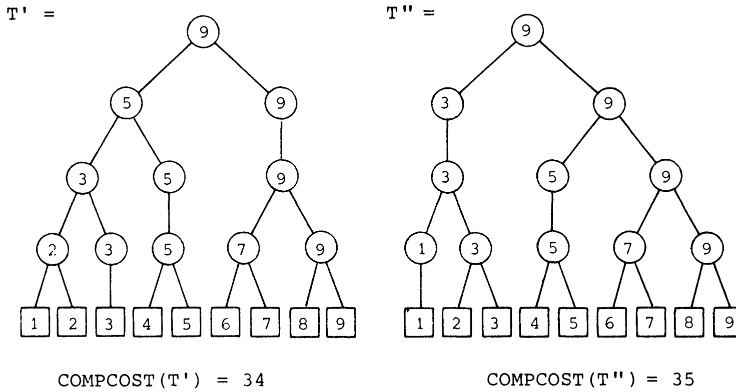
Case 3.2 [$x < value(\lambda p)$]

if λp is a leaf **then** nil **else** $search_b(\lambda p, x)$;

Case 3.3 [$x > value(\lambda p)$]

$search_b(\rho p, x)$.

This example shows why a uniform definition of COMPCOST is impossible for brother leaf search trees. We must consider a definition of COMPCOST with respect to a particular routing scheme (or a class of routing schemes). Furthermore even when a particular routing scheme is chosen the COMPCOST may not even be characterized by the detailed profile. For example, the following brother leaf search trees T' and T' both have the same detailed profile, and the same routing scheme, but different COMPCOSTs:



Fortunately no such difficulties arise when defining SPACECOST for brother leaf search trees. Since the leaves are now used, a brother leaf search tree T with profile $\pi(T) = \nu_0,\ldots,\nu_h$, $h = \text{height}(T)$, has space-cost

$$\text{SPACECOST}(T) = \sum_{i=0}^{h} \nu_i .$$

Since the SPACECOST of an N-key brother leaf search tree is $N+1+\sum_{i=0}^{h-1} \nu_i$, minimizing the SPACECOST amounts to minimizing the number of internal nodes. Thus we obtain the same characterization of SCO brother leaf search trees as already proved for 1-2 brother trees in §5.

8. Concluding remarks and history. We have characterized the 1-2 brother trees which are optimal with respect to one of three cost measures, NVCOST, COMPCOST and SPACECOST. In particular we have shown that SCO and CCO 1-2 brother trees having N binary nodes have the same detailed profile. Moreover an NVO 1-2 brother tree with N binary nodes has a SPACECOST differing by at most 50% from the SPACECOST of an SCO 1-2 brother tree with N binary nodes. Similarly the NVCOST of an SCO 1-2 brother tree differs by at most 33% from that of an NVO 1-2 brother tree with the same number of binary nodes. Thus the optimal trees according to one measure are nearly optimal with respect to the other measures.

We have also considered, albeit briefly, the optimality of brother leaf search trees. Although SPACECOST is defined and characterized as for 1-2 brother trees, we demonstrate that NVCOST and COMPCOST are both dependent upon the particular routing scheme that is used. Moreover we have shown that even when a routing scheme is chosen it is possible that the CCO and NVO trees cannot be characterized solely by detailed profiles.

This leads us to a number of open problems. First for the classical routing

scheme (an internal binary node is assigned the maximum value in its left subtree) characterize NVO and CCO brother leaf search trees. Second do this for the other routing scheme mentioned here (an internal node is assigned the maximum value in its subtrees). Third, investigate 1-2 brother trees which are simultaneously SCO and NVO. Fourth, as in Rosenberg and Snyder [1981] consider the effects of insertions (and, possibly, deletions) on an initially optimal 1-2 brother tree. Fifth, investigate update schemes which maintain near optimality of 1-2 brother trees according to one of the desired measures. Sixth, how close are "random" 1-2 brother trees to being NVO, CCO and SCO? There are some preliminary results for some of these questions by one of us, namely H.W.Six.

Finally, in closing let us mention something of the history of the present paper. In 1978 the authors, apart from D. Stott Parker, submitted a first draft of this paper to SICOMP. However the results and proofs of §3, although correct, did not then have their present precise combinatorial flavor, along the lines of Miller et al. [1978]. Fortunately the referee insisted through two further drafts that such an approach was possible. Eventually, after we argued that it was impossible, he produced a detailed proof outline of a new §3, which became, in essence, the present §3. This so changed the orientation of the paper that we felt that the referee should be included as a coauthor. The referee was D. Stott Parker.

REFERENCES

G. M. Adelson-Velskii and Y. M. Landis, *An algorithm for the organization of information*, Dokl. Akad. Nauk SSSR, 146 (1962), pp. 263-266. (In Russian.)

A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

D. E. Knuth, *The Art of Computer Programming, Vol.* I, *Fundamental Algorithms*, Addison-Wesley, Reading, MA 1969.

K. Mehlhorn, *Effiziente algorithmen*, Teubner Studienbücher Informatik, Stuttgart, 1977.

R. E. Miller, N. J. Pippenger, A. L. Rosenberg and L. Snyder, *Optimal* 2,3-*trees*, SIAM J. Comput., 8 (1979), pp. 42-59.

Y. S. Kwong and D. Wood, *On B-trees: Routing schemes and concurrency*, Proceedings of the 1980 ACM/SIGMOD International Conference on Management of Data, 1980, pp. 207-213.

Th. Ottmann and H. W. Six, *Eine neue Klasse von ausgeglichenen Binärbäumen*, Angewandte Informatik, 18 (1976), pp. 395-400.

Th. Ottmann, H. W. Six and D. Wood, *On the correspondence between AVL trees and brother trees*, Computing, 23 (1979), pp. 43-54.

Th. Ottmann, H. W. Six and D. Wood, *Right brother trees*, Comm. ACM, 21 (1981), pp. 769-776.

Th. Ottmann and D. Wood, 1-2 *brother trees or AVL trees revisited*, Comput. J., 23 (1981), pp. 248-255.

A. L. Rosenberg and L. Snyder, *Minimal comparison* 2,3-*trees*, SIAM J. Comput, 7 (1978), pp. 465-480.

A. L. Rosenberg and L. Snyder, *Time- and space- optimality in B-trees*, ACM Trans. Data Base Systems, 6 (1981), pp. 174-183.

SIAM J. COMPUT.
Vol. 13, No. 2, May 1984

© 1984 Society for Industrial and Applied Mathematics
001

TWO TAPES ARE BETTER THAN ONE FOR NONDETERMINISTIC MACHINES*

PAVOL ĎURIŠ† AND ZVI GALIL‡

Abstract. It is known that k tapes are no better than two tapes for nondeterministic machines. We show here that two tapes are better than one. In fact, we show that two pushdown stores are better than one tape. Also, k tapes are no better than two for nondeterministic reversal-bounded machines; and we show that even two reversal-bounded pushdown stores are better than one reversal-bounded tape. We also show that for one-tape nondeterministic machines, unrestricted operation is better than reversal-bounded operation.

Key words. Turing machines, k-tape, two-tape, one-tape, real-time, nondeterministic machines, reversal-bounded, computational power

1. Introduction. This paper tries to complete our knowledge on the effect of the number of tapes on the computational power of nondeterministic machines. Our model of computation is a Turing machine with a *one-way read-only* input tape and with several working tapes. The input string is followed by a special endmarker. This is the model which is usually used when investigating the dependence of computational power on the number of tapes. A machine is *real-time* if it reads a new input symbol in every step, and the machine stops immediately after reading the endmarker.

The structure of the paper is the following: in the rest of this section we describe the results; in § 2 we give two basic lemmas; in § 3 we give the proofs; and in § 4, the conclusion, we list some open problems and consider briefly other models of Turing machines.

For deterministic Turing machines it is well known that by adding one more work tape one increases the computational power of the machine. Rabin [7] first showed that two tapes are better than one, and Aanderaa [1] showed that k tapes are better than $k-1$ tapes for $k \geqq 2$. By better we mean that there are languages accepted in real-time by the former but not by the latter. In fact, Aanderaa proved a slightly stronger result: there are languages accepted in real-time by k pushdown stores but not by $k-1$ tapes.

For nondeterministic machines the situation is completely different. Book and Greibach [2] proved that the hierarchy collapses very early, that the families of languages accepted by the following four machine types are the same:

I. linear-time multi-tape Turing machines,
II. real-time multi-tape Turing machines,
III. real-time two-tape Turing machines (a stack and a pushdown store suffice),
IV. real-time Turing machines with three pushdown stores.

Consequently, the problems that have been left open for nondeterministic machines are:

1. Are two tapes better than one?
2. Are two pushdown stores better than one tape?
3. Are three pushdown stores better than two?

One easily observes that the language used by Rabin to give an affirmative answer to Question 1 in the deterministic case, and the language used by Valiev [8] to prove

* Received by the editors March 19, 1982, and in final revised form February 15, 1983.

† Computer Center, Slovak Academy of Science, Bratislava, Czechoslovakia.
‡ School of Mathematical Sciences, Tel-Aviv University, Tel-Aviv, Israel and Computer Science Department, Columbia University, New York, New York 10027.

a similar result, can be accepted by a real-time one-tape nondeterministic machine, and thus cannot be used to answer Questions 1 to 3. Moreover, the language used by Rabin and the languages L_k used by Aanderaa to show that k tapes are better than $k-1$ can *all* be accepted by real-time nondeterministic (*one*!) pushdown machines. The main result of this paper answers Questions 1 and 2 in the affirmative. We still do not know the answer to Question 3.

Let $L_1 = \{x \# x | x \in \{0, 1\}^*, |x| = 2^m \text{ for some } m \geqq 0\}$, where $|x|$ denotes the length of x, and $\{0, 1\}^*$ is the set of finite strings over the alphabet $\{0, 1\}$.

THEOREM 1. *L_1 is accepted by no real-time one-tape nondeterministic Turing machine.*

This settles Question 1 because L_1 is accepted by a real-time two-tape machine. (It suffices to show that it is accepted by a linear-time multi-tape machine.)

To settle Question 2, consider the language L_1', $L_1' = \{x \# x^R | x \in \{0, 1\}^*, |x| = 2^m \text{ for some } m \geqq 0\}$, where x^R is the reversal of x.

THEOREM 2. *L_1' is accepted by a real-time (deterministic) two-pushdown machine, but by no real-time one-tape nondeterministic machine.*

Next we consider nondeterministic reversal-bounded machines. These are machines whose heads are allowed to make only a constant number of reversals. Our first observation is that the families of languages accepted by the reversal-bounded versions of the four machine types defined above are the same. The equality of the language families corresponding to I, II and IV is shown in [3]. The equality to the language family corresponding to III follows immediately from the observation that the proof in [2] for the unrestricted (not necessarily reversal-bounded) case yields a reversal-bounded simulation if the simulated machines are reversal-bounded. The fact that the four families of languages are the same raises Questions 1 to 3 about reversal-bounded machines.

One can show that L_1 can be accepted by a real-time two-tape reversal-bounded nondeterministic machine. (Again, it suffices to show that it is accepted by a linear-time multi-tape machine.) So, Question 1 is answered in the affirmative.

The following theorem answers in the affirmative Question 2 for reversal-bounded machines.

THEOREM 3. *L_1' can be accepted by a real-time nondeterministic two-pushdown reversal-bounded machine.*

It is still an open problem whether real-time nondeterministic Turing machines are better than similar reversal-bounded Turing machines. (For deterministic machines the answer is positive [4].) For one-tape machines there is a similar problem, which we settle as follows:

For $k>0$, let $L^k = \{w_1 \# w_1^R \$ w_2 \# w_2^R \$ \cdots \$ w_r \# w_r^R | 1 \leqq r \leqq k, w_i \in \{0, 1\}^* \text{ for } i = 1, \cdots, r\}$ and let $L_2 = \bigcup_{k \geqq 1} L^k$, the marked Kleene closure of the set of center-marked palindromes. Note that L_2 is accepted even by a real-time one-pushdown deterministic machine. Thus the following theorem affirmatively answers our question.

THEOREM 4. *L_2 is accepted by no real-time one-tape reversal-bounded nondeterministic machine.*

As was noted above, it is known that real-time deterministic machines are better than similar machines which are reversal-bounded. The proof given in [4] does not, however, yield a specific language over a two-symbol alphabet that cannot be accepted by the latter. An alternative proof follows from Theorem 5.

THEOREM 5. *L_2 is accepted by no real-time deterministic reversal-bounded machine.*

The proofs of Theorems 4 and 5 actually show that L^{k+1} cannot be accepted by a machine which makes only k turns. Thus, they actually establish a hierarchy which corresponds to the number of reversals. The proofs of Theorems 1 and 4 use a refinement of the bottleneck method introduced by Rabin [7].

2. Two basic lemmas. There are two well-known tricks that use a crossing sequence argument for one-tape (deterministic or nondeterministic) Turing machines, in which the input tape is the same as the working tape [5, Lemmas 10.1 and 10.2].

Trick 1 (*cut and paste*). Assume there are two accepting computations by machine M on inputs $x = x_1x_2$ and $y = y_1y_2$ with identical crossing sequences at the boundaries between x_1 and x_2 and between y_1 and y_2. Then M also accepts x_1y_2 (and y_1x_2).

Trick 2 (*cut out*). Assume there is an accepting computation by machine M on $x = x_1x_2x_3$ with two identical crossing sequences at the boundaries between x_1 and x_2 and between x_2 and x_3. Then M also accepts x_1x_3.

The two basic lemmas of this section are just the analogous tricks for real-time one (working) tape (deterministic or nondeterministic) Turing machines. Recall that we consider here a Turing machine with a separate input tape. We now consider crossing sequences on the working tape. (At each step the input head reads one symbol and moves right.) Note that any crossing sequence at a given boundary on the working tape defines a partition of the input string x into segments $x = x_1 \cdots x_s$. Each time this boundary is crossed, a new segment is determined. Similarly, a pair of crossing sequences defines a partition of the input string x into segments $x = x_1y_1 \cdots x_sy_sx_{s+1}$, where the y's are the segments of the input that correspond to the time intervals, during which the machine scans the part of the working tape between the two boundaries.

LEMMA 1 (*cut and paste*). *Assume there are two accepting computations by a real-time one-tape machine M on inputs $x = x_1x_2 \cdots x_k$ and $y = y_1y_2 \cdots y_k$ with two identical crossing sequences, and that the k segments of x and y are defined by each crossing sequence. Then M also accepts $x_1y_2x_3y_4 \cdots$ (and $y_1x_2y_3x_4 \cdots$).*

LEMMA 2 (*cut out*). *Assume there is an accepting computation by a real-time one-tape machine M on input $x = x_1y_1 \cdots x_sy_sx_{s+1}$ with two identical crossing sequences, where these crossing sequences define the partition of x. Then M also accepts $x' = x_{j_1}x_{j_2} \cdots x_{j_{k+1}}$, where $j_1, \cdots, j_{k+1}$ is a permutation of $\{1, 2, \cdots, k+1\}$.*

Remark. The permutation in Lemma 2 is $j_1 = 1, j_{k+1} = k+1$ and for $i \geqq 1$, $x_{j_{2i+1}}(x_{j_{2i}})$ is the part of the input that starts when M crosses the first (second) boundary from right to left (left to right) for the ith time and ends when M crosses this boundary back to the part of the working tape between the two boundaries. In Fig. 1, $x = x_1y_1 \cdots x_5y_5x_6$ and $x' = x_1x_3x_2x_5x_4x_6$.

The proofs of both of the lemmas are immediate. Rabin [7] used the first lemma to show that two tapes are better than one in the deterministic case. We do not know whether the second lemma has ever been used before. However, it is actually a special case of the first lemma with $x = y$.

In the next section we consider nondeterministic machines. For every input accepted by such a machine, we arbitrarily fix an accepting computation and refer to it as the accepting computation that corresponds to that input.

3. The proofs. We first sketch the proof of Theorem 1. Assume, to the contrary, that M is a one-tape nondeterministic Turing machine that accepts L_1 in real-time. Take $m = 2^n$ for some n and consider the strings in L_1 of size $2m+1$. A simple counting argument shows that for at least one of these strings, call it y, the size of

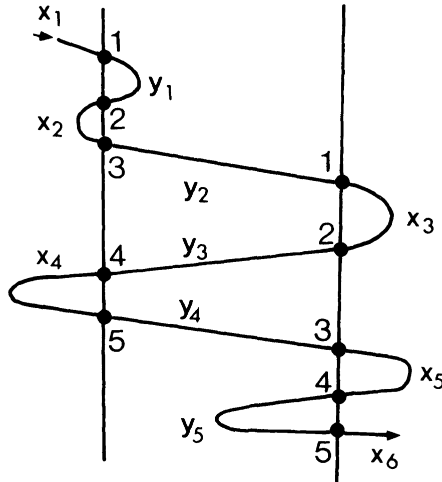


FIG. 1

the working tape used must be at least $m/(2 \log r)$,[1] where r is the size of the working tape alphabet of M. We now concentrate on the accepting computation of M on input y. We fix a parameter l and consider blocks of the working tape of size l. Another counting argument shows that there is a block A_{i_0} on the working tape that is not scanned much during the accepting computation of M on y. A third counting argument implies that there must be two boundaries in A_{i_0} that have the same crossing sequences (see Fig. 2). Lemma 2 implies that M accepts a shorter input. A contradiction follows from the fact that the length of that input cannot be $2 \cdot 2^k + 1$ for any k.

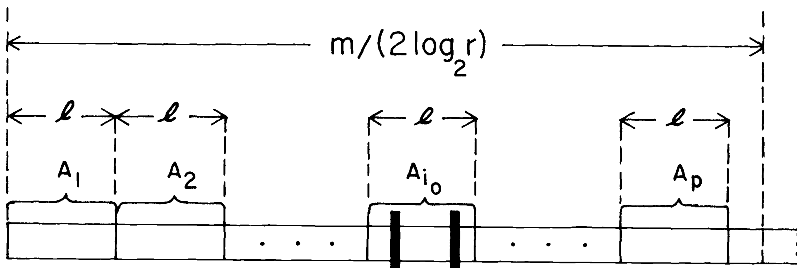


FIG. 2. *The working tape of M.*

Proof of Theorem 1. Assume to the contrary that L_1 is recognized by a real-time one-tape nondeterministic machine M with a set of internal states Q and with a working tape alphabet of size r, $r = 2^s$. For a sufficiently large m, where $m = 2^n$, let $S = \{x \# x \mid x \# x \in L_1 \text{ and } |x| = m\}$. For every string y in S, we consider the time when M scans the $\#$ in the accepting computation of M on y, and denote the string on the working tape by $w(y)$.

FACT 1. *There is a string y in S with $|w(y)| \geqq m/(2 \log r)$.*

Proof. A configuration of M is a triple (w, i, q), where w is the contents of the working tape, i is the position of the head on the working tape and q is the internal state. We consider in the accepting computation of M on a string in S the configuration of M when it scans $\#$. For two strings in S, $x_1 \# x_1$ and $x_2 \# x_2$, the corresponding configurations of M must be distinct. (Otherwise it would accept $x_1 \# x_2$). The number

[1] All logarithms in this paper are base 2.

of such configurations with $|w| < m/(2 \log r)$ is at most $r^{m/(2 \log r)+1}(m/(2 \log r)) \cdot |Q| < 2^m$ for large enough m. The number of strings in S is exactly 2^m. Consequently, there is y in S with $|w(y)| \geqq m/(2 \log r)$. □

From now on we consider the accepting computation of M on y. Let

$$l = 2|Q|^{10 \log r+1} + 2, \tag{1}$$

and divide the part of the working tape used by M into blocks of size l: $A_1, A_2, \cdots$. If the last segment is shorter than l, it is not a block.

FACT 2. *There is a block A_{i_0} such that the number of steps in which A_{i_0} is scanned is at most $5l \log r$.*

Proof. By the choice of y, the final length of the working tape of M is at least $m/(2 \log r)$. Consequently, the number p of blocks that M scans is at least $m/(2l \log r) - 1$. But M spends on y exactly $2m+1$ steps. So there is a block A_{i_0} such that the number of steps that it is scanned is at most $(2m+1)/p$. For a sufficiently large m, this number is at most $5l \log r$. □

The number of the crossing boundaries in A_{i_0} with crossing sequences of length of at most $10 \log r$ is at least $l/2$, because the sum of the lengths of all crossing sequences for boundaries in A_{i_0} is at most $5l \log r$. Since the number of all crossing sequences of length at most $10 \log r$ is at most $|Q|^{10 \log r+1} < l/2$ (by (1)), there are (in A_{i_0}) two crossing boundaries with the same crossing sequence (of length at most $10 \log r$). By Lemma 2 the string can be written in the form $y = x_1 y_1 x_2 y_2 \cdots x_s y_s x_{s+1}$, and M also accepts $y' = x_{j_1} \cdots x_{j_{s+1}}$, where $(j_1, \cdots, j_{s+1})$ is a permutation of $(1, \cdots, s+1)$. The y_i's are segments of the input read when M scans A_{i_0} (between the two boundaries). By the choice of A_{i_0}, $0 < \sum_{i=1}^{s} |y_i| \leqq 5l \log r$. But for a large enough m, $m = 2^n$,

$$2^{n+1} + 1 = |y| > |y'| = |y| - \sum_{i=1}^{s} |y_i| \geqq 2^{n+1} + 1 - 5l \log r > 2^n + 1.$$

So, $y' \notin L_1$—a contradiction. □

Proof of Theorem 2. The proof of Theorem 1 with some minor modifications shows that L_1' also is not accepted by a real-time one-tape nondeterministic Turing machine. L_1' is accepted by a real-time deterministic machine M with two pushdown stores (*pds*'s) as follows: For input $x \# y$, M uses its two pushdown stores to check whether $|x| = 2^n$ and simultaneously to store x in a special way. For $i \geqq 0$, let x_i be the prefix of x of length 2^i, and assume $x_{i+1} = x_i y_i$. M works in phases. In phase 0 it reads x_0 (the first symbol of x) and stores it in *pds* 1. Inductively, at the end of phase i M has completed reading x_i and it has stored it in a shuffled way in one of its *pds*'s. We denote this string *shuf* (x_i). Initially, *shuf* $(x_0) = x_0$. Reading y_i one symbol at a time, popping out one symbol of *shuf* (x_i) at a time and storing both symbols at the other *pds*, M generates *shuf* (x_{i+1}). Formally, if *shuf* $(x_i) = \alpha_{2^i} \cdots \alpha_1$ and $y_i = \beta_1 \cdots \beta_{2^i}$, then *shuf* $(x_{i+1}) = \alpha_1 \beta_1 \alpha_2 \beta_2 \cdots \alpha_{2^i} \beta_{2^i}$. (We use the convention that the rightmost symbol is the top symbol of the *pds*.) The test $|x| = 2^n$ is immediate: $|x| = 2^n$ if M scans the # at the end of a phase (when one of its *pds*'s is empty). To check whether $y = x^R$, M reads y one symbol at a time popping out two symbols from the *pds* comparing the first to the symbol of y and sorting the second in the second *pds*. Each time the first *pds* becomes empty, M changes the roles of the two *pds*'s. □

Remark. The machine we use in the proof of Theorem 2 (and the one in the proof of Theorem 3) reads an input symbol every two steps. Using a larger alphabet in the *pds* (pairs of input symbols), it can be simulated by a machine that reads an input symbol every step.

Proof of Theorem 3. We describe a real-time nondeterministic two-pushdown one-reversal machine M that accepts L_1'. On input $x \# y$, M pushes x into its first *pds*. Simultaneously, it nondeterministically pushes, on a second track of *pds* 1 and on *pds* 2, a collection of blocks $A_r, A_{r-1}, \cdots, A_1, A_0$. (See Fig. 3.) After reading the $\#$, M simultaneously checks (i) if $y = x^R$, and (ii) if $|A_0| = |A_1| = 1$ and $|A_{i+1}| = 2|A_i|$ (comparing $|A_{i+1}|$ in *pds* 1 with $|A_i|$ in *pds* 2). Note that (ii) implies that $|x| = 2^r$. □

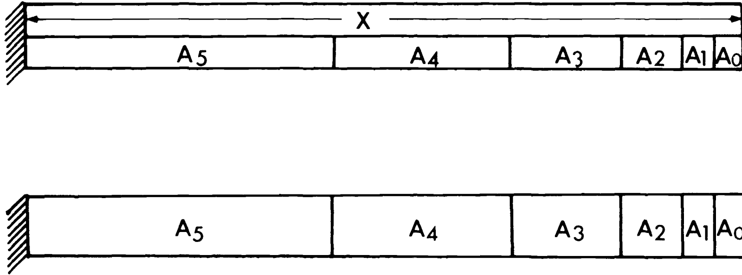


FIG. 3. *The two pds's of M.*

We now sketch the proof of Theorem 4. Assume, to the contrary, that M is a one-tape Turing machine that accepts L_2 in real-time and makes at most k reversals. We restrict attention to a subset of L^{k+1} with $|w_i| = n$ for all i. We choose an n large enough so that the counting arguments mentioned below are valid. The first counting argument implies that for a large subset S of the subset mentioned above all strings in S agree on w_i for $i \neq i_0$ and during reading $w_{i_0} \# w_{i_0}^R$ there is no head reversal. We now assume that the head of the working tape does not move during a certain number N of steps. An easy counting argument shows that a pumping can be applied to yield a contradiction. Consequently, when M accepts a string in S, the largest prefix of $w_{i_0}^R$ during which the head of the working tape does not move is bounded above by N. A counting argument that uses the size of S, the number of such prefixes and the fact that M makes at most k reversals implies that there are two strings in S with identical prefixes but different suffices (or $w_{i_0}^R$), such that the two crossing sequences defined by the times immediately after M reads the entire prefix are the same. Lemma 1 implies that M must accept another string not in L_2. This contradiction completes the proof.

Proof of Theorem 4. We assume, to the contrary, that L_2 is accepted by a real-time one-tape reversal-bounded nondeterministic machine M with a set of internal states Q, with a working tape alphabet of size r and with at most k reversals. Choose a sufficiently large n, so that

$$2^n/(k+1) \geqq 2^{r|Q|+1} \cdot k \cdot |Q|^{k+1} + 1, \tag{2}$$

and let $\hat{S} = \{w_1 \# w_1^R \$ \cdots \$ w_{k+1} \# w_{k+1}^R | w_i \in \{0, 1\}^n \text{ for } i = 1, \cdots, k+1\}$.

FACT 3. *There is a subset S of $\hat{S}$ and $1 \leqq i_0 \leqq k+1$ such that*:

(a) $|S| \geqq 2^{r|Q|+1} \cdot k \cdot |Q|^{k+1} + 1$;

(b) *for all w, w' in S, $w = w_1 \# w_1^R \$ \cdots \$ w_{k+1} \# w_{k+1}^R$ and $w' = w_1' \# w_1'^R \$ \cdots \$ w_{k+1}' \# w_{k+1}'^R$, $w_i = w_i'$ for $1 \leqq i \leqq k+1$ and $i \neq i_0$*;

(c) *for all strings in S, there is no head reversal when M reads $w_{i_0} \# w_{i_0}^R$ in the corresponding accepting computations.*

Proof. There are $2^{(k+1)n}$ strings in $\hat{S}$. For each one of them there is $1 \leqq i \leqq k+1$ such that there is no head reversal when M reads $w_i \# w_i^R$. Therefore, there are $1 \leqq i_0 \leqq k$ and a subset S_1 of $\hat{S}$ of size $\geqq 2^{(k+1)n}/(k+1)$ such that there is no head

reversal when M reads $w_{i_0} \# w_{i_0}^R$ for all strings in S_1. There are 2^{kn} possible k-tuples $(w_1, \cdots, w_{i_0-1}, w_{i_0+1}, \cdots, w_{k+1})$ with $w_i \in \{0, 1\}^n$. Hence, there is a subset S of S_1 of size $\geqq (2^{(k+1)n}/(k+1))/2^{kn}$ that satisfies (b) and (c). It also satisfies (a) because of (2). □

We choose $x = w_1 \# w_1^R \$ \cdots \$ w_{i_0-1} \# w_{i_0-1}^R \$$ ($x = \varepsilon$, the empty string, if $i_0 = 1$) $y = \$ w_{i_0+1} \# w_{i_0+1}^R \$ \cdots \$ w_{k+1} \# w_{k+1}^R$ ($y = \varepsilon$ if $i_0 = k+1$). Hence, each string in S is of the form $xw_{i_0} \# w_{i_0}^R y$ (with the same x and y).

FACT 4. *For all strings in* S, *during the* $r|Q|+1$ *steps after reading* $w_{i_0}\#$, *the working head of* M *must move* (*left or right*) *at least once.* (*This count includes the step that follows reading the* $\#$.)

Proof. Otherwise there is $xw_{i_0} \# w_{i_0}^R y$ in S such that the working head of M does not move for $r|Q|+1$ steps after reading $w_{i_0}\#$. Using a pumping technique (M eventually repeats a state with the same symbol scanned by the working head, also $n > r|Q|+1$ by (2)), $w_{i_0}^R = v_1 v_2 v_3$, $v_2 \neq \varepsilon$, and M also accepts $xw_{i_0} \# v_1 v_2^m v_3 y$ for all $m > 0$—a contradiction. □

Hence, every string in S can be written in the form $xv_1v_2 \# v_2^R v_1^R y$, where $v_1 v_2 \in \{0, 1\}^n$, $|v_2^R| \leqq r|Q|$, and after reading the $\#$ the head of M moves for the first time immediately after it completes reading $\# v_2^R$. (Possibly $v_2^R = \varepsilon$.) This head movement defines a boundary on the working tape of M, a crossing sequence at that boundary, and $1 \leqq p \leqq k$ which denotes the pth crossing of this boundary.

The number of different crossing sequences of M is at most $|Q|^{k+1}$, and the number of possible v_2^R's is $2^{r|Q|+1}$. Hence by (a) of Fact 3, there are two strings in S: $w = xv_1v_2 \# v_2^R v_1^R y$ and $w' = xv_1'v_2' \# v_2'^R v_1'^R y$ with $v_2 = v_2'$, $v_1' \neq v_1$, with the *same* crossing sequences at the corresponding boundaries and the *same* p. By (c) of Fact (3), M crosses the corresponding boundary exactly once while reading $w_{i_0} \# w_{i_0}^R$ for both w and w'. By Lemma 1, M also accepts two mixed versions of w and w'. One is of the form $\tilde{x} v_1 v_2 \# v_2^R v_1'^R \tilde{y}$ and one is of the form $\tilde{\tilde{x}} v_1' v_2' \# v_2'^R v_1^R \tilde{\tilde{y}}$. Both strings are not in L_2—a contradiction. □

Proof of Theorem 5. We show that L^{k+1} cannot be accepted by a real-time deterministic reversal-bounded machine which makes no more than k reversals. Assume, to the contrary, that L^{k+1} is accepted by such a machine with t tapes, a working tape alphabet size r and a set of internal states Q. Choose a large enough n such that

$$2^n/(k+1) > |Q| r^t (2n+1)^t + 1 \tag{3}$$

and consider $\hat{S}$ as in the proof of Theorem 4. As in Fact 3, there is a subset S of $\hat{S}$ and $1 \leqq i_0 \leqq k+1$ such that:

(a) $|S| \geqq |Q| r^t (2n+1)^t + 1$, with (b) and (c) as in Fact 3.

Note that the strings in S are of the form $xw_{i_0} \# w_{i_0}^R y$, and M accepts $xw_{i_0} \# w_{i_0}^R$. By (a) there are two strings in S, $xw_{i_0} \# w_{i_0}^R y$ and $xw_{i_0}' \# w_{i_0}'^R y$, such that M scans the same t-tuple of symbols at the same positions on the working tapes and is in the same state when it reads the $\#$ that follows w_{i_0} and the one that follows w_{i_0}'. (Note that after reading x in both computations M is in the same configuration, and during reading w_{i_0} or w_{i_0}' each of its heads can move at most n positions.) Since the heads of M do not change direction when it reads $w_{i_0} \# w_{i_0}^R$ (and $w_{i_0}' \# w_{i_0}'^R$), M must also accept $xw_{i_0} \# w_{i_0}'^R$—a contradiction. □

Note that we have used the fact that M is deterministic when we derived an accepting computation on $xw_{i_0} \# w_{i_0}^R$ from (an initial segment of) an accepting computation on $xw_{i_0} \# w_{i_0}^R y$.

4. Conclusion. We now list some open problems. We have solved Questions 1 and 2 for unrestricted and reversal-bounded machines. We still do not know the answer to Question 3: Are three (reversal-bounded) pushdown stores better than two? We have shown that one-tape nondeterministic machines are better than one-tape nondeterministic reversal-bounded machines. We still cannot prove a similar result for multi-tape machines.

We have considered only real-time computations. Again, each one of Questions 4 to 7 below has two versions (one for the unrestricted case and one for the reversal-bounded case), and all machines are nondeterministic.

4. Are linear-time one-tape machines better than real-time one-tape machines?
5. Are linear-time two-tape machines better than linear-time one-tape machines?
6. Are linear-time two-pushdown machines better than linear-time one-tape machines?
7. Are linear-time three-pushdown machines better than linear-time two-pushdown machines?

Finally, we consider briefly two other models of Turing machines. Recall that in the model we used, the machine has a one-way read-only input. In the two other models the machine has (i) a two-way read-only input, and (ii) an input tape which is also a working tape.

First, consider the model with a two-way read-only input. Note that for real-time machines two-way is no better than one-way simply because the input head has no time to go to the left. It is easy to see that the four families of languages in the introduction are the same in this model too. Consequently, Questions 1 to 3 arise, and our answers to Questions 1 and 2 also apply to this model. Two-way access to the input seems to make a difference for linear-time computations. It is easy to see that L_1 can be accepted by a deterministic (!) linear-time one-tape machine, and that L_2 can be accepted by such a machine which is also reversal-bounded. So both versions of Questions 4 are answered in the affirmative in this model.

Now, consider machines which are allowed to use their input tape as a working tape. For this model the questions about pushdown stores are meaningless. We also do not consider here reversal-bounded machines. It is well-known [5, Thm. 10.7] that such machines with one tape are very weak: the language $L=\{x \# x^R | x \in \{0, 1\}^*\}$ requires cn^2 time on a nondeterministic one-tape machine. On the other hand, even a deterministic one-tape machine accepts L in real-time in any one of the previous models. Note that for real-time machines, the ability to write on the input tape is useless, and, for all k, $k+1$ tapes in this model are equivalent to k tapes in the other models. Consequently, for any k, k tapes are no better than three, three tapes are better than two, and two tapes are better than one (real-time one-tape machines accept only regular sets). The ability to write on the input tape seems useful in linear-time computations. It was observed in [6] that in this model any k-tape nondeterministic linear-time machine can be simulated by a two-tape linear-time machine. Consequently, Questions 4 and 5 are obviously answered affirmatively.

Note added in proof. We have recently twice improved Theorem 1. In [9] (together with W. J. Paul and R. Reischuk) we showed that any one-tape on-line nondeterministic Turing machine that accepts L_1 requires $\Omega(n \log\log n)$ time. More recently we improved this lower bound to $\Omega(n \log n)$. Note that L_1 can easily be accepted by a *deterministic* one-tape on-line Turing machine in time $O(n \log n)$.

REFERENCES

[1] S. O. AANDERAA, *On k-tape versus $(k-1)$-tape real-time computation*, Complexity of Computation (SIAM-AMS Proc. 7), R. M. Karp, ed., American Mathematical Society, Providence, RI, 1974, pp. 75–96.

[2] R. V. BOOK AND S. A. GREIBACH, *Quasi-realtime languages*, Math. Systems Theory, 4 (1970), pp. 97–111.

[3] R. V. BOOK, M. NIVAT AND M. PATERSON, *Reversal-bounded acceptors and intersection of linear languages*, this Journal, 3 (1974), pp. 283–295.

[4] R. V. BOOK AND C. K. YAP, *On the computational complexity of reversal-bounded machines*, Proc. 7th ICALP, 1977, pp. 111–119.

[5] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[6] J. I. SEIFERAS, M. J. FISCHER AND A. R. MEYER, *Separating nondeterministic time complexity classes*, J. Assoc. Comput. Mach., 1 (1978), pp. 146–167.

[7] M. O. RABIN, *Real-time computation*, Israel Journal of Mathematics, 4 (1963), pp. 203–211.

[8] M. K. VALIEV, *Certain estimates of the time of computations on Turing machines with an input*, *Kibernetika*, 6, 6 (1970), pp. 26–32; Cybernetics, 6, 6 (1973), pp. 734–740.

[9] P. DURIS, Z. GALIL, W. J. PAUL AND R. REISCHUK, *Two nonlinear lower bounds*, Proc. 14th STOC, 1983, pp. 127–132.

SIAM J. COMPUT.
Vol. 13, No. 2, May 1984
© 1984 Society for Industrial and Applied Mathematics
002

COMPUTATION OF MATRIX CHAIN PRODUCTS. PART II*

T. C. HU† AND M. T. SHING†

Abstract. This paper considers the computation of matrix chain products of the form $M_1 \times M_2 \times \cdots \times M_{n-1}$. If the matrices are of different dimensions, the order in which the matrices are computed affects the number of operations. An optimum order is an order which minimizes the total number of operations. Some theorems about an optimum order of computing the matrices have been presented in Part I [SIAM J. Comput., 11 (1982), pp. 362–373]. Based on those theorems, an $O(n \log n)$ algorithm for finding the optimum order is presented here.

1. Introduction. In Part I of this paper [6], we have transformed the matrix chain product problem into the optimum partitioning problem and have stated several theorems about the optimum partitions of an n-sided convex polygon. Some theorems in Part I can be strengthened and are stated here (the detailed proofs are in [7]).

THEOREM 1. *For every choice of $V_1, V_2, \cdots$ (as prescribed in Part I), if the weights of the vertices of the n-gon satisfy the following condition,*

$$w_1 = w_2 = \cdots = w_k < w_{k+1} \leqq \cdots \leqq w_n$$

for some k, $3 \leqq k \leqq n$, then every optimum partition of the n-gon contains the k-gon $V_1 - V_2 - \cdots - V_k$. Furthermore, if $k = 2$ in the above condition, i.e. $w_1 = w_2 < w_3 \leqq w_4 \leqq \cdots \leqq w_n$, then every optimum partition of the n-gon must contain a triangle $V_1 V_2 V_p$ for some vertex V_p with weight equal to w_3.

Note that if $w_1 = w_2 < w_3 < w_4 \leqq \cdots \leqq w_n$, then every optimum partition must contain the triangle $V_1 V_2 V_3$ since there is a unique choice of V_3.

Now, whenever we have three or more vertices with weights equal to w_1 in the n-gon, we can decompose the n-gon into subpolygons by forming the k-gon in the first part of Theorem 1. The partition of the k-gon can be arbitrary, since all vertices of the k-gon are of equal weight. For any subpolygon with two vertices of weights equal to w_1, we can always apply the second part of Theorem 1 and decompose the subpolygon into smaller subpolygons. Hence, we have only to consider the polygons with a unique choice of V_1; i.e., each polygon has only one vertex with weight equal to w_1.

Because of the above theorem, Theorems 1 and 3 of Part I can be generalized as follows.

THEOREM 2. *For every choice of $V_1, V_2, \cdots$ (as prescribed in Part I), if the weights of the vertices satisfy the condition*

$$w_1 < w_2 \leqq w_3 \leqq \cdots \leqq w_n,$$

then $V_1 - V_2$ and $V_1 - V_3$ exist in every optimum partition of the n-gon.

THEOREM 3. *Let V_x and V_z be two arbitrary vertices which are not adjacent in a polygon, and V_w be the smallest vertex from V_x to V_z in the clockwise manner ($V_w \neq V_x$, $V_w \neq V_z$), and V_y be the smallest vertex from V_z to V_x in the clockwise manner ($V_y \neq V_x$, $V_y \neq V_z$). This is shown in Fig. 1. Assume that $V_x < V_z$ and $V_y < V_w$. The necessary condition for $V_x - V_z$ to exist as an h-arc in any optimum partition is*

$$w_y < w_x \leqq w_z < w_w.$$

* Received by the editors August 4, 1981, and in final revised form February 7, 1983. This research was supported in part by the National Science Foundation under grant MCS-77-23738 and in part by the U.S. Army Research Office under grant DAAG29-80-C-0029.

† University of California at San Diego, La Jolla, California 92093.

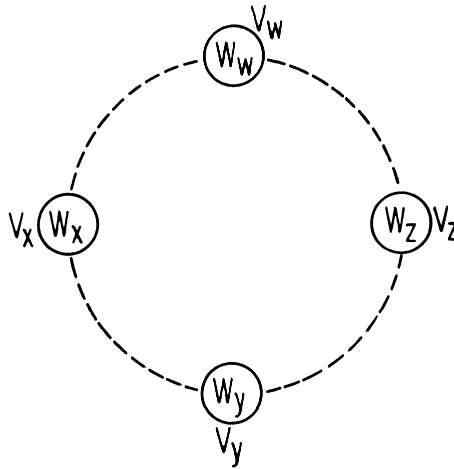


FIG. 1

We shall use "the l-optimum partition" to mean "the lexicographically smallest optimum partition." Based on these theorems, we now present algorithms for finding the unique l-optimum partition.

Using the same notation as in Part I of this paper [6], we can assume that we have uniquely labelled all vertices of the n-gon. A partition is called a *fan* it is consists of only v-arcs joining the smallest vertex to all other vertices in the polygon. We shall denote the fan of a polygon $V_1 - V_b - V_c - \cdots - V_n$ by Fan $(w_1|w_b, w_c, \cdots, w_n)$. The smallest vertex V_1 is called the *center* of the fan.

We define a vertex as a *local maximum* vertex if it is larger than its two neighbors and define a vertex as a *local minimum* vertex if it is smaller than its two neighbors. A polygon is called a *monotone* polygon if there exist only one local maximum and one local minimum vertex. We shall first give an $O(n)$ algorithm for finding the l-optimum partition of a monotone polygon and then give an $O(n \log n)$ algorithm for finding the l-optimum partition of a general convex polygon.

2. Monotone basic polygon. In this section, let us consider the optimum partition of a monotone polygon, i.e. a polygon with only one local minimum vertex and one local maximum vertex. It follows from Theorems 1 and 2 that we can consider a monotone basic polygon only. (A polygon having V_1 adjacent to V_2 and V_3 by sides is called a *basic* polygon.) The understanding of this special case is necessary in finding the optimum partition of a general convex polygon.

Consider a monotone basic n-gon $V_1 - V_2 - V_c - \cdots - V_3$, the fan of the polygon is denoted by

$$\text{Fan } (w_1|w_2, w_c, \cdots, w_3)$$

where the smallest vertex V_1 is the center of the fan.

The definition of a fan can also be applied to subpolygons as well. For example, if V_2, V_3 are connected in the basic n-gon and V_2 becomes the smallest vertex in the $(n-1)$-sided subpolygon, the partition formed by connecting V_2 to all vertices in the $(n-1)$-gon is denoted by

$$\text{Fan } (w_2|w_c, \cdots, w_3).$$

LEMMA 1. *If none of the potential h-arcs appears in the l-optimum partition of the n-gon, the l-optimum partition must be the fan of the n-gon.*

Proof. Omitted. See [7] for details. □

A potential h-arc will dissect a polygon into two parts, and the subpolygon which contains the larger vertices is called the *upper subpolygon*. Let $V_i - V_j$ and $V_p - V_q$ be two potential h-arcs of any n-gon. We say that $V_p - P_q$ is *above* (or *higher than*) $V_i - V_j$ (and $V_i - V_j$ is *below*, or *lower than*, $V_p - V_q$) if the upper subpolygon of $V_i - V_j$ contains the upper subpolygon of $V_p - V_q$.

Let P be the set of all potential h-arcs in a monotone basic n-gon. P can have at most $n-3$ arcs.

LEMMA 2. *For any two arcs in P, say $V_i - V_j$ and $V_p - V_q$, we must have either $V_i - V_j$ above $V_p - V_q$ or $V_p - V_q$ above $V_i - V_j$.*

Proof. See [7] for details. □

We can actually show this ordering of potential h-arcs pictorially by drawing a monotone basic polygon in such a way that the local maximum vertex is always at the top and the local minimum vertex is at the bottom. Then a potential h-arc $V_p - V_q$ is physically above another potential h-arc $V_i - V_j$ if the upper subpolygon of $V_i - V_j$ contains the upper subpolygon of $V_p - V_q$. From the definition of the upper subpolygon and the monotone property, we can see that $\max(w_i, w_j) < \min(w_p, w_q)$ if $V_p - V_q$ is above $V_i - V_j$.

Consider the monotone basic n-gon which is shown symbolically in Fig. 2. V_n is the local maximum vertex and $V_i - V_j$, $V_p - V_q$ are potential h-arcs of the monotone basic n-gon. The subpolygon $V_i - \cdots - V_p - V_q - \cdots - V_j$ which is formed by two potential h-arcs $V_p - V_q$ and $V_i - V_j$ and the sides of the n-gon from V_i to V_p and from V_q to V_j in the clockwise direction is said to be *bounded above* by the potential h-arc $V_p - V_q$ and *bounded below* by the potential h-arc $V_i - V_j$, or simply as the subpolygon between $V_i - V_j$ and $V_p - V_q$ for brevity.

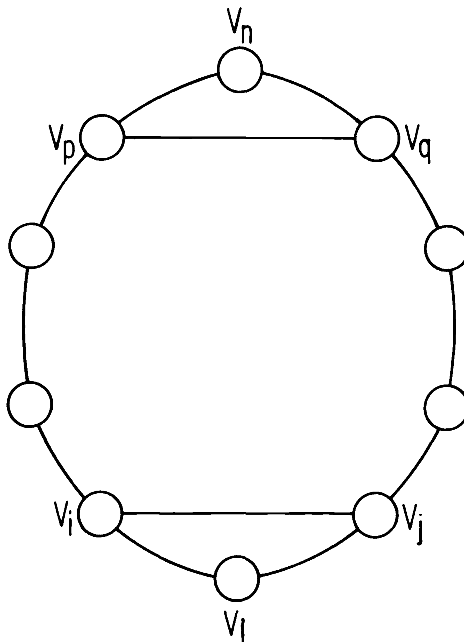


FIG. 2

LEMMA 3. *Any subpolygon which is bounded by two potential h-arcs of the monotone basic n-gon is itself a monotone polygon.*

Proof. See [7] for details. □

LEMMA 4. *Any potential h-arc of a subpolygon bounded above and below by two potential h-arcs of the monotone basic n-gon is also a potential h-arc of the monotone basic n-gon.*

Proof. See [7] for details. □

We can now summarize what we have discussed. If there is no h-arc in the l-optimum partition of a monotone basic n-gon, the l-optimum partition must be a fan. Otherwise, the h-arcs in the l-optimum partition are all layered, one above another. If we consider the local maximum vertex V_n and the local minimum vertex V_1 as two degenerated h-arcs, then the l-optimum partition of a monotone basic n-gon will contain one or more monotone subpolygons, each bounded above and below by two h-arcs and the l-optimum partition of each of these monotone subpolygons is a fan. Then, in finding the l-optimum partition of a monotone basic polygon, we have only to consider those partitions which contain one or more potential h-arcs and each of the subpolygons between two potential h-arcs is partitioned by a fan.

Since there are at most $n-3$ nondegenerated potential h-arcs in a monotone basic n-gon, there will be at most 2^{n-3} such partitions and we can divide all these partitions into $(n-2)$ classes by the number of nondegenerated potential h-arcs a partition contains. These classes are denoted by $H_0, H_1, \cdots, H_{n-3}$ where the subscript indicates the number of nondegenerated potential h-arcs in each partition of that class.

There is no potential h-arc in the partitions in the class H_0. Hence the class consists of only one partition, namely the fan

$$\text{Fan}\,(w_1|w_2, \cdots, w_3).$$

In the class H_1, each partition has one nondegenerated potential h-arc. Once the potential h-arc is known, the rest of the arcs must all be vertical arcs forming two fans, one in each subpolygon.

Two typical partitions in H_1 of a monotone basic polygon are shown in Fig. 3. In Fig. 3a, there is one nondegenerated potential h-arc, $V_c - V_i (V_c < V_i)$. The upper

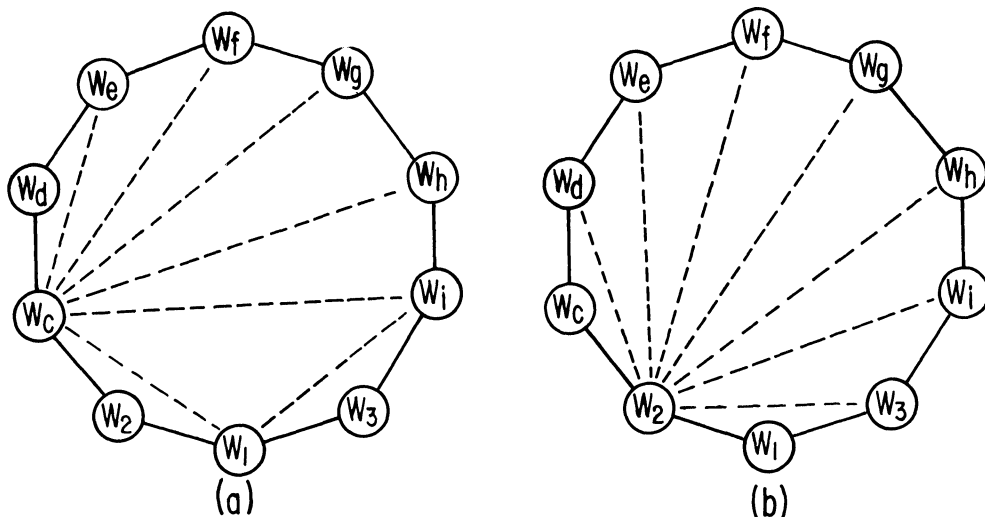


FIG. 3. *Two typical partitions in H_1 of a monotone 10-gon.*

subpolygon is a fan

$$\text{Fan}\,(w_c|w_d, \cdots, w_i)$$

and the lower subpolygon is a fan

$$\text{Fan}\,(w_1|w_2, w_c, w_i, w_3).$$

In Fig. 3b, there is one potential h-arc, $V_2 - V_3$, and the upper subpolygon is a fan

$$\text{Fan}\,(w_2|w_c, \cdots, w_3)$$

and the lower subpolygon is a degenerated fan, a triangle.

The cost of the partition in Fig. 3b is

$$\begin{aligned}(1)\quad & w_1w_2w_3 + w_2(w_cw_d + w_dw_e + w_ew_f + w_fw_g + w_gw_h + w_hw_i + w_iw_3) \\ & \qquad = w_1w_2w_3 + w_2(w_c : w_3),\end{aligned}$$

where $w_c : w_3$ is the shorthand notation of the sum of adjacent products from w_c to w_3 in the clockwise direction.

Note that the cost of H_0 of the polygon shown in Fig. 3 is

$$\text{Fan}\,(w_1|w_2, \cdots, w_3) = w_1(w_2 : w_3). \tag{2}$$

The condition for (1) to be less than (2) is

$$\frac{w_2 \cdot (w_c : w_3)}{(w_2 : w_3) - w_2 \cdot w_3} < w_1.$$

Similarly, the condition for the partition in Fig. 3a to be less than H_0 is

$$\frac{w_c \cdot (w_d : w_i)}{(w_c : w_i) - w_c \cdot w_i} < w_1. \tag{3}$$

We say that a partition is said to be *l-optimal* among the partitions in a certain class (or several classes) if it is the lexicographically smallest partition among all the partitions with minimum cost in that class (or several classes). Hence, the l-optimum partition is l-optimal among all partitions in the classes H_0, $H_1, \cdots$, and H_{n-3}.

Now, assume that the l-optimal partition among all the partitions in $H_1, H_2, \cdots, H_{n-3}$ contains only one potential h-arc $V_i - V_k$, as shown in Fig. 4. (Note that $V_i - V_k$ will exist in this partition as an h-arc.) This partition will be the l-optimum partition of the monotone basic n-gon if it costs less than that of the fan in H_0. The condition that the partition with $V_i - V_k$ as the single h-arc costs less than H_0 is

$$\frac{w_i \cdot (w_j : w_k)}{(w_i : w_k) - w_i \cdot w_k} < w_1 \quad \text{if } w_i \leqq w_k$$

or

$$\frac{w_k \cdot (w_i : w_g)}{(w_i : w_k) - w_i \cdot w_k} < w_1 \quad \text{if } w_k < w_i.$$

Combining the two inequalities above, we have

$$\frac{C(w_i, \cdots, w_k)}{(w_i : w_k) - w_i \cdot w_k} < w_1 \tag{4}$$

where $C(w_i, \cdots, w_k)$ denotes the cost of the optimum partition of the subpolygon $w_i - w_j - \cdots - w_g - w_k$ and is equal to the cost of the fan in this case.

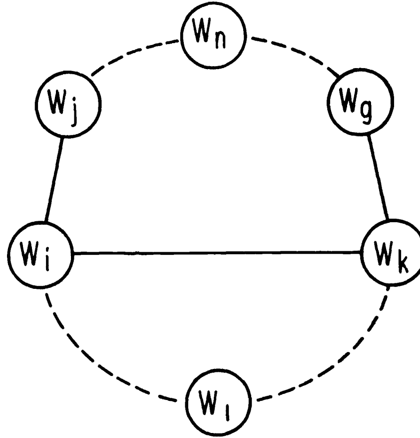


FIG. 4. *A monotone polygon with a single h-arc.*

An h-arc $V_i - V_k$ which divides a polygon into two subpolygons is called a *positive* arc with respect to the polygon if condition (4) is satisfied; i.e., the partition with the arc as the only h-arc and a fan in each of the two subpolygons costs less than the fan in the same polygon. Otherwise, it is called a *negative* arc with respect to the polygon.

When an n-gon is divided into subpolygons, an h-arc is defined as positive in a subpolygon if the cost of partition of the subpolygon with the h-arc as the only h-arc is less than the fan in the subpolygon.

Let us consider a partition with two h-arcs as shown in Fig. 5, and assume that this partition is l-optimal among all partitions in the classes $H_2, H_3, \cdots, H_{n-3}$.

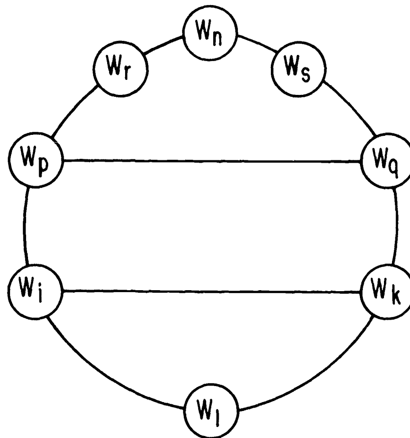


FIG. 5. *A monotone 8-gon with two h-arcs.*

If $V_i - V_k$ is positive with respect to the subpolygon $V_1 - V_i - V_p - V_q - V_k$, then the condition analogous to (4) is

$$\text{(5a)}\qquad \frac{C(w_i, w_p, w_q, w_k)}{\{(w_i : w_k) - [(w_p : w_q) - w_p : w_q]\} - w_i \cdot w_k} < w_1.$$

If $V_i - V_k$ is positive with respect to the whole polygon $V_1 - V_i - \cdots - V_n - \cdots - V_k$, then the condition is

$$\text{(5b)} \qquad \frac{C(w_i, w_p, w_r, w_n, w_s, w_q, w_k)}{(w_i : w_k) - w_i \cdot w_k} < w_1.$$

Note that condition (5b) implies (5a).

The condition for the arc $V_p - V_q$ to be positive with respect to the subpolygon $V_i - V_p - V_r - V_n - V_s - V_q - V_k$ is

$$\text{(6a)} \qquad \frac{C(w_p, w_r, w_n, w_s, w_q)}{(w_p : w_q) - w_p \cdot w_q} < \min(w_i, w_k).$$

If the arc $V_p - V_q$ is positive with respect to the whole polygon $V_1 - V_i - V_p - V_r - V_n - V_s - V_q - V_k$, it must satisfy

$$\text{(6b)} \qquad \frac{C(w_p, w_r, w_n, w_s, w_q)}{(w_p : w_q) - w_p \cdot w_q} < w_1.$$

Since $w_1 < \min(w_i, w_k)$, condition (6b) implies (6a).

Here, the presence of $V_i - V_k$ will divide the original polygon into two subpolygons where $V_p - V_q$ appears in the upper subpolygon. If $V_p - V_q$ is a positive arc with respect to the original polygon, then $V_p - V_q$ is certainly positive in the upper subpolygon. But if $V_p - V_q$ is positive in the subpolygon, the arc $V_p - V_q$ may become negative if $V_i - V_k$ is removed; i.e., $V_p - V_q$ becomes negative with respect to the original polygon.

Similarly, if the arc $V_i - V_k$ is positive with respect to a subpolygon, the arc $V_i - V_k$ may become negative if the arc $V_p - V_q$ is removed.

The preceding discussions can be summarized as:

THEOREM 4. *If an h-arc is positive with respect to a polygon then the arc is positive with respect to any subpolygon containing that arc. If an h-arc is positive with respect to a subpolygon, it may or may not be positive with respect to a larger polygon which contains the subpolygon.*

There are two *intuitive* approaches to finding the *l*-optimum partition of a monotone basic polygon. The first approach is to put in the potential *h*-arcs one by one. Each additional potential *h*-arc will improve the cost until the correct number of *h*-arcs is reached. Any further increase in the number of *h*-arcs will increase the cost. To introduce an *h*-arc into the polygon, we can test each potential *h*-arc (at most $n-3$) to see if it is positive with respect to the whole polygon. If yes, that positive arc must exist in the *l*-optimum partition, and the polygon will be divided into two subpolygons, each being a monotone polygon. We can repeat the whole process of testing positiveness of the *h*-arcs. The trouble is that all these arcs may be negative individually with respect to the whole polygon and yet H_0 may not be the optimum. For example, two arcs $V_i - V_j$ and $V_p - V_q$ may be negative individually with respect to the whole polygon, but the partition with both $V_i - V_j$, $V_p - V_q$ present at the same time may cost less than H_0, as shown in Fig. 6a. This shows that we cannot guarantee an optimum partition simply because no more potential *h*-arcs can be added one at a time.

The second approach is to put all the potential *h*-arcs in first, and then take out the potential *h*-arcs one by one, where each deletion will decrease the cost until the correct number of *h*-arcs is reached. Any further deletions will increase the cost. Unfortunately, even if all *h*-arcs are positive with respect to their subpolygon, the partition may not be optimum. In Fig. 6b, each *h*-arc is positive with respect to its

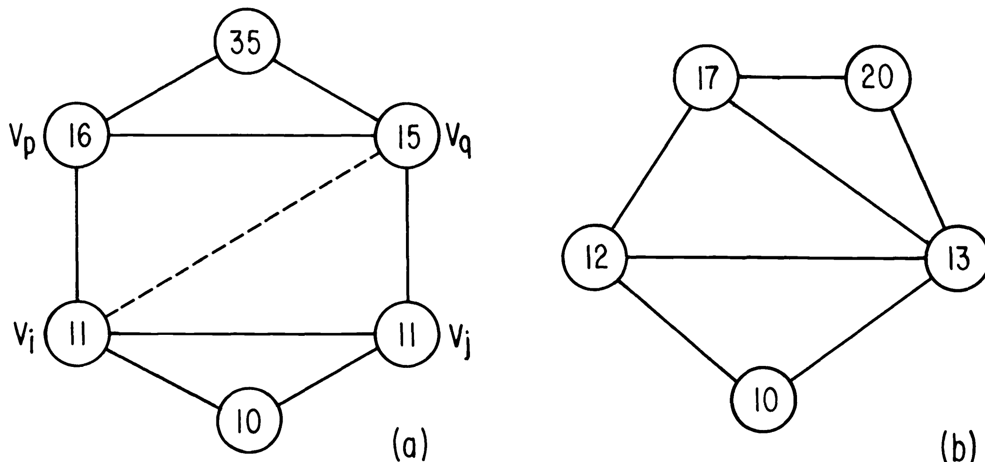


FIG. 6. *Counterexamples for the intuitive approaches.*

local subpolygon, but the partition is not optimum. (Note that positiveness of an h-arc in a quadrilateral is the same as stability. But the idea of stability applies to vertical arcs as well.) This means that we cannot guarantee an optimum partition simply because no h-arc can be deleted one at a time.

Let us outline the idea of an $O(n)$ algorithm for finding the l-optimum partition of a monotone basic polygon. First, we get all the potential h-arcs by the one-sweep algorithm. Then, we start from the highest potential h-arc and process each potential h-arc from the highest to the lowest. For each potential h-arc, we try to get the l-optimum partition of the upper subpolygon above that arc. The l-optimum partition in the subpolygon is obtained by comparing the cost of the l-optimal partition among the partitions of the upper subpolygon which contain one or more potential h-arcs with that of the fan in the upper subpolygon.

If we use the dynamic programming approach to find the l-optimum partition in the upper subpolygon of each potential h-arc, we need $O(n^3)$ operations to find the l-optimum partition of the whole monotone basic n-gon. Fortunately, there are some dependence relationships among these potential h-arcs. Hence, certain subsets of the potential h-arcs will either all exist or all disappear in the l-optimum partition of the monotone polygon. We shall be dealing with potential h-arcs most of the time, so we shall use "arcs" instead of "potential h-arcs" when there is no ambiguity.

Consider the monotone basic polygon shown symbolically in Fig. 7. There are three potential h-arcs, denoted by h_k, h_j and h_i. For any arc h_a, we shall use w_a, w'_a to denote the weights associated with the end vertices of the arc h_a. V_n is the local maximum vertex and V_1 is the local minimum vertex. Without loss of generality, we can assume $w_a \leqq w'_a$ for $a = i, j$ and k. Since we shall deal with subpolygons bounded by two potential h-arcs, let us use h_n for V_n and h_1 for V_1 (i.e., we consider these vertices as degenerated arcs). From Lemmas 1 and 3, the l-optimum partitions of the subpolygons bounded by two potential h-arcs (i.e. the white area of the polygon in Fig. 7) are all fans.

Assume (i) h_k is positive in the subpolygon bounded by h_n and h_j, but h_k is negative in the subpolygon bounded by h_n and h_i;

(ii) h_j is positive in the subpolygon bounded by h_k and h_i, but h_j is negative in the subpolygon bounded by h_k and h_1;

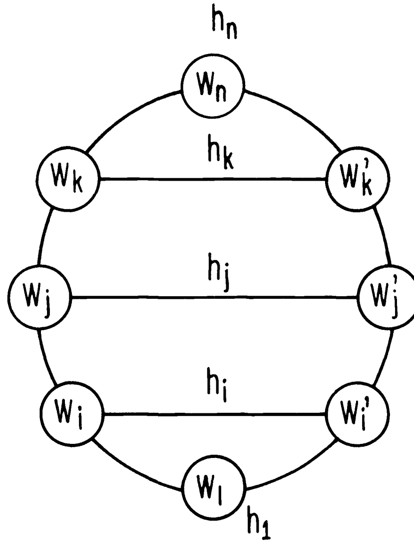


FIG. 7. *An octagon with three potential h-arcs.*

(iii) h_i is positive in the subpolygon bounded by h_j and h_1 only.

Then either the three arcs h_k, h_j, h_i all exist or no h-arcs exists in the optimum partition.

This shows that the existence of an h-arc depends on the existence of another h-arc.

In Fig. 7, the condition for h_k to be positive with respect to the whole polygon is (compare with the condition (5a))

$$\frac{C(w_k, w_n, w_k')}{(w_k : w_k') - w_k \cdot w_k'} < w_1. \tag{7}$$

The left-hand side of (7) is denoted by

$$S(h_k \backslash h_n)$$

and is called the *supporting weight* of the arc h_k with respect to the upper subpolygon bounded above by h_n.

The supporting weight of an arc h_k is an indicator of the existence of h_k in a subpolygon. To specify the subpolygon, we have to specify the arc above h_k, e.g. h_n in this case, and an arc below h_k. Once the upper subpolygon of h_k is specified, we can calculate the supporting weight of h_k since the left-hand side of (7) depends only on weights of vertices in the upper subpolygon. To find the arc below h_k which is the lower boundary of the subpolygon, we can use the supporting weight of h_k to test each arc h_i below h_k. (The h_i has two vertices with weights w_i and w_i'.)

If $S(h_k \backslash h_n) < \min(w_i, w_i')$ then h_k will exist in the subpolygon between h_i and h_n. Otherwise, h_k cannot exist in the subpolygon.

Let h_i, h_j and h_k be three potential h-arcs where h_j lies below h_k and above h_i. Let

$$S(h_i \backslash h_j) = \frac{a}{b} \quad \text{and} \quad S(h_j \backslash h_k) = \frac{c}{d}.$$

Then it follows from the definition of supporting weight that

$$S(h_i \backslash h_k) = \frac{a+c}{b+d}. \tag{8}$$

If $S(h_i\backslash h_j)<S(h_j\backslash h_k)$, we have $S(h_i\backslash h_j)<S(h_i\backslash h_k)<S(h_j\backslash h_k)$. On the other hand, if $S(h_i\backslash h_j)>S(h_j\backslash h_k)$, we have $S(h_i\backslash h_j)>S(h_i\backslash h_k)>S(h_j\backslash h_k)$.

In terms of the supporting weights, we can rewrite the previous conditions (i), (ii) and (iii) as follows:

(i) $w_i<S(h_k\backslash h_n)<w_j$;

(ii) $w_1<S(h_j\backslash h_k)<w_i$;

(iii) $S(h_i\backslash h_j)<w_1$.

Note that if $S(h_j\backslash h_k)\leqq S(h_k\backslash h_n)$, then it follows from (7) and (8) that $S(h_j\backslash h_k)\leqq S(h_j\backslash h_n)\leqq S(h_k\backslash h_n)$.

Because of conditions (i) and (ii), the l-optimum partition of the subpolygon bounded by h_i and h_n must either be a fan or consist of both h_j and h_k as h-arcs. Hence, in order that both h_j and h_k exist in the l-optimum partition of the subpolygon bounded by h_i and h_n, $S(h_j\backslash h_n)$ must be less than w_i. Suppose $S(h_j\backslash h_n)<w_i$ and $S(h_i\backslash h_j)<w_1$. Then all three arcs h_i, h_j and h_k will exist in the l-optimum partition of the whole polygon if $S(h_i\backslash h_n)<w_1$. If $S(h_i\backslash h_n)\geqq w_1$, then the l-optimum partition will consist of a fan instead.

Define $S(h_n\backslash h_n)$ to be zero. We say that an arc h_k is the *ceiling* of another arc h_i if either condition (i) or conditions (iia), (iib), and (iic) are satisfied:

(i) $h_k=h_n$ if $h_i=h_n$, i.e., h_n is its own ceiling;

or

(ii) a) h_k is above h_i,

b) $S(h_i\backslash h_k)>S(h_k\backslash h_k\text{'s ceiling})$,

c) h_k is the lowest arc which satisfied (iia) and (iib). ("Lowest" means closest to the minimum vertex.)

The ceiling of an arc h_i is the lowest arc (above h_i) which *may* exist in an optimum partition even though h_i does not exist.

We say that an arc h_j is a *son* of another arc h_i if the following conditions are satisfied:

(i) h_j is above h_i (the son is above its father);

(ii) $S(h_j\backslash h_j\text{'s ceiling})<\min(w_i, w_i')$ where w_i, w_i' are the weights associated to the end vertices of h_i;

(iii) $S(h_i\backslash h_j)\leqq S(h_j\backslash h_j\text{'s ceiling})$; i.e., h_j is *not* a ceiling of h_i;

(iv) h_i is the highest arc which satisfies (i), (ii) and (iii). ("Highest" means closest to the maximum vertex.)

We shall prove in Theorem 6 that:

(i) if the father of any arc h_j exists in the l-optimum partition, then the arc h_j will also exist in the same partition;

(ii) if the father of h_j does not exist in the l-optimum partition, then the arc h_j also does not exist in the same partition.

From the definitions of the ceiling and the father–son relationship, we have the following observations:

(i) Every arc can have *at most* one father but an arc can have many sons. Also, the ancestor–descendant relationship is a transitive relationship. (Note that the ancestor–descendant relationship applies to arcs which are positive with respect to the whole monotone polygon as well.)

(ii) Every arc can have *at most* one ceiling but an arc can be the ceiling of many arcs.

(iii) All the h-arcs in the l-optimum partition of the subpolygon bounded by an arc h_i and its ceiling are descendants of h_j.

(iv) The ceiling of h_j cannot lie below any of the ceilings of h_j's descendants.

In other words, the subpolygon between h_j and its ceiling is nested completely inside the subpolygon bounded by h_j's father and the ceiling of h_j's father. If we treat each subpolygon bounded by an arc h_j and its ceiling as a *block*, then the ancestor–descendant relationship imposes a "nested block structure." For example, if h_k's father is h_j and h_j's father is h_i, then

h_k and its ceiling form the innermost block,
h_j and its ceiling form the middle block, and
h_i and its ceiling form the outermost block.

We shall show that the h-arcs in the l-optimum partition of an inner block exist in the l-optimum partition of the monotone polygon if and only if their ancestors; i.e., the h-arcs, forming the bottoms of the outerblocks, exist.

THEOREM 5. *Let h_j be a potential h-arc. If h_j is present in the l-optimum partition of a monotone polygon, its ceiling h_k will also be present in the l-optimum partition.*

Proof (by contradiction). Suppose there exists an h-arc h_j in the l-optimum partition while its ceiling h_k does not exist in the l-optimum partition. Without loss of generality, we can assume h_j to be the highest arc among those potential h-arcs which are present in the l-optimum partition and violate the theorem. From the definition of supporting weight, i.e. the left-hand side of inequality (7), we have $S(h_j\backslash h_k) < \min(w_j, w_j')$. Let h_c be the lowest h-arc above h_j in the l-optimum partition. The ceiling of h_c must be present in the l-optimum partition and we have $S(h_c\backslash h_c$'s ceiling$) < \min(w_j, w_j')$. Since there is no other h-arc between h_j and h_c in the l-optimum partition, the fan is l-optimum in the subpolygon between h_j and h_c. We have the following two cases.

Case 1. If h_c is the ceiling of h_k, we have $S(h_k\backslash h_c) < S(h_j\backslash h_k) < \min(w_j, w_j')$. Hence, the partition with h_k and its descendants as h-arcs costs less than the fan in the subpolygon between h_j and h_c, and we have a contradiction.

Case 2. If h_c is not the ceiling of h_k, we have the following two subcases.

Case 2a. Suppose h_c has a father which lies between h_j and h_c. It follows from the definition of the father–son relationship that $S(h_c$'s father$\backslash h_c) \leqq S(h_c\backslash h_c$'s ceiling$) < \min(w_j, w_j')$. Hence, the partition with h_c's father and its descendants costs less than the fan in the subpolygon bounded by h_j and h_c, and we have a contradiction.

Case 2b. Now h_c is not the ceiling of h_k and has no ancestor between h_j and h_c. Then among the potential h-arcs which lie between h_j and h_c, there exists a set of arcs $h_d, h_e, \cdots, h_f, h_k$ such that

h_c is the ceiling of h_d,
h_d is the ceiling of h_e,
⋮
h_f is the ceiling of h_k,
h_k is the ceiling of h_j,

and none of these arcs exists in the l-optimum partition. It follows from the definition of a ceiling that

$$S(h_d\backslash h_c) < S(h_e\backslash h_d) < \cdots < S(h_k\backslash h_f) < S(h_j\backslash h_k) < \min(w_j, w_j').$$

Now, the partition with h_d and all its descendants as h-arcs costs less than the fan in the subpolygon bounded by h_j and h_c, and we have a contradiction. In fact, using the same argument, we can show that the arcs $h_d, h_e, \cdots, h_f, h_k$ and all the descendants of these arcs should be in the l-optimum partition of the monotone polygon. □

THEOREM 6. *The sons of an arc h_j will exist in the l-optimum partition of a monotone polygon if and only if h_j is present in the l-optimum partition.*

Proof. (i) Instead of proving the "only if" part of the theorem directly, we will prove, by contradiction, that the existence of any son of h_j implies the existence of h_j in the l-optimum partition.

Among all the potential h-arcs in the monotone polygon, let h_j be the highest arc which is not present in the l-optimum partition of the polygon even though it has one or more sons present in the l-optimum partition. Among all the sons of h_j, let h_k be the lowest son which is present in the l-optimum partition. Finally, among all the potential h-arcs below h_j, let h_i be the highest h-arc which is present in the l-optimum partition. Hence, the l-optimum partition in the subpolygon bounded by h_i and h_k must be a fan. It follows from Theorem 5 that h_k's ceiling also exists in the l-optimum partition and we have $S(h_k\backslash h_k\text{'s ceiling}) < \min(w_i, w_i')$. Otherwise, the l-optimum partition in the subpolygon bounded by h_i and h_k's ceiling should be a fan and h_k as well as its descendants cannot be present in the l-optimum partition. From the definition of the father–son relationship, we know that $S(h_j\backslash h_k) \leqq S(h_k\backslash h_k\text{'s ceiling}) < \min(w_i, w_i')$. This means that in the subpolygon bounded by h_i and h_k, the partition consisting of h_j and its descendants as h-arcs costs less than the fan. This contradicts our assumption that the fan is l-optimum in the subpolygon bounded by h_i and h_k.

(ii) We shall prove the "if" part of the theorem directly by contradiction. Among all the potential h-arcs in the monotone polygon, let h_k be the highest arc which is not present in the l-optimum partition of the polygon even though its father h_j is present in the l-optimum partition. Among all the potential h-arcs present in the l-optimum partition, let h_c be the lowest h-arc above h_k and let h_b be the highest h-arc below h_k in the l-optimum partition as shown in Fig. 8. Hence, the l-optimum partition in the subpolygon bounded by h_b and h_c must be a fan. Note that h_c must be a ceiling of h_k because h_k is the highest arc not satisfying the necessary condition of the theorem. Otherwise, h_c is a descendant of h_k, and by part (i) of this proof, h_k will exist in the l-optimum partition of the polygon. The arc h_b must either be h_j itself or lie above h_j. Hence, we have $\min(w_b, w_b') \geqq \min(w_j, w_j')$. By the definition of the father–son

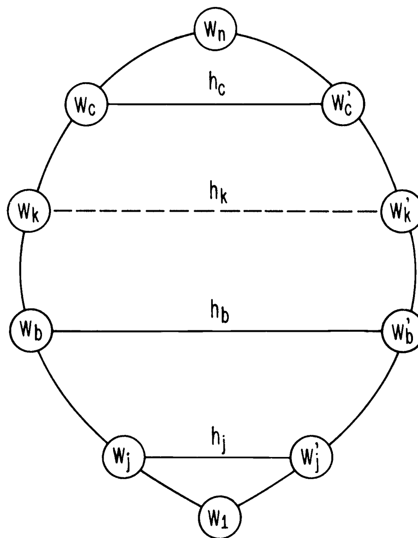


FIG. 8

relationship, we have $S(h_k \backslash h_c) < \min(w_j, w'_j) \leqq \min(w_b, w'_b)$. This means that in the subpolygon bounded by h_b and h_c, the partition consisting of h_k and its descendants is cheaper than the fan. This contradicts our assumption that the fan is l-optimum in the subpolygon bounded by h_b and h_c. □

COROLLARY 1. *The descendants of any arc h_j will exist in the l-optimum partition of a monotone polygon if and only if h_j exists in the l-optimum partitions.*

Proof. The corollary follows from Theorem 6. □

It follows from Corollary 1 that if a potential h-arc h_j is present in the l-optimum partition of a monotone polygon, all its descendants, all its ancestors and all potential h-arcs which have some ancestors common to those of h_j will be present in the l-optimum partition.

THEOREM 7. *Let h_i and h_j be two potential h-arcs such that h_j is above h_i and the l-optimum partition in the subpolygon bounded by h_i and h_j is a fan. If $S(h_j \backslash h_j$'s ceiling$) \geqq \min(w_i, w'_i)$, then h_j and all its descendants cannot exist in the l-optimum partition of any subpolygon bounded above by h_n and below by any potential h-arc not higher than h_i.*

Proof (by contradiction). Assume that there exist such two potential h-arcs but that h_j is present in the l-optimum partition of a subpolygon bounded above by h_n and below by a potential h-arc lower than h_i. Without loss of generality, let h_j be the lowest arc among all the potential h-arcs which are present in the l-optimum partition and which satisfy the assumption. Hence, none of the potential h-arcs between h_i and h_j can exist in the l-optimum partition. Let h_b be the highest potential h-arc below h_j in the l-optimum partition. Since h_b can either be h_i itself or a potential h-arc below h_i, we have $\min(w_b, w'_b) \leqq \min(w_i, w'_i) \leqq S(h_j \backslash h_j$'s ceiling). The partition with h_j and all its descendants costs more than the fan in the subpolygon bounded by h_b and h_j's ceiling and we have a contradiction. □

Using Theorem 6, we can start from an innermost block and work our way out. Suppose we have located the ceiling of a potential h-arc h_i. Then we can treat h_i and all the sons (and descendants) of h_i as a unit; i.e., all h_i's sons are *condensed* into h_i. Let h_b be the potential h-arc immediately below h_i in the monotone polygon. The l-optimum partition in the subpolygon bounded by h_b and the ceiling of h_i must consist of either h_i and all its descendants as h-arcs or of a fan, depending on whether $S(h_i \backslash h_i$'s ceiling$) < \min(w_b, w'_b)$ or $S(h_i \backslash h_i$'s ceiling$) \geqq \min(w_b, w'_b)$. If the fan is cheaper, we can delete h_i and all its descendants since none of these arcs can appear as h-arcs in the l-optimum partition of the polygon (Theorem 7).

Now, what we have to do is to find an innermost block to start our computations. After obtaining the list of potential h-arcs of the monotone polygon using the one-sweep algorithm, we know that the degenerated arc h_n is the ceiling of the highest potential h-arc in the list, and this potential h-arc does not have any descendants. So, we should start from the highest potential h-arc and work our way down the list of potential h-arcs.

We now give two examples to illustrate the concepts, notation and algorithm. Then a formal description of the algorithm will be given.

Consider a monotone basic polygon with five potential h-arcs, $h_6, h_5, \cdots, h_2$ where h_6 is the highest arc as shown symbolically in Fig. 9. Let $w_i \leqq w'_i$ for $i = 2, 3, \cdots$. The maximum vertex, which lies above h_6, has the weight w_7 and the minimum vertex, which lies below h_2, has the weight w_1. We can regard w_7 (and w_1) as a degenerated arc and use h_7 to represent w_7 (and h_1 to represent w_1).

Example 1. There are two possible candidates for the l-optimum partition in the subpolygon bounded by h_5 and h_7. We shall use $C(\underline{h_5}, h_6, \overline{h_7})$ to denote the cost

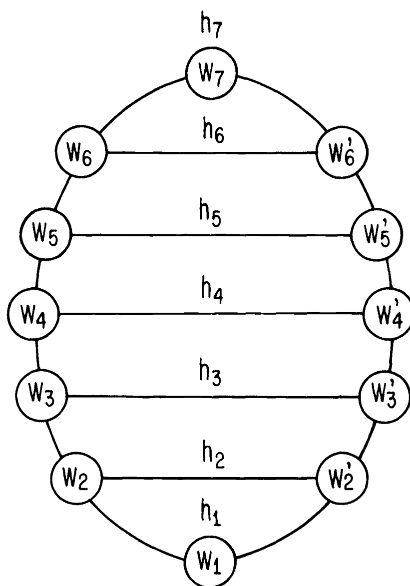


FIG. 9. *A 12-gon with 5 h-arcs.*

of the partition with h_6, and $H_0(\underline{h_5}, \overline{h_7})$ to denote the cost of the fan in the subpolygon. Similarly, we shall use $C(\underline{h_2}, h_5, h_6, \overline{h_7})$ to denote the cost of the partition with h_5 and h_6 as the only 2 h-arcs in the subpolygon bounded by h_2 and h_7. Note that there is a bar underneath the h-arc which forms the bottom of the subpolygon and a bar above the h-arc which forms the top of the subpolygon.

The necessary computations and results of the comparisons are shown in Table 1.

If $S(h_2\backslash h_7) < w_1$, the partition with h_2, h_3, h_4, h_5 and h_6 as h-arcs will be l-optimum in the polygon. Otherwise, the fan $H_0(\underline{h_1}, \overline{h_7})$ will be l-optimum.

Now, let us consider a more complicated example.

Example 2. Consider the 6 potential h-arcs shown in Fig. 9. Assume that we have the computations and results shown in Table 2.

If $S(h_2\backslash h_7) < w_1$, the partition with h_2, h_5 and h_6 as h-arcs is l-optimum. Otherwise, the fan $H_0(\underline{h_1}, \overline{h_7})$ will be l-optimum.

Let us give the algorithm for finding the l-optimum partition of a monotone basic polygon.

ALGORITHM M

(I) Get all the potential h-arcs of the polygon by the one-sweep algorithm [6]. (All these arcs form a vertical list, with the highest arc closest to the maximum vertex V_n and the lowest arc closest to the minimum vertex V_1.)

(II) Process the potential h-arcs one by one, from the top to the bottom. Let h_j be the potential h-arc being processed, let h_k be the potential h-arc immediately above h_j, and let h_i be the potential h-arc immediately below h_j in the monotone polygon. (If h_j is the highest potential h-arc in the polygon, h_k will be the degenerate arc h_n; if h_j is the lowest potential h-arc in the polygon, h_i will be the degenerated arc h_1.) Note that by the time we start processing h_j, we have already obtained the l-optimum partition of the subpolygon between h_j and h_n. We have also located the ceilings of every h-arc in the l-optimum partition of this subpolygon. When we process h_j, we

TABLE 1

Computations	Observations	Remarks
1. $S(h_6 \backslash h_7)$	$w_4 < S(h_6 \backslash h_7) < w_5$	h_7 is the ceiling of h_6: $S(h_6 \backslash h_7) < w_5 \Rightarrow C(\underline{h_5}, h_6, \overline{h_7}) < H_0(\underline{h_5}, \overline{h_7})$
2. $S(h_5 \backslash h_6)$	$w_3 < S(h_5 \backslash h_6) < w_4$	$S(h_5 \backslash h_6) < S(h_6 \backslash h_7) \Rightarrow h_6$ is a son of h_5; condense h_6 into h_5 and calculate $S(h_5 \backslash h_7)$
3. $S(h_5 \backslash h_7)$	$w_3 < S(h_5 \backslash h_7) < w_4$	h_7 is the ceiling of h_5; $S(h_5 \backslash h_7) < w_4 \Rightarrow C(\underline{h_4}, h_5, h_6, \overline{h_7}) < H_0(\underline{h_4}, \overline{h_7})$.
4. $S(h_4 \backslash h_5)$	$w_2 < S(h_4 \backslash h_5) < w_3$	$S(h_4 \backslash h_5) < S(h_5 \backslash h_7) \Rightarrow h_5$ is a son of h_4; condense h_5 into h_4 and calculate $S(h_4 \backslash h_7)$
5. $S(h_4 \backslash h_7)$	$w_2 < S(h_4 \backslash h_7) < w_3$	h_7 is the ceiling of h_4; $S(h_4 \backslash h_7) < w_3 \Rightarrow C(\underline{h_3}, h_4, h_5, h_6, \overline{h_7}) < H_0(\underline{h_3}, \overline{h_7})$
6. $S(h_3 \backslash h_4)$	$w_1 < S(h_3 \backslash h_4) < w_2$	$S(h_3 \backslash h_4) < S(h_4 \backslash h_7) \Rightarrow h_4$ is a son of h_3; condense h_4 into h_3 and calculate $S(h_3 \backslash h_7)$
7. $S(h_3 \backslash h_7)$	$w_1 < S(h_3 \backslash h_7) < w_2$	h_7 is the ceiling of h_3; $S(h_3 \backslash h_7) < w_2 \Rightarrow C(\underline{h_2}, h_3, h_4, h_5, h_6, \overline{h_7}) < H_0(\underline{h_2}, \overline{h_7})$
8. $S(h_2 \backslash h_3)$	$S(h_2 \backslash h_3) < w_1$	$S(h_2 \backslash h_3) < S(h_3 \backslash h_7) \Rightarrow h_3$ is a son of h_2; condense h_3 into h_2 and calculate $S(h_2 \backslash h_7)$
9. $S(h_2 \backslash h_7)$	?	

TABLE 2

Computations	Observations	Remarks
1. $S(h_6 \backslash h_7)$	$w_1 < S(h_6 \backslash h_7) < w_2$	h_7 is the ceiling of h_6; $S(h_6 \backslash h_7) < w_5 \Rightarrow C(\underline{h_5}, h_6, \overline{h_7}) < H_0(\underline{h_5}, \overline{h_7})$
2. $S(h_5 \backslash h_6)$	$S(h_6 \backslash h_7) < S(h_5 \backslash h_6) < w_2$	$S(h_5 \backslash h_6) > S(h_6 \backslash h_7) \Rightarrow h_6$ is the ceiling of h_5; $S(h_5 \backslash h_6) < w_4 \Rightarrow C(\underline{h_4}, h_5, \overline{h_6}) < H_0(\underline{h_4}, \overline{h_6})$
3. $S(h_4 \backslash h_5)$	$w_2 < S(h_4 \backslash h_5) < w_3$	$S(h_4 \backslash h_5) > S(h_5 \backslash h_6) \Rightarrow h_5$ is the ceiling of h_4; $S(h_4 \backslash h_5) < w_3 \Rightarrow C(\underline{h_3}, h_4, \overline{h_5}) < H_0(\underline{h_3}, \overline{h_5})$
4. $S(h_3 \backslash h_4)$	$w_1 < S(h_3 \backslash h_4) < w_2$	$S(h_3 \backslash h_4) < S(h_4 \backslash h_5) \Rightarrow h_4$ is a son of h_3; condense h_4 into h_3 and calculate $S(h_3 \backslash h_5)$
5. $S(h_3 \backslash h_5)$	$w_2 < S(h_3 \backslash h_5) < w_3$	$S(h_3 \backslash h_5) > S(h_5 \backslash h_6) \Rightarrow h_5$ is the ceiling of h_3; $S(h_3 \backslash h_5) > w_2 \Rightarrow C(\underline{h_2}, h_3, h_4, \overline{h_5}) > H_0(\underline{h_2}, \overline{h_5})$; both h_3 and h_4 cannot exist in the l-optimum partition and should be deleted from the list of potential h-arcs; we should then check to see if the fan is cheaper in the subpolygon bounded by h_2 and h_6; $S(h_5 \backslash h_6) < w_2 \Rightarrow C(\underline{h_2}, h_5, \overline{h_6}) < H_0(\underline{h_2}, \overline{h_6})$
6. $S(h_2 \backslash h_5)$	$S(h_2 \backslash h_5) < w_1$	$S(h_2 \backslash h_5) < S(h_5 \backslash h_6) \Rightarrow h_5$ is a son of h_2; we should condense h_5 into h_2 and calculate $S(h_2 \backslash h_6)$
7. $S(h_2 \backslash h_6)$	$S(h_2 \backslash h_6) < w_1$	$S(h_2 \backslash h_6) < S(h_6 \backslash h_7) \Rightarrow h_6$ is a son of h_2; we should condense h_6 into h_2 and calculate $S(h_2 \backslash h_7)$.
8. $S(h_2 \backslash h_7)$	?	

first locate the ceiling of h_j and condense all h_j's descendants into h_j. Then we obtain the l-optimum partition of the subpolygon between h_i and h_n by deleting those blocks of arcs which cannot exist in the l-optimum partition of the subpolygon between h_i and h_n.

While ($h_j \neq$ the degenerated arc h_1) do
Begin
1. [To locate the ceiling of h_j].
While $S(h_j \backslash h_k) \leqq S(h_k \backslash h_k$'s ceiling) do
Begin
a. Comment: Now, h_k is a son of h_j.
b. We will combine h_k and all its descendants into h_j and calculate the combined supporting weight $S(h_j \backslash h_k$'s ceiling).
c. Replace h_k by h_k's ceiling; i.e., h_k is always used to denote the lowest h-arc above h_j which is not yet combined into h_j.
End.
2. [To delete those blocks of arcs which cannot exist in the l-optimum partition of the subpolygon between h_i and h_n].
While $C(\underline{h_i}, h_j$ and h_j's descendants, $\overline{h_j\text{'s ceiling}}) \geqq H_0(\underline{h_i}, \overline{h_j\text{'s ceiling}})$;
i.e., $S(h_j \backslash h_j$'s ceiling) $\geqq \min(w_i, w_i')$. Do
Begin
a. Delete h_j and all its descendants from the list of potential h-arcs.
b. Replace h_j by the ceiling of h_j; i.e., h_j is always used to denote the arc immediately above h_i in the subpolygon between h_i and h_n.
End.
3. [Prepare to process next arc].
Replace h_k by h_j, h_j by h_i and h_i by the arc immediately below h_i in the list of potential h-arcs.
End.

(III) Output the l-optimum partition consisting of the arcs which remain in the list of potential h-arcs after Step II as h-arcs.
Then stop.

THEOREM 8. *The partition produced by Algorithm* M *is l-optimum.*

Proof. We have shown in Part I of this paper [6] that all h-arcs present in the l-optimum partition of the polygon are potential h-arcs, and all potential h-arcs are included in the list obtained by the one-sweep algorithm. We claim that (i) whenever Algorithm M finishes Step II.1, the ceiling of h_j is correctly located, (ii) whenever Algorithm M finishes Step II.2, the arcs which have been deleted by Algorithm M cannot exist in the l-optimum partition of the subpolygon bounded above by h_n and below by an arc lower than h_i, and (iii) the partition consisting of all the potential h-arcs remaining above h_i as h-arcs is l-optimum in the subpolygon bounded by h_i and h_n after Step II.2. (If the claim is true, the partition output by Algorithm M will be l-optimum in the monotone polygon.)

We shall prove the claim by induction on the number of h-arcs above an arc h_j.

It is easy to see that the claim is true when $h_j =$ the highest arc in the list of potential h-arcs.

Suppose the claim is true for all potential h-arcs above some arc h_j. Let h_i be the arc immediately below h_j in the list of potential h-arcs. Just before Algorithm M starts processing h_j, all the potential h-arcs which remain above h_j exist as h-arcs in the

l-optimum partition of the subpolygon between h_j and h_n. We can divide these arcs into two groups: (i) those which are descendants of some other arcs in the subpolygon, and (ii) those which have no ancestor in the subpolygon.

It follows from the definition of the father–son relationship that only arcs in group (ii) can be sons of h_j. Let the set of arcs in group (ii) be $h_t, h_{t-1}, \cdots, h_p, h_{p-1}, \cdots, h_{j+2}, h_{j+1}$ such that h_n is above h_t, h_t is above $h_{t-1}, \cdots, h_p$ is above $h_{p-1}, \cdots, h_{j+2}$ is above h_{j+1} and h_{j+1} is above h_j. Note that there exists no other h-arc between h_{j+1} and h_j in the l-optimum partition of the subpolygon. Since none of these arcs has an ancestor in the subpolygon, we must have

h_n as the ceiling of h_t,
h_t as the ceiling of h_{t-1},
$\vdots$
h_p as the ceiling of h_{p-1},
$\vdots$
h_{j+2} as the ceiling of h_{j+1}.

It follows from the definition of the ceiling that

$$S(h_{j+1}\backslash h_{j+2}) > \cdots > S(h_{p-1}\backslash h_p) > \cdots > S(h_{t-1}\backslash h_t) > S(h_t\backslash h_n).$$

Since h_{j+1} is the lowest h-arc in the l-optimum partition of the subpolygon bounded by h_j and h_n, we have

$$\min(w_j, w_j') > S(h_{j+1}\backslash h_{j+2}) > \cdots > S(h_t\backslash h_n).$$

Now, if $S(h_j\backslash h_{j+1}) \leqq S(h_{j+1}\backslash h_{j+2})$, all four conditions of the father–son relationship are satisfied and Algorithm M will correctly condense h_{j+1} and its descendants into h_j. Using the same argument repeatedly, we conclude that Algorithm M correctly locates the ceiling of h_j at the end of Step II.1. Whenever the potential h-arc h_j and its descendants are removed in Step II.2, the conditions in Theorem 7 are satisfied. Hence h_j and its descendants cannot exist in the l-optimum partition of any subpolygon bounded above by h_n and below by a potential h-arc lower than h_i. Now, at the end of Step II.2, we can again divide the potential h-arcs remaining above h_i into two groups:

(i) those which are descendants of some other arcs in the subpolygon, and
(ii) those which have no ancestor in the subpolygon.

Let h_j be the h-arc immediately above h_i after Step II.2. The arc h_j must be the lowest arc in group (ii). It follows from the definition of ceiling that for any arc h_k above h_j in group (ii), we have

$$\min(w_i, w_i') > S(h_j\backslash h_j\text{'s ceiling}) > S(h_k\backslash h_k\text{'s ceiling}).$$

From Theorem 6, if any of the arcs in group (ii) does not exist in the l-optimum partition, all its descendants in group (i) will not exist in the l-optimum partition. Suppose the partition consisting of all the potential h-arcs remaining above h_i as h-arcs is not l-optimum in the subpolygon between h_i and h_n. Then some of these potential h-arcs in group (ii) and their descendants should not exist in the l-optimum partition. Assume that h_k is the highest potential h-arc remaining above h_i after Step II.2, but h_k should not exist in the l-optimum partition. Let h_b be the highest h-arc below h_k in the l-optimum partition. Hence, the fan should be l-optimum in the subpolygon between h_b and h_k's ceiling. Since $S(h_k\backslash h_k\text{'s ceiling}) < \min(w_i, w_i') \leqq \min(w_b, w_b')$, the

partition with h_k and its descendants as h-arcs in the subpolygon bounded by h_b and h_k's ceiling is always cheaper than the fan, and we have a contradiction.

Hence, the claim is true, and the partition output by Algorithm M is l-optimum. □

In order for Algorithm M to run efficiently, we need a data structure which enables us to calculate the supporting weights, to keep track of the ceiling of each potential h-arc and to update the list of potential h-arcs easily. One way to implement Algorithm M is to place all potential h-arcs obtained in Step I in a linear linked list, with the highest arc at the head of the list and the lowest arc at the tail of the list. Each of these potential h-arcs, say h_i, is associated with a record variable with the following fields:

(i) the label of the end vertex which is closer to V_1 in the clockwise direction;
(ii) the label of the other end vertex;
(iii) the ceiling of h_i;
(iv) the list of sons of h_i;
(v) the cost of the l-optimum partition in the subpolygon between h_i and its ceiling, i.e. the numerator of $S(h_i \backslash h_i$'s ceiling);
(vi) the quantity $(w_i : w_j + w_j \cdot w'_j + w'_j : w'_i) - w_i \cdot w'_i$ where w_i, w'_i are weights of the end vertices of the potential h-arc h_i and w_j, w'_j are the weights of the end vertices of h_i's ceiling, i.e. the denominator of $S(h_i \backslash h_i$'s ceiling) (it is obtained by subtracting the product $w_i \cdot w'_i$ from the sum of the adjacent products from w_i to w'_i around the subpolygon $w_i - \cdots - w_j - w'_j - \cdots - w'_i$); and
(vii) the supporting weight $S(h_i \backslash h_i$'s ceiling).

Note that only the first three fields of each potential h-arc are defined at the end of Step I, the other four fields of each potential h-arc are set to the correct value when the potential h-arc is being processed in Step II. Since the sums of adjacent products of the form $w_i : w_j$ are used repeatedly in calculating the cost of the fan between two adjacent potential h-arcs and the denominators of the supporting weights, we can eliminate a lot of repeated calculations by initializing the elements of an array CP to

$$\text{CP}[1] = 0 \quad \text{and} \quad \text{CP}[i] = w_1 : w_i \quad \text{for } 2 \leqq i \leqq n.$$

Then the sum of the adjacent products $w_i : w_j$ can be obtained from $\text{CP}[j] - \text{CP}[i]$.

As we process the arcs in the list of potential h-arcs one by one from the top to the bottom, we shall remove a potential h-arc from the list if (i) the arc is found to be a son of another potential h-arc in Step II.1, or (ii) the partition with the arc and all its descendants is not l-optimum in some subpolygon in Step II.2. Let h_k be an arc which is removed from the list in Step II.1 and let h_j be its father. After h_k is removed from the list of potential h-arcs, it will be added to the list of h_j's sons, i.e. the fourth field of h_j. Then, we have to calculate the supporting weight $S(h_j \backslash h_k$'s ceiling). The numerator of $S(h_j \backslash h_k$'s ceiling) can be obtained by adding the numerator of $S(h_k \backslash h_k$'s ceiling) in the fifth field of h_k to the numerator of $S(h_i \backslash h_j)$. Similarly, the denominator of $S(h_j \backslash h_k$'s ceiling) can be obtained by adding the denominator of $S(h_k \backslash h_k$'s ceiling) in the sixth field of h_k to the denominator of $S(h_j \backslash h_k)$. Hence, we can calculate $S(h_j \backslash h_k$'s ceiling) in a constant amount of time. Note that whenever Algorithm M finishes Step II.2, only those potential h-arcs which are present in the l-optimum partition of the subpolygon between h_i and h_n and yet have no ancestors above h_i remain above h_i in the list of potential h-arcs.

THEOREM 9. *Algorithm* M *runs in* $O(n)$ *time.*

Proof. It takes $O(n)$ time to sweep around the monotone polygon twice, once to obtain all potential h-arcs in Step I and once to initialize the array CP. There are two while loops in Step II, and it only takes a constant amount of time to execute either while loop once. Whenever the while loop in Step II.1 is executed once, a potential h-arc is removed from the list and condensed into its father. Whenever the while loop in Step II.2 is executed once, a potential h-arc is deleted from the list. Once an arc is removed or deleted from the list, it will never be considered again. Since there are at most $n-3$ arcs in the list obtained in Step I, Algorithm M can execute both while loops at most $n-3$ times. So is takes $O(n)$ time to process all the potential h-arcs in Step II and to output the l-optimum partition in Step III. Hence, Algorithm M runs in $O(n)$ time. □

3. The convex polygon. In this section we shall extend the results in § 2 to the case of a general convex polygon.

There may be several local maximum vertices in a general convex polygon. Let us still draw the polygon in such a way that the global minimum vertex is at the bottom. From Theorem 4 of Part I, we know that all potential h-arcs are still compatible in a general convex polygon. However, unlike those in a monotone polygon, the potential h-arcs no longer form a linear list. Instead, they form a tree, called an *arc-tree*. In Fig. 10a, there is a 12-gon with 6 potential h-arcs, and they are labelled as h_2, h_3, h_4, h_5, h_6 and h_7. (Note that we also obtain V_4-V_3, V_7-V_6 and V_6-V_8 from the one-sweep algorithm. In order to have a simpler example, let us assume that all three of these arcs are unstable and hence are not shown in Fig. 10a.) To get a better feeling of the arc-tree, we can redraw the 12-gon as shown in Fig. 10b. By regarding V_1 as a degenerated arc h_1, V_{12} as a degenerated arc h_8, and V_{11} as a degenerated arc h_9, we have h_1 as the root of the arc tree and the arcs h_8 and h_9 as the leaves.

An arc h_j is *above* another arc h_i (and h_i is *below* h_j) if h_j is in one of the subtrees of h_i. We shall be dealing with subpolygons, each bounded below by a potential h-arc and above by a set of potential h-arcs. We can define the supporting weights of the potential h-arcs in a similar way. For example, the supporting weight of the arc h_2

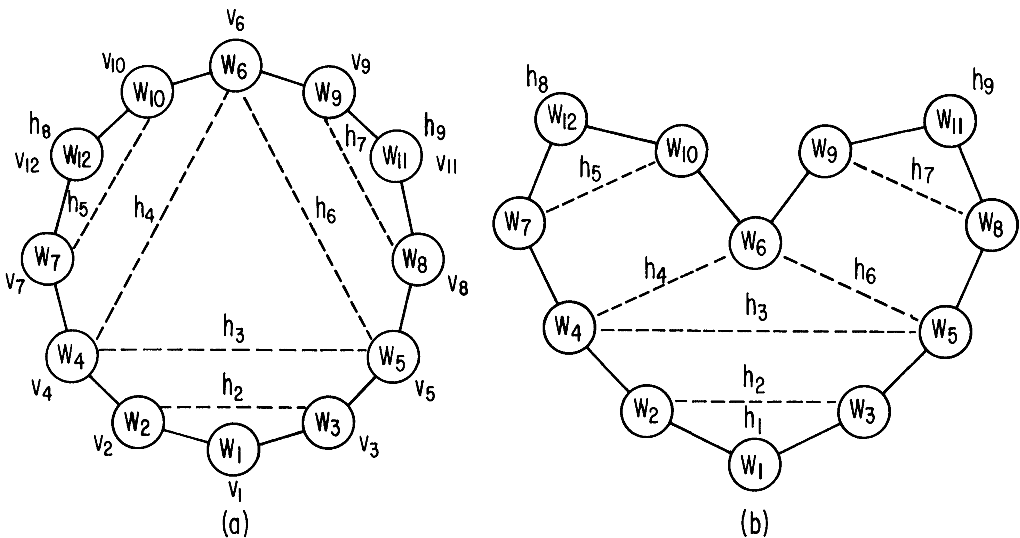


FIG. 10. *A general 12-gon.*

with respect to the subpolygon bounded above by $\{h_4, h_6\}$ in Fig. 10b equals

$$\frac{C(w_2, w_4, w_6, w_5, w_3)}{[w_2 : w_3 - (w_4 : w_6 - w_4 \cdot w_6) - (w_6 : w_5 - w_6 \cdot w_5)] - w_2 \cdot w_3}$$

and is denoted by $S(h_2 \backslash \{h_4, h_6\})$. Again, for any leaf node h_n, we define $S(h_n \backslash \{h_n\})$ to be zero.

We say that a set of potential h-arcs U_i is the *ceiling* of another potential h-arc h_i (or simply h_i's ceiling for short) if either condition (i) or conditions (iia), (iib), (iic) and (iid) are satisfied:

(i) $U_i = \{h_i\}$ if h_i is a leaf node;

or

(ii) for all $h_k \in U_i$,

a) h_k is above h_i;

b) $S(h_i \backslash U_i) > (h_k \backslash h_k$'s ceiling);

c) for all $h_j \in U_i$ such that $h_j \neq h_k$, neither h_j is above h_k nor h_k is above h_j; and

d) conditions (iia), (iib) or (iic) will be violated if h_k is replaced by any arc below h_k in the subpolygon between h_i and U_i.

We say that an arc h_j is a *son* of another arc h_i if the following conditions are satisfied:

(i) h_j is above h_i (the son is above its father);

(ii) $S(h_j \backslash h_j$'s ceiling$) < \min(w_i, w_i')$ where w_i, w_i' are weights associated to the end vertices of h_i;

(iii) h_j is *not* in the ceiling of h_i; and

(iv) h_i is the highest arc which satisfies (i), (ii) and (iii).

It is easy to see that all the previous discussions on the ceilings and the ancestor–descendant relationships in § 2 still hold under the new definition of ceilings and father–son relationships. Using arguments similar to those used in the proofs of Theorems 5, 6 and 7, we can generalize Theorems 5, 6, 7 and Corollary 1 as follows:

THEOREM 10. *If a potential h-arc h_j exists in the l-optimum partition of a convex polygon, all potential h-arcs in its ceiling will also exist in the l-optimum partition.*

Proof. Omitted. □

THEOREM 11. *The sons of an arc h_j will exist in the l-optimum partition of a convex polygon if and only if h_j is present in the l-optimum partition.*

Proof. Omitted. □

COROLLARY 2. *The descendants of an arc h_j will exist in the l-optimum partition of a convex polygon if and only if h_j exists in the l-optimum partition.*

Proof. The corollary follows from Theorem 11. □

THEOREM 12. *Let X be a set of potential h-arcs above another potential h-arc h_i such that* (i) *for any two arcs $h_j, h_k \in X$, neither h_j is above h_k nor h_k is above h_j if $h_j \neq h_k$, and* (ii) *the l-optimum partition in the subpolygon between h_i and the arcs in X is a fan. Let h_j be a potential h-arc in X such that for any $h_k \in X$, $S(h_j \backslash h_j$'s ceiling$) \geqq S(h_k \backslash h_k$'s ceiling). If $S(h_j \backslash h_j$'s ceiling$) \geqq \min(w_i, w_i')$ where w_i, w_i' are the weights associated with the end vertices of h_i, then h_j and all its descendants cannot exist in the l-optimum partition of any upper subpolygon bounded below by a potential h-arc no higher than h_i.*

Using the facts in Theorems 10, 11, 12 and Corollary 2, we can again start from the potential h-arcs which lie immediately below the leaf nodes and work our way down. The leaf nodes are the ceiling of these arcs. Before we can locate the ceiling of any arc which does not lie immediately below the leaf nodes, we must first process all the arcs above it, i.e. the arcs in its subtrees. Hence, we can do a postorder traversal through the arc tree. When we process a potential h-arc, we first find the l-optimum

partition of the subpolygon bounded below by the arc and above by the leaf nodes in its subtrees; then we will locate the ceiling of the potential h-arc. Let us consider the following example.

Example 3. Consider the 12-gon with six potential h-arcs as shown in Figs. 10a and 10b. The necessary computations and the results of comparisons are shown in Table 3.

TABLE 3

Computations	Observations	Remarks
1. $S(h_5\backslash\{h_8\})$	$w_1 < S(h_5\backslash\{h_8\}) < w_2$	The fan is l-optimum in the subpolygon between h_5 and h_8; $\{h_8\}$ is the ceiling of h_5 h_4 is the next arc to be processed.
2. $S(h_4\backslash\{h_5\})$	$w_3 < S(h_4\backslash\{h_5\}) < w_4$	$S(h_5\backslash\{h_8\}) < w_4 \Rightarrow C(\underline{h_4}, h_5, \overline{h_8}) < H_0(\underline{h_4}, \overline{h_8})$ $S(h_4\backslash\{h_5\}) > S(h_5\backslash\{h_8\}) \Rightarrow \{h_5\}$ is the ceiling of h_4 Before we can process h_3, we have to process h_7 first
3. $S(h_7\backslash\{h_9\})$	$w_2 < S(h_7\backslash\{h_9\}) < w_3$	The fan is l-optimum in the subpolygon between h_7 and h_9 $\{h_9\}$ is the ceiling of h_7 h_6 is the next arc to be processed
4. $S(h_6\backslash\{h_7\})$	$w_3 < S(h_6\backslash\{h_7\})$ $< S(h_4\backslash\{h_5\}) < w_4$	$S(h_7\backslash\{h_9\}) < w_5 \Rightarrow C(\underline{h_6}, h_7, \overline{h_9}) < H_0(\underline{h_6}, \overline{h_9})$ $S(h_6\backslash\{h_7\}) > S(h_7\backslash\{h_9\}) \Rightarrow \{h_7\}$ is the ceiling of h_6 h_3 is the next arc to be processed
5. $S(h_3\backslash\{h_4, h_6\})$	$w_2 < S(h_3\backslash\{h_4, h_6\})$ $< S(h_6\backslash\{h_7\})$ $< S(h_4\backslash\{h_5\})$	$S(h_6\backslash\{h_7\}) < S(h_4\backslash\{h_5\}) < w_4$ $\Rightarrow C(\underline{h_3}, h_4, h_6, \overline{h_5}, \overline{h_7}) < H_0(\underline{h_3}, \overline{h_5}, \overline{h_7})$ Both h_4 and h_6 may be sons of h_3 since $S(h_4\backslash\{h_5\}) > S(h_6\backslash\{h_7\})$, test h_4 first to see if h_4 is a son of h_3 $S(h_3\backslash\{h_4, h_6\}) < S(h_4\backslash\{h_5\}) \Rightarrow h_4$ is a son of h_3 Condense h_4 into h_3 and calculate $S(h_3\backslash\{h_5, h_6\})$
6. $S(h_3\backslash\{h_5, h_6\})$	$w_2 < S(h_3\backslash\{h_5, h_6\})$ $< S(h_6\backslash\{h_7\})$	$S(h_3\backslash\{h_5, h_6\}) < S(h_6 \Rightarrow \{h_7\})\backslash h_6$ is a son of h_3 Condense h_6 into h_3 and calculate $S(h_3\backslash\{h_5, h_7\})$.
7. $S(h_3\backslash\{h_5, h_7\})$	$S(h_5\backslash\{h_8\})$ $< w_2 < S(h_7\backslash\{h_9\})$ $< S(h_3\backslash\{h_5, h_7\}) < w_3$	$S(h_3\backslash\{h_5, h_7\}) > S(h_7\backslash\{h_9\}) > S(h_5\backslash\{h_8\})$ $\Rightarrow \{h_5, h_7\}$ is the ceiling of h_3 h_2 is the next arc to be processed.
8. $S(h_2\backslash\{h_5, h_9\})$	$S(h_2\backslash\{h_5, h_9\}) < w_1$	$S(h_3\backslash\{h_5, h_7\}) > w_2$ $\Rightarrow C(\underline{h_2}, h_3, h_4, h_6, \overline{h_5}, \overline{h_7}) > H_0(\underline{h_2}, \overline{h_5}, \overline{h_7})$ h_3, h_4 and h_6 cannot exist in the l-optimum partition and should be deleted from the arc tree Now, h_5, h_7 are the two arcs immediately above h_2 since $S(h_7\backslash\{h_9\}) > S(h_5\backslash\{h_8\})$, test h_7 first to see if h_7 can be deleted from the arc tree $S(h_7\backslash\{h_9\}) > w_2$ $\Rightarrow C(\underline{h_2}, h_7, \overline{h_5}, \overline{h_9}) > H_0(\underline{h_2}, \overline{h_5}, \overline{h_9})$ h_7 should be deleted from the arc tree $S(h_5\backslash\{h_8\}) < w_2$ $\Rightarrow C(\underline{h_2}, h_5, \overline{h_8}, \overline{h_9}) < H_0(\underline{h_2}, \overline{h_8}, \overline{h_9})$ $S(h_2\backslash\{h_5, h_9\}) < S(h_5\backslash\{h_8\}) \Rightarrow h_6$ is a son of h_2 Condense h_5 into h_2 and calculate $S(h_2\backslash\{h_8, h_9\})$
9. $S(h_2\backslash\{h_8, h_9\})$	?	

If $S(h_2\backslash\{h_8, h_9\}) < w_1$, the partition with h_2 and h_5 as h-arcs is l-optimum. Otherwise, the fan $H_0(\underline{h_1}, \overline{h_8}, \overline{h_9})$ will be l-optimum.

From the above example, we have the following observations. Let h_j be the arc being processed and let X be the set of arcs immediately above h_j in the arc tree. By the time we process h_j, we have already obtained (i) the l-optimum partitions of the subpolygons between the leaf nodes and the arcs in X and (ii) the ceilings of all the arcs in X. For any arc h_k in X, the l-optimum partition in the subpolygon bounded below by h_j and above by the arcs in $X-\{h_k\}\cup h_k$'s ceiling must either be a fan or consist of h_k and its descendants as h-arcs depending on whether $S(h_k\backslash h_k\text{'s ceiling}) \geqq \min(w_j, w_j')$ or $S(h_k\backslash h_k\text{'s ceiling}) < \min(w_j, w_j')$, where w_j, w_j' are the weights associated with the end vertices of h_j. If the fan is cheaper, h_k and h_k's descendants will be removed from the arc tree and the set X becomes $X-\{h_k\}\cup h_k$'s ceiling. We can repeat the above process until the l-optimum partition in the subpolygon bounded below by h_j and above by the leaf nodes in the subtrees of h_j is obtained. Since $\max_{h_k\in X} S(h_k\backslash h_k\text{'s ceiling}) < \min(w_j, w_j')$ implies $(\forall h_k \in X)(S(h_k\backslash h_k\text{'s ceiling}) < \min(w_j, w_j'))$, the arc with maximum supporting weight in X should be chosen and tested for possible deletion. Similarly, since $\max_{h_k\in X} S(h_k\backslash h_k\text{'s ceiling}) < S(h_j\backslash X)$ implies $(\forall h_k \in X)(S(h_k\backslash h_k\text{'s ceiling}) < S(h_j\backslash X))$, the arc with maximum supporting weight should also be chosen and tested for possible condensation.

Now, let us give the algorithm for finding the l-optimum partition of a general convex polygon.

ALGORITHM P

(I) Get all the potential h-arcs of the polygon by the one-sweep algorithm [6]. (All these arcs form a tree.)

(II) Append the degenerated arcs to the arc tree obtained in Step I and label all leaf nodes as "processed."

(III) Process the potential h-arcs, one by one, from the leaves to the root. (We cannot process a potential h-arc until all the potential h-arcs in its subtrees have been processed.) Let h_j be the arc to be processed, h_i be the arc immediately below h_j in the arc tree, X be the set of potential h-arcs immediately above h_j in the arc tree, and h_m be an arc in X such that

$$S(h_m\backslash h_m\text{'s ceiling}) = \max_{h_k\in X} S(h_k\backslash h_k\text{'s ceiling}).$$

Repeat
  Begin
1. [To delete those blocks of arcs which cannot exist in the l-optimum partition of the subpolygon between h_j and the leaf nodes in its subtrees.]
   While $S(h_m\backslash h_m\text{'s ceiling}) \geqq \min(w_j, w_j')$ do
     Begin
a.   Delete h_m and its descendants from the arc tree.
b.   Replace X by $X-\{h_m\}\cup h_m$'s ceiling and then update h_m accordingly.
     End.
2. [To locate the ceiling of h_j.]
   If $h_j \neq h_1$
   then
     While $S(h_j\backslash X) \leqq S(h_m\backslash h_m\text{'s ceiling})$ do
       Begin
a.   Comment: h_m is a son of h_j.

b. Combine h_m and all its descendants into h_j and calculate the combined supporting weight

$$S(h_j\backslash X - \{h_m\} \cup h_m\text{'s ceiling}).$$

c. Replace X by $X - \{h_m\} \cup h_m$'s ceiling and then update h_m accordingly.
End.

3. [Prepare to process next arc.]
If $h_j \neq h_1$
then
If h_i has a subtree which has not been processed then pick a subtree of h_i which has not been processed and apply Step II to this subtree recursively
else
Begin
Replace X by the arcs immediately above h_i in the arc tree, h_j by h_i and h_i by the arc immediately below h_i in the arc tree.
End.
End.
Until $(h_j = h_1)$.

(IV) Output the l-optimum partition consisting of the arcs which remain in the arc tree after Step II as h arcs. Then stop.

Using arguments similar to those in the proof of Theorem 8, we have the following theorem.

Theorem 13. *The partition produced by Algorithm* P *is l-optimum.*

Proof. Omitted.

One way to implement Algorithm P is to place all the potential h-arcs obtained in Step I in a linked tree. Each potential h-arc in the arc tree is again associated with a record variable similar to those described in § 2. We shall also initialize the ith element of the array CP to the quantity $w_1 : w_i$ for $2 \leqq i \leqq n$ and set CP[1] to zero. Hence, from our discussions in § 2, we know that we can calculate the supporting weights in a constant amount of time. Since we always test the arc with the largest supporting weight for possible deletion or condensation among all the arcs in X in Step II of the algorithm, we should keep track of the arcs in X and in each ceiling by means of the priority queues. When an arc h_m in X is deleted from the arc tree, we remove h_m from X, then we merge X and the ceiling of h_m into one priority queue. Similarly, when an arc h_m in X is condensed into h_j, we remove h_m from X and add it to the list of h_j's sons, then we merge X and the ceiling of h_m into one priority queue and set the ceiling of h_m, i.e. the third field of h_m, to NIL. Hence, it takes $O(\log n)$ time for each update of X to $X - \{h_m\} \cup h_m$'s ceiling in both Step II.1 and Step II.2.

THEOREM 9′. *Algorithm* P *runs in $O(n \log n)$ time.*

Proof. It takes $O(n)$ time to sweep around the monotone polygon twice, once to obtain all potential h-arcs in Step I and once to initialize the array CP. It also takes $O(n)$ time to append the degenerated arcs in the arc tree. There are two while loops in Step III, and it takes $O(\log n)$ time to execute either while loop once. Whenever the while loop in Step III.1 is executed once, a potential h-arc is deleted from the arc tree. Whenever the while loop in Step III.2 is executed once, a potential h-arc is removed from the arc tree and condensed to its father. Once an arc is removed or deleted from the list, it will never be considered again. Since there are at most $n-3$ arcs in the arc tree, Algorithm P can execute both while loops at most $n-3$ times.

So, it takes $O(n \log n)$ time to process all the potential h-arcs in Step III. Finally, it takes $O(n)$ time to output the l-optimum partition in Step IV. Hence, Algorithm P runs in $O(n \log n)$ time. □

4. Conclusions. In this paper, we have presented an $O(n \log n)$ algorithm to find the unique lexicographical smallest optimum partition of a general convex polygon. Both Algorithm M and Algorithm P have been implemented in Pascal [7]. We have also compared Algorithm P with the $O(n^3)$ dynamic programming algorithm and found that Algorithm P runs faster than the dynamic programming algorithm when n is greater than or equal to 7.

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] S. S. GODBOLE, *An efficient computation of matrix chain products*, IEEE Trans. Computers, C-22 (1973), pp. 864–866.

[3] H. HOROWITZ AND S. SAHNI, *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, MD 1978.

[4] T. C. HU AND M. T. SHING, *Computation of matrix chain products*, in 1981 Army Numerical Analysis and Computer Conferences, February 1981.

[5] ———, *An $O(n)$ algorithm to find a near-optimum partition of a convex polygon*, J. Algorithms, to appear.

[6] ———, *Computation of matrix chain products*, I, this Journal, 11 (1982), pp. 362–373.

[7] ———, *Computation of matrix chain products*, *Part* I, *Part* II, Report STAN-CS-81-875, Stanford Univ., Stanford, CA, Sept. 1981.

SIAM J. COMPUT.
Vol. 13, No. 2, May 1984

© 1984 Society for Industrial and Applied Mathematics
003

N BY N CHECKERS IS EXPTIME COMPLETE*

J. M. ROBSON†

Abstract. The game of Checkers can easily be generalized to be played on an N by N board and the complexity of deciding questions about positions regarded as a function of N. This paper considers mainly the question of whether a particular player can force a win from a given position and also the question of what is the best move in a given position. Each of these problems is shown to be complete in exponential time. This means that any algorithm to solve them must take time which rises exponentially with respect to some power of N and moreover that they are amongst the hardest problems with such a time bound. For instance if there are any problems solvable in exponential time but not in polynomial space, then these two problems are amongst them.

Key words. checkers, two person game, exponential time complete

1. Introduction.

1.1. N by N Checkers. This paper considers the complexity of algorithms to solve certain problems concerning the game of Checkers generalized to an N by N board. It is assumed that the reader is familiar with the rules of 8 by 8 Checkers. The generalization to an N by N board is fairly obvious except for the initial disposition of the pieces which does not concern us. In case the generalization is not obvious, [1] discusses the interesting points fully. It is assumed here that there is no rule which will cause a game to be declared a draw when, apart from this rule, one player could eventually force a win.

The forced capture rule will be of vital importance in the analysis and so it is repeated here: a player who has any capture move available on his/her turn must make one of the available capture moves and such a move is not complete until the capturing piece reaches a square from which no further capture is possible for it. Thus a player may be obliged to capture an indefinite number of pieces on a single move but if several captures are available, there is no obligation to choose the one which captures the most pieces.

1.2. Exptime completeness. The terminology of this paper is taken from [6] which contains a discussion of many related topics as well as the foundations of our arguments. The relevant definitions are:

A language is in exponential time (abbreviated Exptime) if it is recognized by a deterministic Turing machine with running time bounded by $O(c^{p(n)})$ for some c and p a polynomial where n is the length of the input string.

A language is log-complete in exponential time (abbreviated exponential time complete or Exptime complete) if it is in Exptime and every other language in Exptime is reducible to it by a Turing machine using $O(\log(n))$ workspace on inputs of length n. Since some languages are already known to be Exptime complete, to show that a language L is also Exptime complete, it suffices to show that L is in Exptime and that there is a reduction to L from some known Exptime complete language.

If a language L is shown to be Exptime complete, the principal conclusion which follows is that any deterministic algorithm to recognize L must take time $\Omega(c^{n^k})$ for some $c>1$ and $k>0$ since it is known that there are some languages in Exptime with this property. Moreover if it were shown, as most researchers would suspect, that

* Received by the editors August 25, 1981, and in revised form August 20, 1982.

† Department of Computer Science, Australian National University, Canberra, ACT 2600, Australia.

there are languages in Exptime which are not recognizable in polynomial space, then the same would be true of L.

1.3. Results. The main result which will be proved is that the set of all N by N Checkers positions from which White (say) can force a win is Exptime complete. This appears to be a considerable strengthening of the result of [1] that recognizing this set is P space hard.

Since the number of possible positions at N by N Checkers is easily seen to be $<5^{N^2}$, the proof that the language is in Exptime is simple and is omitted. Thus the bulk of the paper is devoted to the description of a reduction from a known Exptime complete language to Checkers positions. Finally §§ 8 and 9 will justify the claim that the reduction is logspace computable and draw some conclusions.

Similar Exptime completeness results are already known for Chess [2] and various artificial games [3], [6]. A proof of the same result for GO is currently in preparation, again strengthening a previous P space hardness result [4].

2. A global view of the Checkers positions. The overall form of the Checkers positions which will be of interest is illustrated in Fig. 1. In the centre a group A consisting of $|A|$ pieces is arranged in a configuration to be described in detail later so that moves there constitute a simulation of a game already known to be Exptime complete. Surrounding A and very distant from it is a spiral whose arms consist of parallel rows of Black and White kings four squares apart. The form of the spiral is shown in more detail in Fig. 2. (Squares denote kings and circles denote single pieces in most of the figures.) The number of circuits of the spiral and the distance from A to the inside of the spiral are $O(|A|)$ and $O(|A|^2)$ respectively. The inside of each arm of the spiral is four squares from the outside of the previous one and each arm has an even number of pieces of each color.

Consider what would happen if one player (Black say) was suddenly left with a large number of possible capture moves in A. White could capture all the Black pieces in the spiral in a small number of moves (independent of the length of the spiral) by

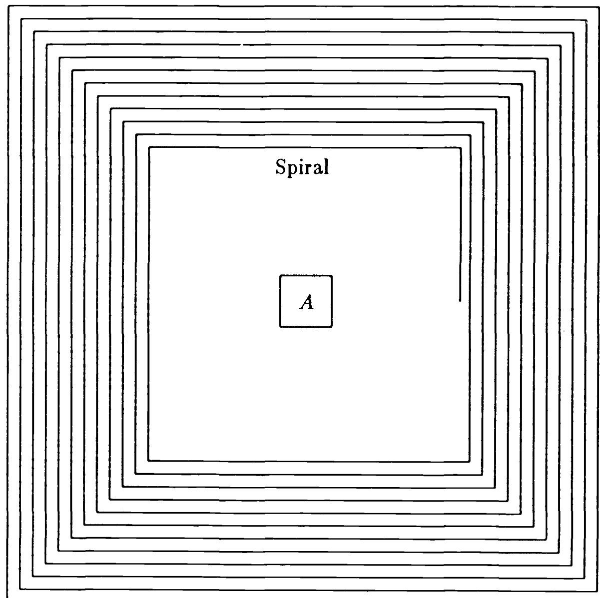


FIG. 1

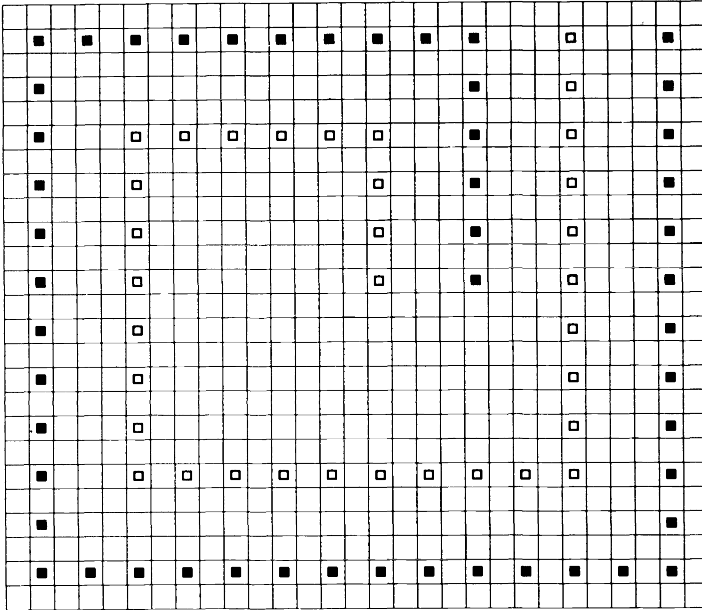


FIG. 2. *The start of the spiral.*

taking advantage of Black's being bound by the forced capture rule. Figure 3 illustrates one way in which White could do this. Having set up the pattern shown, White moves from A to B and then on his next move, whether or not Black captures the piece at B, White starts capturing along the path marked by Ds. This captures all the Black pieces in the spiral apart from one or two of those marked C which can then be mopped up in three or four more moves.

Thus White will have obtained a massive material advantage which we claim will be enough to enable him to force a win. Of course in general no material advantage guarantees a win if the pieces are badly placed but [1] showed that if a group of pieces like A is surrounded by $O(|A|)$ "picket lines" that is rectangles like those shown in Fig. 4 with at least four squares between them, then White can force a win whatever the details of A. Since converting White's spiral to a set of $O(|A|)$ picket lines involves moving $O(|A|)$ pieces, White can complete this process by moving pieces from the outer arms of the spiral in $O(|A|^2)$ moves before any Black pieces from A can approach and interfere with the process.

Of course Black will realize that, if he is left with a number of forced captures, White can force a win in this way. Accordingly Black may try to frustrate the plan by making moves in the spiral. But even if Black makes up to two moves in the spiral, White can still capture all Black's pieces in the spiral in a certain finite number of moves. This is because Black's moves merely split his spiral into three vulnerable sections each of which can be attacked and destroyed as in Fig. 3 and the bounded number of Black pieces remaining can then be mopped up individually. Instead of trying to determine the minimum number of moves which White may ever need after any pair of moves by Black, we simply call it X, and conclude that White can force a win provided, by play in A, he can force Black to make a sequence of X captures in A (with no White captures forced in A) before Black is able to make more than two noncapture moves outside A. In exactly the same way, Black can force a win if

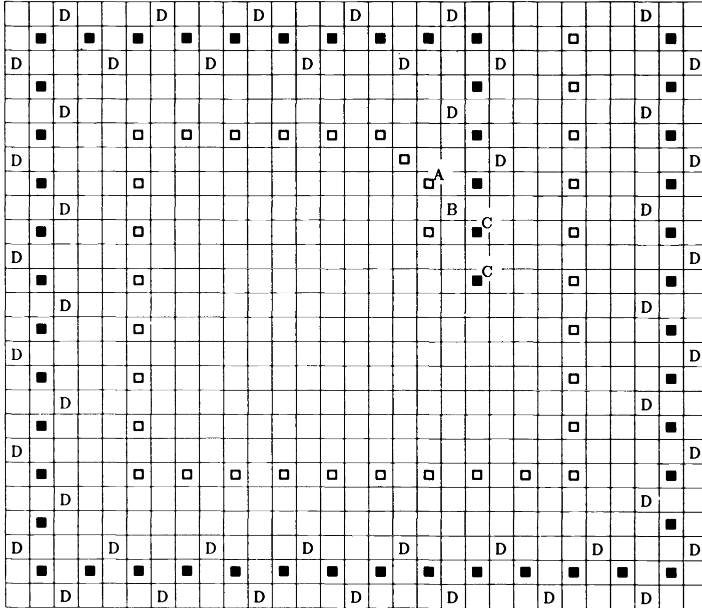


FIG. 3. *Attacking the spiral.*

White is forced to make the X captures in A. The structure of A will be such that there is no other way of forcing a win.

3. A known Exptime complete game. The reduction to Checkers starts from the game G3 shown in [6] to have an Exptime complete decision problem. The notation of that paper has been changed so that the names are more meaningful in the context of Checkers.

G3 is played by changing values of boolean variables W_1 to W_m and B_1 to B_m. Player W (or B) moves by changing the value of exactly one of the W (B) variables. Moves alternate between the two players and the result of a game is decided by two boolean expressions WWIN and BWIN in DNF over the variables. W wins immediately if his expression WWIN is ever true when B has just moved (and vice versa).

The aim of the reduction is to produce, given an instance of G3, a configuration to put at A in Fig. 1, which models the structure of the instance of G3 and is such that certain moves (called "normal play") simulate the playing of G3 and, if W wins at G3, then White can force X Black captures in A (and vice versa).

However, there is a complication in that a player may choose to "cheat" by making a legitimate Checkers move either in A or in the spiral which is not a normal simulation move. The solution to this is to modify the expressions WWIN and BWIN so that a player who "cheats" by not changing one of his variables on his move loses at once. In other words we must ensure than a normal W move always leaves WWIN true and a normal B move always leaves BWIN true. It is not obvious that this can be done for arbitrary WWIN and BWIN without increasing their length (when expressed in DNF) exponentially.

Fortunately, for the particular instances of G3 reached by the reduction to G3 in [6], this modification can be done. Their expressions WWIN and BWIN use a number of subexpressions a_i' over W_1 to W_m and b_i' over B_1 to B_m which change

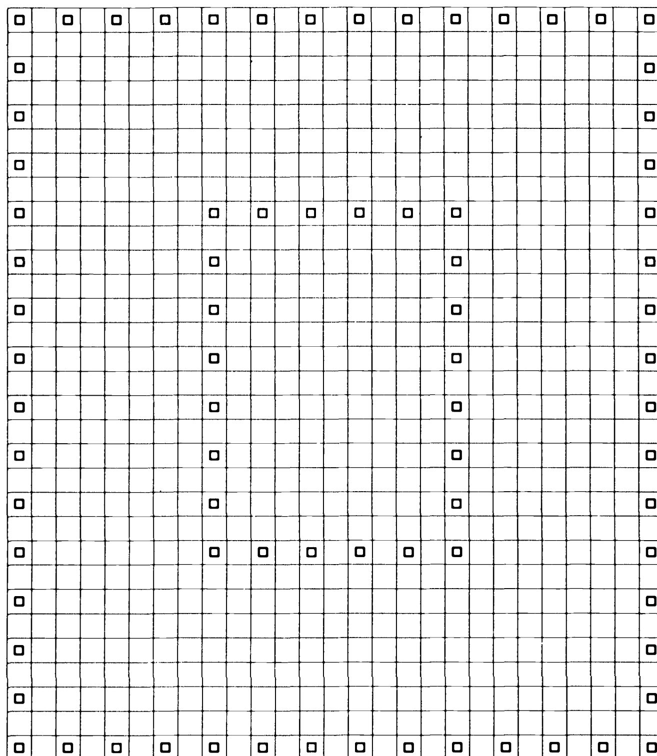


FIG. 4. *Picket lines.*

cyclically; that is as long as no player makes a stupid move which loses at once, every move by W sets exactly one a'_i true, the following move by B sets b'_i true (for the same i) and all other b' false, and the following move by W sets a'_{i+1} true et cetera (all subscripts being taken modulo $2m+2$). Hence after $W(B)$ moves WGONE (BGONE) is always true where

$$\text{WGONE} = \bigvee_{i=1}^{2m+2} (a'_{i+1} \wedge b'_i), \qquad \text{BGONE} = \bigvee_{i=1}^{2m+2} (a'_i \wedge b'_i)$$

and moreover no move by $W(B)$ leaves BGONE (WGONE) true. Hence, if we play G3 on the expressions WWIN $\vee$ WGONE and BWIN $\vee$ BGONE, the outcome will be the same as the original game but any attempt to cheat by not moving at G3 means that the offending player has lost.

The reduction to Checkers will be from these particular modified instances of G3 with the further trivial restriction that every disjunct in WWIN (BWIN) contains at least two $W(B)$ variables. Even with these restrictions the reduction will prove the Exptime completeness of Checkers.

4. The simulation of G3. This section gives a general description of the Checkers configuration to be placed at A in Fig. 1 to model an instance of G3 and describes the sequence of moves known as "normal play" which first simulates the game of G3 and then allows the winner of this game to force his opponent to make a sequence of X captures allowing the G3 winner to make his winning X moves in the Checkers spiral. Subsequent sections will describe certain parts of the configuration and play

in greater detail and examine the effect of deviation from normal play. We describe the position from the point of view of White who is assumed to play from the bottom of the board; the description from Black's point of view is obtained by interchanging "Black" with "White" and "B" with "W" and rotating all figures through 180 degrees.

Corresponding to each variable W_i is a "boolean controller" where White can move a king between two squares called T (representing true) and F (false). In normal play White and Black simply make moves in their respective boolean controllers representing W and B's moves at G3 until the simulated game of G3 is over.

Corresponding to each disjunct of WWIN is an "attack zone" providing the mechanism by which White can force the sequence of X captures by Black provided the disjunct is true. The W variables in the disjunct are called the "attack variables"; provided they all have the value required for the disjunct to be true, White can mount an "attack" in the zone. The B variables in the disjunct are called the "defence variables"; Black can successfully defend against White's attack provided *any* defence variable does *not* have the value required for the disjunct to be true. Thus the condition that White can mount an attack and Black can not defend against it is precisely the truth of the disjunct.

The connection between boolean controllers and attack zones is provided by a mechanism called "firing" the controller (or the variable). Firing a controller is a move within the controller by the "owner" of the variable which initiates a sequence of forced captures which will eventually affect an attack zone. Which attack zone is affected, is determined firstly by the value of the simulated variable when the controller is fired and then by choices made by the owner of the variable. The detailed structure of the expressions WWIN and BWIN is reflected in the possible paths provided from controllers to attack zones.

When White (about to move) has one of his WWIN disjuncts true, he fires the variables W_i occurring in it in order of increasing i and directs the capture sequences to the attack zone corresponding to the disjunct. Each capture sequence except the penultimate leaves White to make the next move and so able to fire the next variable. Eventually after the penultimate is fired, Black has a nonforced move. Now if Black had a defence variable with the right value, he could fire it and when the capture sequence reached the boolean controller, because of the changes produced by White's attack, Black could use it to produce X forced captures by White. Since Black has no such defence variable, he can only make delaying moves after which White fires his last attack variable, setting up X forced captures by Black and winning.

Figure 5 illustrates the geometry of the position for a trivial instance of G3. The diagonal lines represent paths of possible capture sequences and the places where two such paths cross or diverge are the "crossovers" and "forks" described later. The picture is slightly more complex than has been described above so as to give White a multiplicity of winning methods ensuring that he can still win even if Black "cheats"; there are two attack zones for each disjunct and the last attack variable has two possible capture paths to each of these attack zones; to allow for these four possible capture paths, the boolean controller has up to four exits which may be taken on firing it for each value of the variable.

The case illustrated is

$$\text{BWIN} = B_1 \wedge B_2 \wedge W_1 \wedge \sim W_2, \qquad \text{WWIN} = C_1 \vee C_2,$$

where

$$C_1 = W_1 \wedge W_2 \wedge B_1, \qquad C_2 = W_1 \wedge \sim W_2 \wedge \sim B_2.$$

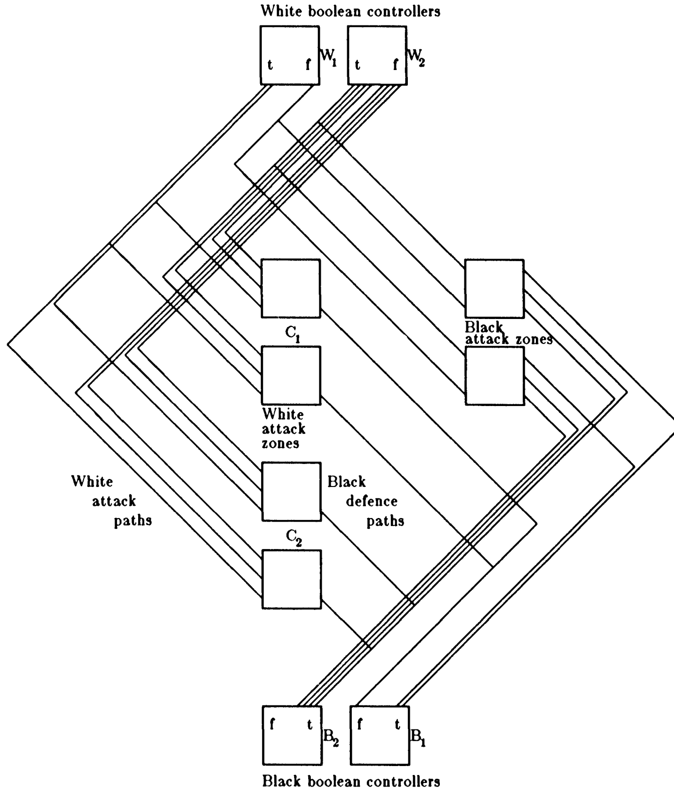


FIG. 5. *The simulation of* G3.

5. Capture sequences. This section starts the detailed description of the configuration outlined in § 4. The discussion continues to be given from White's point of view, so it describes only the White boolean controllers and attack zones and the potential capture paths shown in the top and left parts of Fig. 5 running from White boolean controllers to attack zones of both colors. The form of these potential capture paths is in general like that shown in Fig. 6. Any length of one of these paths can be traversed

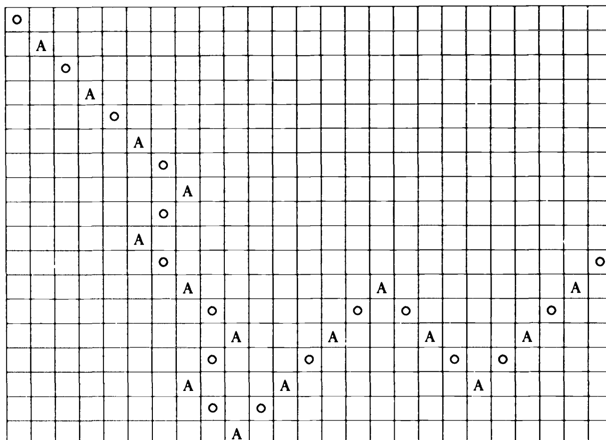


FIG. 6. *Potential capture path.*

(along the "*A*"s) in a single move by a Black king or by a single Black piece as long as the direction remains downwards. These simple paths are interrupted by a variety of "gadgets" within boolean controllers and attack zones and where they cross and fork. These gadgets are now described in detail.

5.1. Adjusting a capture sequence. Before a capture sequence can behave in the required way in a gadget, it may be necessary to change the color of the piece making the capture. Figure 7a shows how a capture by a Black piece or king, entering the figure at *D* and coming to rest at *C* can force a recapture by the White king at *B* which will leave the figure at *A* and can then enter a gadget.

It may also be necessary to "shift" a capture sequence to obtain the correct spatial relationship between two sequences. Figure 7b shows a capture by White, entering along the line of *E*s which forces a recapture by the Black king at *I* after which White must start a new capture with the king at *F* which leaves the diagram along the line of *G*s. This new sequence is displaced one square diagonally from the line of the original. A combination of up to two of these shifts with the more obvious zigzags of Fig. 6 can cause whatever displacement of a capture path is required.

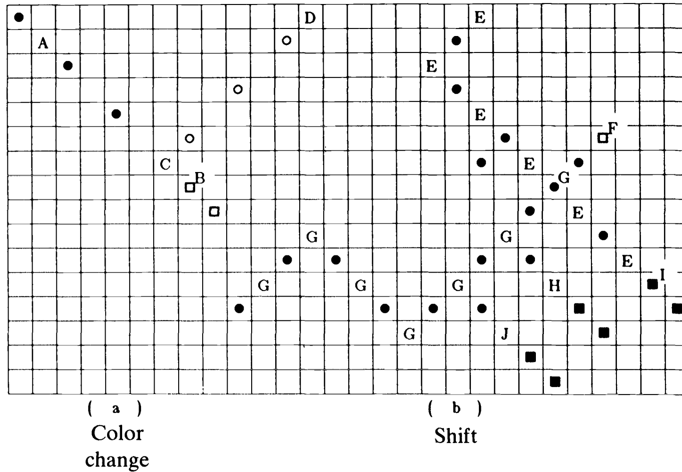


FIG. 7

These two "components", color changes and shifts, are to be added before gadgets as necessary to enable capture sequences to enter gadgets on the correct squares and with the correct color capturing. The reverse color change may also be necessary but the shift shown in Fig. 7b is the only one to be used on paths from White boolean controllers. (If the corresponding component to shift a Black capture sequence were used here, Black could take one of the "side turnings" to *H* or *J* nullifying the effect White wants from firing his controller. It is never to White's advantage to take these side turnings in normal play; the appendix points out why they are necessary.)

5.2. Crossovers and forks. Figure 8a shows how two potential captures from *A* to *a* and from *B* to *b* can cross each other without permitting the player making the capture to switch from one path to the other. The fact that after one of the captures has been made the other is impossible, is not significant in normal play.

Figure 8b shows a simple fork where a White king entering at *C* can choose between two possible exit routes *D* and *E*.

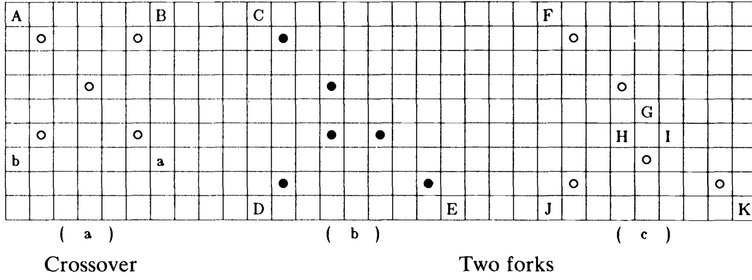


FIG. 8

Figure 8c shows a more complex fork to be used only in White boolean controllers. A single Black piece enters the fork by F and stops at G; now a White piece can go to either H or I and force a continuation in either of two directions leaving by J or K. Of course White also has the option of not moving to H or I and halting this capture sequence.

5.3. Enablers and delays. Figure 9 shows how one capture (crossing the figure diagonally from A) makes it possible for a later one to proceed (from B). This gadget will be used in attack zones.

Figures 10a and 10b show how a capture by Black may leave behind a forced capture by White or Black respectively. In each case, Black captures down the line of "A"s and this leaves a piece (B or C) able to capture by jumping into a newly vacant square. These will be called respectively White and Black "delays" and lines of X of them will be used in boolean controllers and attack zones. Such a line of X

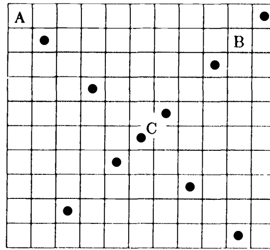


FIG. 9. *Enabler.*

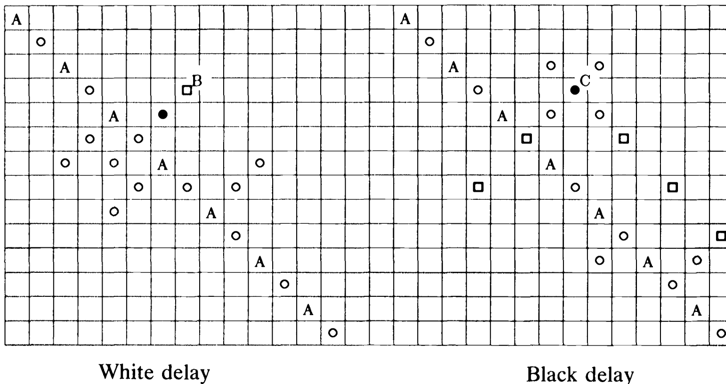


FIG. 10

delays will be represented in diagrams by MBD (Multiple Black Delay) or MWD (Multiple White Delay).

6. Boolean controllers and attack zones. Figure 11 illustrates a White boolean controller and the symbolism used for it in Fig. 5. The lines in the middle part of the figure represent lines of White pieces able to be captured in a single move, broken only by forks of the type shown in Fig. 8c. A White king is at either T or F.

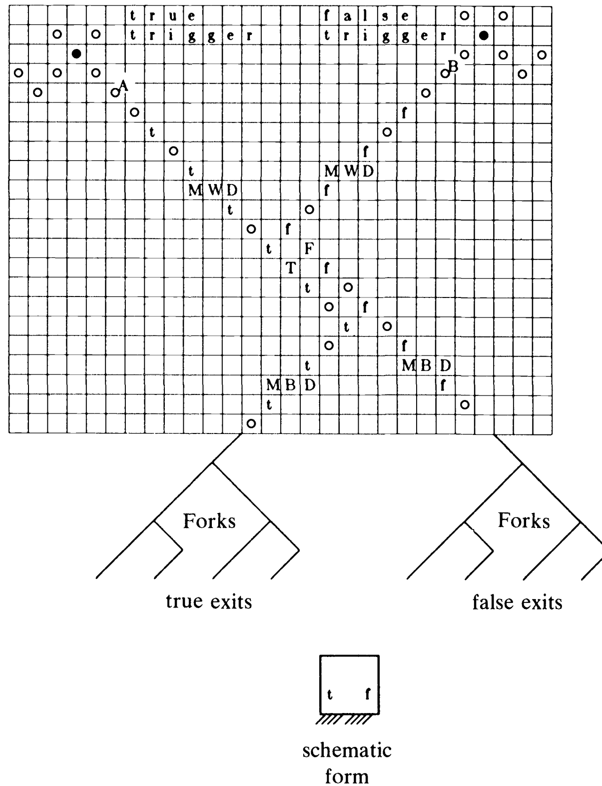


FIG. 11. *White boolean controller.*

In normal play White moves his king between T and F to simulate changes in the corresponding G3 variable between true and false. To fire the controller when the king is at T, White moves the piece at A in the "true trigger." This allows (forces) Black to capture down the path marked t, through White delays, the king at T and Black delays before coming to rest at the first fork in the true side of the controller. Now White and Black are forced to spend their next X moves making the captures in their respective delays after which White makes his moves in the forks forcing a Black capture move to emerge from the controller at whichever of the true exits White chooses. The process of firing the controller when the king is at F is similar but uses the false trigger to produce a capture emerging from one of the false exits.

Once a Black capture energes from the controller, a sequence of forced captures continues through forks and crossovers until eventually it reaches an attack zone. Since White makes the choices at all the forks, he can choose to direct the capture sequence to any of the attack zones to which a path exists.

From the true exits of the controller for W_i, there will be paths to all attack zones for disjuncts in WWIN which contain W_i (in nonnegated form) and to zones for disjuncts in BWIN containing $\sim W_i$. Conversely from false exits there will be paths to WWIN disjuncts containing $\sim W_i$ and BWIN disjuncts containing W_i.

Figure 12 illustrates the form of an attack zone for a WWIN disjunct with three attack variables and two defence variables. The figure incorporates several enablers

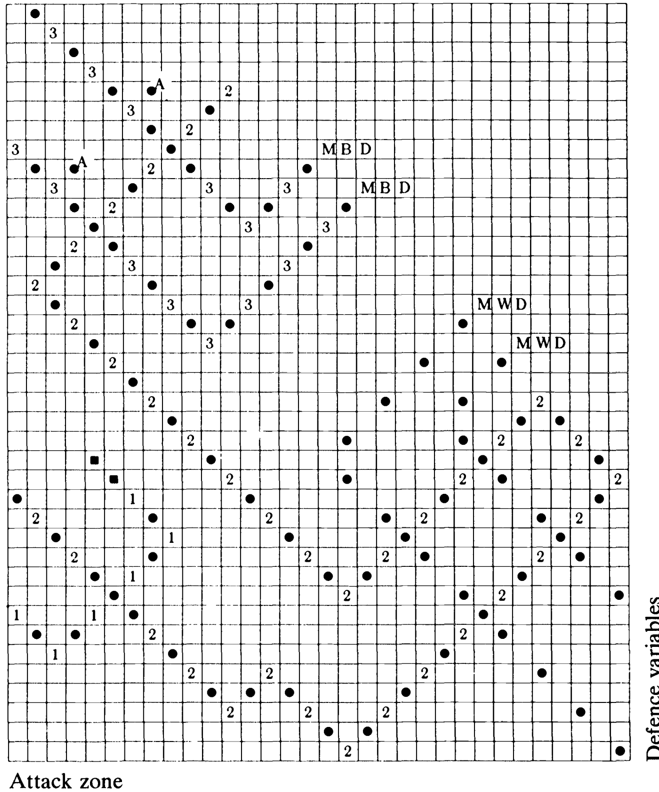


FIG. 12

as discussed in § 5.3. All capture sequences reaching an attack zone cause a White king to enter the zone along one of the lines of Black pieces. The delays in this zone, being astride lines of Black pieces are those obtained by reversing colors and directions from those of Fig. 10.

We recall from § 4 that the attack is supposed to succeed if every attack variable has the correct value to fire at the zone and no defence variable has the correct value. Thus the attack zone is constructed so that *all* attack variables must fire for the attack's success but *any* defence variable firing is sufficient to prevent it. The way in which the illustrated zone achieves this is as follows.

Each attack variable, except the last two, traverses a path like that marked "1" enabling its successor and terminating in a simple Black delay. The penultimate attack variable traverses a path like that marked "2" enabling all the defence variables and the last attack variable before terminating leaving Black to move. Now if Black had defence variable able to fire at this zone, he could fire it and win by activating one of the Multiple White Delays. Otherwise White will win by firing his last attack variable

and causing a Multiple Black Delay: Black cannot use his free move to sabotage the path of this last attack variable because we have provided White with two separate paths.

7. Abnormal play. From now on it is assumed that W has a forced win at G3. It should be clear from the above discussion that, as long as Black plays moves which simulate G3 moves by B, White will eventually have some attack zone where he is ready to fire all his attack variables and Black has no defence variable ready to fire. Thus White will win unless Black finds some "unexpected" move which either disrupts the simulation of G3 or sabotages the attack in the attack zone. This section will rule out that possibility, completing the proof of the correctness of the reduction.

Moves by Black other than those simulating G3 moves will be classified as either "irrelevant" moves which White can ignore since they do not hinder his winning strategy, or "catastrophic" moves which enable White to win immediately by forcing X Black captures, or the majority of "cheating" moves which can be regarded as not altering a G3 variable and so enable White to win by using the disjunct of WGONE which was true after his move.

7.1. Irrelevant moves. These are the very small number of possible moves which, after a sequence of forced captures by each side, leave Black again with the next nonforced move so that he can resume the G3 simulation. They consist of firing a Black boolean controller with the variable having the correct value (e.g. using the true trigger with the king at T) followed by either halting the captures in the forks of the controller or allowing capture sequences to reach Black attack zones in the correct enabling orders but not as far as the penultimate attack variable in any zone.

These moves in no way interfere with White's ability eventually to mount a successful attack. They merely prevent Black from later altering the values of the affected G3 variables, an effect which cannot be to Black's advantage.

7.2. Catastrophic moves. These are two types of move by Black which are punished by the immediate activation of a Multiple Black Delay.

The first type is for Black to fire a boolean controller using the "wrong" trigger; for instance to use the true trigger with his king at F. The resulting capture by White goes through a Multiple Black Delay but stops before T and so does not activate the compensating Multiple White Delay. (Note that these delays are essential to the whole argument; if they were not there, Black could now move his king from F to T.)

The second type of catastrophic move may occur if Black fires attack variables in the correct enabling order so that eventually the penultimate attack variable enters some attack zone and enables the last attack variable. Now by the assumption that W has a winning strategy at G3 (and that White has been simulating it), either White has a defence variable ready to fire or Black does not have his last attack variable ready. In the first case, Black's move has been catastrophic and White will fire his defence variable and activate a Multiple Black Delay. In the other case, Black's last move has simply wasted time and White treats it as a "cheating" move of the type discussed below.

7.3. Cheating moves. This category covers all other Black moves. That is, all moves in the spiral, in boolean controllers, attack zones and other gadgets or in the capture paths connecting them except for (i) normal simulation of G3 moves and (ii) the irrelevant and catastrophic moves described above. In all these cases White responds by mounting an attack in an attack zone whose disjunct was true before Black's move.

The fact that White can always do this depends on three properties of moves in the various gadgets etc., which can only be verified by tedious checking of many cases:

(1) Any cheating move loses a tempo. That is although it may result in a sequence of forced captures, after this sequence White is the first player to have a nonforced move. (This may depend on White making the correct choice when forced to choose between two possible captures.)

(2) Any such move, together with its following forced captures, causes at most localized damage to the configurations enabling White to mount an attack. To be more precise, although it may destroy one potential capture path outside a boolean controller, it cannot

(a) prevent White from firing a boolean controller and choosing which exit is taken,

or

(b) prevent a Black delay from working,

or

(c) damage two potential capture paths from White boolean controllers (except at a fork where the two diverge).

Properties (1) and (2) are sufficient to ensure that after any cheating move by Black, White can still choose an undamaged attack zone corresponding to his true disjunct, which has all the capture paths to it also undamaged. Thus White can fire the attack variables for that attack zone at least up to the penultimate. Then Black has one last opportunity to stave off defeat. Property (1) may now no longer hold; Black may be able to make a move close to his original cheating move which forces a White capture after which Black has a free move and may repeat this process. Calling such a sequence of moves a "supplementary sequence" we now state the last required property:

(3) One cheating move by Black together with a supplementary sequence cannot

(a) prevent White from firing a boolean controller and choosing which exit is taken,

or

(b) damage more than two potential capture paths, from White boolean controllers,

or

(c) regain the lost tempo by causing two forced White captures,

or

(d) affect the type of Black delay occurring in White boolean controllers.

Since White has two potential capture paths to his chosen attack zone for his last attack variable, this ensures that he can eventually activate one of the Multiple Black Delays and win.

The apparent over-complexity of many of the gadgets stems from the need to ensure that (1), (2) and (3) hold. Most of the checking of these properties has been done by a program which did detect one error in an earlier version of Fig. 10b. The appendix points out a few ways in which simpler gadgets are inadequate.

Two points should be mentioned here since they concern the general structure of the attack zone rather than the details of the gadgets within the zone: to ensure that 3(c) holds, the path of a defence variable within attack zones passes through two enablers each of which is opened by an attack variable; otherwise Black could win by moving the piece at C of Fig. 9 (in the enabler for one of his defence variables) as his initial cheating move and later firing that defence variable as his supplementary sequence. Secondly the orientation of the enablers within the attack zone ensures that

either player firing a variable which has not yet been enabled loses a tempo as required by property (1) and the Black pieces marked "A" in Fig. 12 prevent a tricky play by White to try to regain that tempo.

8. Summary. Apart from some minor details concerning the exact placement of the gadgets on the Checkers board and the necessary gaps between various gadgets and the spiral, the Checkers position corresponding to a G3 position has been fully described. Moreover it has been shown that the first player to deviate, except in a few insignificant ways, from the moves simulating G3 loses at Checkers and that, if the simulated game of G3 ends, the winner of that game wins at Checkers. This completes the demonstration of the reduction from G3 to N by N Checkers.

If the length of the description of the instance of G3 is n, then both the number of variables and the number of disjuncts in WWIN and BWIN are $O(n)$. Hence the area A in which the G3 simulation takes place can be enclosed in a square of side $O(n)$ containing $O(n^2)$ pieces. Hence from the discussion in § 2, the whole Checkers position can certainly be placed on a board of side $O(n^4)$.

The regular way in which the placement of pieces on the board depends on the disjuncts of WWIN and BWIN means that the Checkers position could be output by a RAM program using $O(1)$ variables each restricted to a range of $O(n^4)$ values. Finally any reasonable Turing machine simulation of this RAM program will yield the required $O(\log n)$ bound on the workspace complexity of the reduction.

9. Conclusions and some open questions. This completes the proof of the main result that the set of positions at N by N Checkers from which White can force a win is Exptime complete. Thus any algorithm to recognize these positions must have running time $\Omega(c^{N^k})$ for some $c > 1$ and $k > 0$. This conclusion unfortunately reveals nothing about the difficulty of such a recognition algorithm for the 8 by 8 case; an 8 by 8 board would not even hold one boolean controller. This lack of information on particular finite cases is of course common to such lower bound complexity results.

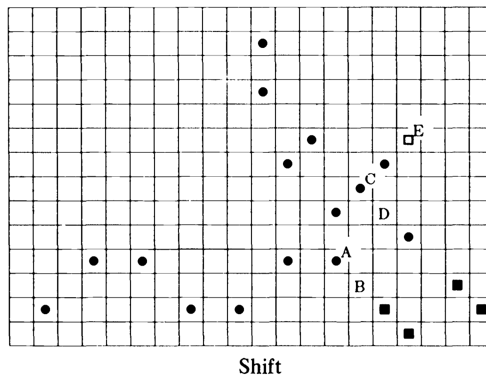
Two other conclusions follow. Firstly any algorithm to decide an optimal move must also require exponential time (by a fairly obvious argument given in [5]). Secondly by following the chain of reductions to Checkers from an arbitrary exponential time Turing machine computation [6], it follows that the number of moves needed to force a win against optimal delaying defensive play also rises exponentially with N. Thus a conventional minimax algorithm has exponential space complexity; of course that does not of itself preclude the possibility of a polynomial space algorithm.

Having shown that Checkers has intractable decision problems despite its simplicity in each player having only two types of piece, one naturally wonders whether the problems would remain intractable if only one type of piece existed. For a game with only kings, it seems likely that a reduction similar to the one given here, but with different gadgets, would prove the same results. On the other hand, in a game played only with single pieces which, if they reached the far end of the board, were not promoted to kings and so became immobile, the length of a game would be bounded by a polynomial in the board size giving decision problems solvable in polynomial space. Another interesting question is whether the forced capture rule, which has been used so heavily, is in fact essential to the conclusion; the methods used in this paper do not appear to throw much light on that question.

Appendix. This appendix discusses how the complexity of some of the gadgets used in the reduction is related to the need for them to satisfy properties (1), (2) and (3) of § 7. In fact they were designed to satisfy a slightly stronger version of (3b),

namely that no cheating move followed by any supplementary sequence within a gadget should cause any repercussions outside the gadget. This "locality" property makes the checking of properties (1) to (3) much easier whether it is done by hand or by machine. No claim is made that the gadgets used are the simplest possible with the required properties. However four examples will be given of how apparently superfluous pieces within gadgets prevent breaches of the properties. This explanation is not part of the proof of the results of the paper but it may make § 7 clearer.

The first example is in Fig. 13 which shows a shift modified by removing one of the side turnings of the original. Now Black can move from A to B removing the second side turning and later as a supplementary move, move from C to D. The resulting capture by the White king at E will cause a capture sequence which may pass through several crossovers certainly invalidating property (3b) and possibly (3c).



Shift

FIG. 13

The remaining three examples are all contained in Fig. 14 which shows the delays slightly modified. The White delay has had two White pieces removed from A and B. The removal of the piece from A enables White to move from C to A forcing a capture by the Black piece D, invalidating property (1). Of course White would not do this in a White delay in a White boolean controller but if White can do it there, then Black can make a similar move in all the Black delays in White attack zones.

Returning to the White delay, Black can move from D to E (cheating) and then E to F (supplementary). The absence of the White piece at B now forces White to capture with G invalidating property (3a).

Finally in the Black delay, Fig. 14 has dropped a White piece from H. The need for this piece at H was discovered by computer testing which found that this Black delay violated the locality property. White can make cheating and supplementary moves which cause nonlocal effects as follows:

(i) move from I to J,

(ii) move out from K forcing Black to capture from L to K and I leaving White still able to move,

(iii) move from M to N forcing Black to capture from I to M,

(iv) move from P to H forcing Black to capture from M to P,

(v) move from Q to R forcing Black to capture from P to Q, et cetera.

In fact it appears that this last modification does not invalidate the reduction. However, it seems easier to design gadgets which satisfy the locality property than to

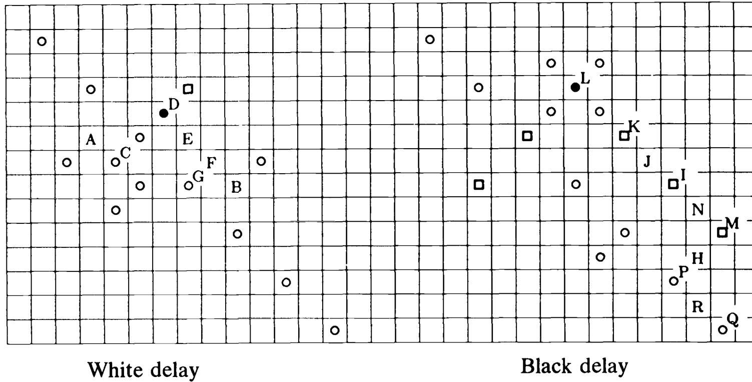


FIG. 14

justify the claim that it is irrelevant by arguments about how one gadget may affect its neighbors.

10. Acknowledgment. I would like to thank the anonymous referee of an earlier version of this paper for many helpful comments, particularly for the suggestion of a simplified version of the boolean controller.

Note added in proof. The Exptime completeness result for GO referred to in § 1.3 has been presented at the 1983 IFIP World Computer Congress.

REFERENCES

[1] A. S. FRAENKEL, M. R. GAREY, D. S. JOHNSON, T. SCHAEFER AND Y. YESHA, *The complexity of Checkers on an N by N board, preliminary report*, Proc. 19th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, 1978, pp. 55–64.

[2] A. S. FRAENKEL AND D. LICHTENSTEIN, *Computing a perfect strategy for n by n chess requires time exponential in n*, J. Combin. Theory, 31 (1981), pp. 199–214.

[3] T. KASAI, A. ADACHI AND S. IWATA, *Classes of pebble games and complete problems*, this Journal, 18, (1979), pp. 574–586.

[4] D. LICHTENSTEIN AND M. SIPSER, *Go is polynomial-space hard*, J. Assoc. Comput. Mach., 27 (1980), pp. 393–401.

[5] J. M. ROBSON, *Optimal storage allocation decisions require exponential time*, Australian Computer Science Communications, 3 (1981), pp. 162–172.

[6] L. J. STOCKMEYER AND A. K. CHANDRA, *Provably difficult combinatorial games*, this Journal, 8 (1979), pp. 151–174.

SIAM J. COMPUT.
Vol. 13, No. 2, May 1984

© 1984 Society for Industrial and Applied Mathematics
004

PARALLEL SOLUTION OF CERTAIN TOEPLITZ LINEAR SYSTEMS*

DARIO BINI†

Abstract. Using the concept of approximate algorithm it is shown that $6 \log n + 6$ parallel steps and $2n$ processors suffice to approximate, with any precision, the solution of a linear system with an $n \times n$ triangular Toeplitz matrix A. Moreover, $7 \log n + 7$ steps are sufficient for an exact computation, whereas the number of processors is increased to $(\frac{5}{2})n^2$. If A is also banded and k is its bandwidth, the number of processors is reduced to $(\frac{5}{2})n\ (k+1)$. Two applications are shown. It is proved that if B is any matrix belonging to the algebra generated over the complex field by a given $n \times n$ matrix, then the system $Bx = b$ can be solved with no more than $9 \log n + 4$ steps with $O(n^2)$ processors. It is proved that, given a Toeplitz matrix $A = (a_{i,j})$ such that $a_{i,j} = 0$ if $i - j > k$ or $j - i > h$, $a_{k,1} \neq 0$, then $13 \log n + O(\log^2 k)$ steps and $\max((\frac{5}{2})n\ (k+h),\ n(n+1)/2)$ processors are sufficient to solve the system $Ax = b$. Such algorithms work under the sole condition $\det A \neq 0$.

Key words. parallel matrix inversion, Toeplitz matrices, approximate algorithms, band matrices

1. Introduction. The evaluation of the complexity of $n \times n$ parallel matrix inversion is as hard to solve as it is well known. The algorithm shown by Csanky [10] yields an $O(\log^2 n)$ complexity by using $O(n^4)$ processors, but it is mainly of theoretic interest and so far the best lower bound known is $O(\log n)$ steps. Even for triangular matrices the best known algorithm employs $O(\log^2 n)$ parallel steps with $O(n^3)$ processors [14], [15], [6], [8]. Stronger structures such as that of Toeplitz (a Toeplitz matrix $A = (a_{i,j})$ is defined by $a_{i,j} = \alpha_{i-j}$) reduce the number of processors but not the number of steps [7]. Furthermore there are few classes of matrices for which there is an algorithm for parallel inversion in $O(\log n)$ steps.

Great progress has been made, after about a 9-year lull, in the analysis of the complexity of matrix multiplication and matrix inversion for sequential algorithms [4], [16], [18], [9], by introducing approximate algorithms [5]. Approximation can effectively reduce complexity and approximate algorithms can be turned into exact algorithms by the interpolation technique [2], slightly increasing the overall complexity. This technique seems to be ideal for parallel processing as it involves solving a set of k problems, which differ only in the values that a parameter can assume, and taking linear combinations of the "wrong" solutions in order to find the right one.

This paper deals with the problem of the parallel inversion of a triangular Toeplitz matrix by using approximate algorithms. It develops a result given in [1], concerning the optimal approximation of certain Toeplitz bilinear forms. We introduce the algebra $\mathcal{T}_\varepsilon$, made by simultaneously diagonalizable matrices, which approximates with any precision the algebra $\mathcal{T}_0$ of triangular Toeplitz matrices (§ 2). We show, in § 3, that the computation of the inverse of an $n \times n$ matrix belonging to $\mathcal{T}_\varepsilon$, as well as the approximation of the inverse of a triangular Toeplitz matrix, can be accomplished in $6 \log n + 3$ parallel steps using $2n$ processors. Here and throughout the paper we assume $\log n = \lceil \log_2 n \rceil$.

Furthermore, the complexity of the resolution of the associated linear system increases to $6 \log n + 6$ parallel steps while the number of processors remains unchanged. Using an interpolation technique we convert these approximate algorithms into exact ones, leaving unchanged the asymptotic complexity but increasing the number of processors. We prove the following result. If $A = (a_{i,j})$ is an upper triangular Toeplitz matrix such that $a_{i,j} = 0$ if $j - i > k$ and $a_{k,1} \neq 0$ (band matrix with bandwidth

* Received by the editors June 2, 1982, and in revised form January 25, 1983.
† Dipartimento di Matematica, Università di Pisa, 56100 Pisa, Italy.

$k+1$), then $6\log n+\log(k+1)+4$ steps suffice to compute exactly A^{-1} by using $\frac{5}{2}n\,(k+1)$ processors, while $6\log n+\log(k+1)+7$ steps are sufficient to solve the system $Ax=b$. Where $k=n-1$ then $7\log n+4$ steps and $\frac{5}{2}n^2$ processors suffice to invert a $n\times n$ triangular Toeplitz matrix versus the $\log^2 n+2\log n-1$ steps and $n^2/4$ processors required by the best algorithm to date [7]. The time complexity is drastically reduced by increasing the number of processors of a multiplicative factor. The threshold value n_0, for which this algorithm is better than the known one if $n>n_0$, is quite small, namely $n_0=32$.

Two applications of this result are shown in § 4. In the first, which is mainly theoretical, we widen the class of matrices whose algorithms are known for parallel matrix inversion running in $O(\log n)$ time. We show that any matrix belonging to the algebra generated over the complex field by any given matrix can be inverted in $O(\log n)$ steps by using $O(n^2)$ processors. In the second application we show how to use the results of § 3, together with the bordering technique [3], to construct fast algorithms for the parallel solution of linear systems with an unbalanced band Toeplitz matrix. If $A=(a_{i,j})$ is a nonsingular Toeplitz matrix such that $a_{i,j}=0$ if $j-i>k$ or $i-j>h$, $k\leqq h$ and $a_{k,1}\neq 0$ then the system $Ax=b$ can be solved in $13\log n+O(\log^2 k)$ parallel steps by using $\frac{5}{2}n(h+k)$ processors or in $8\log n+O(\log^2 k)$ parallel steps by using $\max(\frac{5}{2}n(h+k), n(n+1)/2)$ processors. In particular, for a Toeplitz matrix in Hessenberg form we have $9\log n+7$ steps and $3n^2$ processors. Where $h=k$, that is, when the band is balanced, this algorithm can be compared to the algorithms shown by Grcar and Sameh in [13]. This algorithm works without any restrictive conditions, whereas each of the three algorithms in [13] requires the positive definiteness or the nonsingularity of the leading principal submatrices. Furthermore, this algorithm is faster than the second and third algorithm in [13] while the number of processors is slightly increased.

2. Preliminaries. Consider the following $n\times n$ matrix:

$$H_\varepsilon=(h_{i,j}^{(\varepsilon)}),\qquad h_{i,j}^{(\varepsilon)}=\begin{cases}1 & \text{if } j=i+1,\\ \varepsilon & \text{if } i=n, j=1,\\ 0 & \text{elsewhere,}\end{cases}$$

and set $\mathcal{T}_\varepsilon$ for the algebra generated over the real field $\mathbf{R}$ by the matrix H_ε. Then $\mathcal{T}_0$ is the class of *upper triangular Toeplitz matrices*, while $\mathcal{T}_1$ is the class of *circulant matrices* [12]. The following is a matrix representing the class $\mathcal{T}_\varepsilon$ in the case $n=4$.

$$\begin{bmatrix} a_1 & a_2 & a_3 & a_4\\ \varepsilon a_4 & a_1 & a_2 & a_3\\ \varepsilon a_3 & \varepsilon a_4 & a_1 & a_2\\ \varepsilon a_2 & \varepsilon a_3 & \varepsilon a_4 & a_1\end{bmatrix}.$$

The matrices belonging to $\mathcal{T}_\varepsilon$, used in [1] to evaluate the approximate complexity of certain sets of bilinear forms, have a common set of n independent eigenvectors which are strongly related to the eigenvectors of $\mathcal{T}_1$. This will be proved in Proposition 2.1 and will require some properties of circulant matrices.

Let $\Omega=(\omega_{i,j})$ be the Fourier matrix of order n defined by $\omega_{i,j}=\omega^{(i-1)(j-1)}/\sqrt{n}$, $(1\leqq i,j\leqq n)$, where ω is a primitive nth root of unity, e.g. $\omega=\exp(2\pi\mathbf{i}/n)$ ($\mathbf{i}$ is the imaginary unit). Then Ω is a unitary matrix, that is, $\Omega\Omega^H=I$ (here Ω^H is the transpose conjugate of Ω). Moreover the following result holds.

THEOREM 1 [14]. *Let $A\in\mathcal{T}_1$ be a matrix whose first row is a^T; we then have $\Omega^H A\Omega=\mathrm{Diag}(d_1,d_2,\cdots,d_n)$, where $d=(d_i)$ is given by $d=\sqrt{n}\,\Omega a$.*

Now consider the matrix $D_\varepsilon = \mathrm{Diag}\,(1, \delta, \delta^2, \cdots, \delta^{n-1})$, where $\delta = \sqrt[n]{\varepsilon}$. It is easy to check the following relations:

$$H_\varepsilon = \delta D_\varepsilon H_1 D_\varepsilon^{-1}, \qquad H_\varepsilon^i = \delta^i D_\varepsilon H_1^i D_\varepsilon^{-1}. \tag{2.1}$$

Since $\mathcal{T}_\varepsilon$ is the algebra generated by H_ε over $\mathbf{R}$, if $A_\varepsilon \in \mathcal{T}_\varepsilon$ is the matrix whose first row is $a^T = (a_i)$ then

$$A_\varepsilon = \sum_{i=0}^{n-1} a_{i+1} H_\varepsilon^i.$$

Therefore, setting for $B \in \mathcal{T}_1$ the matrix whose first row is b^T, from (2.1) we have

$$A_\varepsilon = D_\varepsilon B D_\varepsilon^{-1}, \qquad b = D_\varepsilon a. \tag{2.2}$$

PROPOSITION 2.1. *Let $A_\varepsilon \in \mathcal{T}_\varepsilon$ be a matrix whose first row is a^T. Then we have $\Omega^H D_\varepsilon^{-1} A_\varepsilon D_\varepsilon \Omega = \mathrm{Diag}\,(d_1, d_2, \cdots, d_n)$, where $d = (d_i)$ is given by $d = \sqrt{n}\,\Omega D_\varepsilon a$.*

Proof. The proposition holds from Theorem 1 in the light of (2.2).

It is worth pointing out that the matrices belonging to $\mathcal{T}_0$ do not have a common set of n eigenvectors; in other words, they cannot be simultaneously diagonalized by a similarity transformation. However they can be approximated, with any precision, by the matrices belonging to $\mathcal{T}_\varepsilon$, $\varepsilon \neq 0$, which, for Proposition 2.1, have a common set of eigenvectors. This fact will be used in the next section for computational purposes. First it is necessary to state the following result.

PROPOSITION 2.2 [17]. *The matrix-vector product Ωx, as well as $\Omega^H x$, where x is a complex n-vector, can be computed by using $3 \log n$ parallel steps with $2n$ processors.*

COROLLARY 2.1. *Let x, y be two real vectors; then the pairs of products $(\Omega x, \Omega y)$,* as well as *$(\Omega^H x, \Omega^H y)$ and $(\Omega x, \Omega^H y)$, can be computed by using $3 \log n + 3$ steps with $2n$ processors.*

Proof. Let $z = x + \mathbf{i}y$, $Z = \Omega z$, $X = \Omega x$, $Y = \Omega y$; since Ω is symmetric and unitary and $P = \Omega^2$ is the permutation matrix such that $p_{i,j} = 1$ if $i - j = 1 \bmod n$, we have $X = \Omega(z^* + z)/2 = (\Omega^2 \Omega^H z^* + \Omega z)/2 = (PZ^* + Z)/2$, where z^* is the complex conjugate of z. Analogously we get $Y = (PZ^* - Z)/(2\mathbf{i})$. We can thus proceed in the following stages:

	steps	processors
1) compute $z = x + \mathbf{i}y$,	1	$2n$
2) compute $\hat{Z} = \frac{1}{2}\Omega z$,	$3 \log n$	$2n$
3) compute $X = P\hat{Z}^* + \hat{Z}$,	1	$2n$
4) compute $Y = P\hat{Z}^* - \hat{Z}$,	1	$2n$

The overall computational cost is given by $3 \log n + 3$ steps, $2n$ processors. Analogously we proceed in the other cases since $\Omega^H y = (\Omega y)^*$.

3. Main results. A first computational result, concerning the resolution of a linear system with a Toeplitz matrix of a certain form, can be obtained from Propositions 2.1 and 2.2. In the analysis of the parallel complexity we do not count the cost of the evaluation of $\omega_{i,j}$, δ^{i-1} and $\sqrt{n}\,\delta^{i-1}$, $1 \leq i, j \leq n$, since these constants do not depend on the input variables of the problem.

PROPOSITION 3.1. *The system $A_\varepsilon x = b$, where $A_\varepsilon \in \mathcal{T}_\varepsilon$, $\varepsilon \in \mathbf{R}$, $\varepsilon \neq 0$, $\det A_\varepsilon \neq 0$, can be solved by using $6 \log n + 6$ parallel steps with $2n$ processors.*

Proof. Let a^T be the first row of A_ε. From Proposition 2.1 we have $A_\varepsilon^{-1} = D_\varepsilon \Omega D^{-1} \Omega^H D_\varepsilon^{-1}$, $D = \mathrm{Diag}(d_1, d_2, \cdots, d_n)$, $d = (d_i)$, $d = \sqrt{n}\Omega D_\varepsilon a$. Therefore $x = A_\varepsilon^{-1} b$ can be computed in the following stages:

	steps	processors
1) compute $u^{(1)} = D_\varepsilon^{-1} b$, $v^{(1)} = \sqrt{n} D_\varepsilon a$,	1	$2n$
2) compute $u^{(2)} = \Omega^H u^{(1)}$, $d = \Omega v^{(1)}$,	$3 \log n + 3$	$2n$
3) compute $u^{(3)} = D^{-1} u^{(2)}$,	1	n
4) compute $u^{(4)} = \Omega u^{(3)}$,	$3 \log n$	$2n$
5) compute $x = D_\varepsilon u^{(4)}$,	1	n

The evaluation of the number of steps and processors at stages 2 and 4 was attained by using Corollary 2.1 and Proposition 2.2 respectively. The overall computational cost is given by $6 \log n + 6$ steps, $2n$ processors.

Observe that, in order to compute A_ε^{-1}, it is sufficient to solve the system $A_\varepsilon x = b$, where $b^T = (0, 0, \cdots, 0, 1)$. In fact the last column of A_ε^{-1} generates all the elements of the matrix. Now the vector $u^{(2)} = \Omega D_\varepsilon^{-1} b$, has constant elements, not depending on the input variables of the problem, and can be assumed as a given vector. Therefore stage 2 requires only a discrete Fourier transform of a real vector. In this case the algorithm has a slightly lower computational cost.

COROLLARY 3.1. *The matrix* $A_\varepsilon \in \mathcal{T}_\varepsilon$, $\varepsilon \neq 0$, $\det A_\varepsilon \neq 0$, *can be inverted in* $6 \log n + 3$ steps *with* $2n$ *processors.*

Proposition 3.1 yields an interesting result in which the class of approximate algorithms, introduced in [5] for sequential computations, can also be used successfully in the parallel computations. We have in fact the following:

PROPOSITION 3.2. *The solution of the linear system* $Ax = b$, $A \in \mathcal{T}_0$, $\det A \neq 0$, *can be approximated with any precision by using* $6 \log n + 6$ *steps and* $2n$ *processors. The inverse of* A *can be approximated with any precision by* $6 \log n + 3$ *steps and* $2n$ *processors.*

Proof. Since the determinant is a continuous function of the elements of the matrix, and since $\det A \neq 0$, there exists a real positive number ε_0 such that if $|\varepsilon| < \varepsilon_0$ then $\det A_\varepsilon \neq 0$, where $A_\varepsilon \in \mathcal{T}_\varepsilon$ is the matrix whose first row is the first row of A. Since $\lim_{\varepsilon \to 0} A_\varepsilon^{-1} = A^{-1}$, the proposition holds in the light of Proposition 3.1 and Corollary 3.1.

The approximate computation described in the above proposition can be turned into an exact one by means of the interpolation technique used in [2] in the sequential case. For parallel algorithms, when switching from the approximate to the exact computation, we must increase the number of processors only, leaving the number of steps constant; in the sequential case, however, this transformation brings about a slight increase in the number of operations.

Let $A_\varepsilon \in \mathcal{T}_\varepsilon$ be such that $A_\varepsilon = (a_{i,j})$, $a_{i,j} = 0$ if $j - i > k$, $\det A_\varepsilon \neq 0$. Since the inverse matrix of A_ε is given by $A_\varepsilon^{-1} = (\mathrm{adj}\, A_\varepsilon)/\det A_\varepsilon$, the elements of A_ε^{-1} are rational functions of ε; that is, $p_{i,j}(\varepsilon)/q(\varepsilon)$, where $q(\varepsilon), p_{i,j}(\varepsilon)$ are polynomials of degree at most k. Furthermore the solution x_ε of the system $A_\varepsilon x = b$, given by $x_\varepsilon = A_\varepsilon^{-1} b$, has components which are rational functions of ε; namely, $r_{i,j}(\varepsilon)/q(\varepsilon)$, where $r_{i,j}(\varepsilon)$ has degree at most k.

Now let us choose $\varepsilon_0, \varepsilon_1, \cdots, \varepsilon_k$, such that $\varepsilon_i \neq \varepsilon_j$ if $i \neq j$ and $q(\varepsilon_i) \neq 0$, $i = 0, 1, \cdots, k$, in such a way that the $(k+1) \times (k+1)$ Vandermonde matrix $V = (v_{i,j})$, $v_{i,j} = \varepsilon_{i-1}^{j-1}$, be nonsingular and the system

$$V^T (\mathrm{Diag}\,(q(\varepsilon_0), q(\varepsilon_1), \cdots, q(\varepsilon_k)))^{-1} c = e^{(1)}, \tag{3.1}$$

has a solution $c = (c_i)$; here $e^{(1)}$ is the first column of the identity matrix of order $k+1$.

From (3.1) we have that, for any polynomial p of degree less than $k+1$, the following relation holds

$$\sum_{i=0}^{k} c_{i+1}\frac{p(\varepsilon_i)}{q(\varepsilon_i)}=p(0),$$

and in particular, if $\det A_0\neq 0$

$$A_0^{-1}=\left(\sum_{i=0}^{k} c_{i+1}A_{\varepsilon_i}^{-1}\right)/\det A_0,$$

$$x=\left(\sum_{i=0}^{k} c_{i+1}x_{\varepsilon_i}\right)/\det A_0, \tag{3.2}$$

where $x=A_0^{-1}b$.

Relations (3.2) allow us to solve the system $A_0x=b$ or to compute exactly the inverse matrix A_0^{-1} in a small number of parallel steps. We have in fact the following

PROPOSITION 3.3. *The solution of the linear system* $Ax=b$, $A\in\mathcal{T}_0$, $A=(a_{i,j})$, $a_{i,j}=0$ *if* $j-i>k$, $\det A\neq 0$, *can be computed exactly in* $6\log n+\log(k+1)+7$ *parallel steps with* $\frac{5}{2}n(k+1)$ *processors.*

Proof. Let a_1 be the diagonal element of A. Then, from (3.2) we have $x=(\sum_{i=0}^{k} c_{i+1}x_{\varepsilon_i})/a^n$. Therefore we can proceed in the following stages:

1) compute $\hat{q}(\varepsilon_i)=\det A_{\varepsilon_i}/\det A$ and solve $A_{\varepsilon_i}x_{\varepsilon_i}=b$, $i=0,1,\cdots,k$,
2) compute $\hat{c}=c/\det(A)$, where c is the solution of (3.1),
3) compute $\sum_{i=0}^{k}\hat{c}_{i+1}x_{\varepsilon_i}/\det A$.

Now, from Proposition 2.1 the determinant of $A_\varepsilon\in\mathcal{T}_\varepsilon$ is given by $\prod_{j=1}^{n} d_j$; therefore, from Proposition 3.1, stage 1 needs $6\log n+6$ steps and $\frac{5}{2}n(k+1)$ processors. This can be accomplished by following the 5 stages of the algorithm shown in Proposition 3.1, the difference being that at stage 2 of this algorithm we also compute $\det A=a_1^n$, at stage 4 we also have to compute $\prod_{j=1}^{n} d_j$; this requires $n/2$ more processors for each i, which leads to $\frac{5}{2}n(k+1)$ processors; at stage 5 we also compute $\hat{q}(\varepsilon_i)$. The computational cost of stage 2 does not depend on n, but would be given by $O(\log^2 k)$ steps and $O(k^4)$ processors if we solve it every time it is required.

Since the matrix V, as well as the vector $e^{(1)}$, do not depend either on the elements of A or on the elements of b, we can rephrase (3.1) in the following way: $(c_{i+1}/\det(A))=(q(\varepsilon_i)/\det(A))v_{i+1}$, where $v=(v_i)$ is the first row of V^{-1} and can be assumed as a given vector. Therefore stage 2 costs only 1 step and n processors. Stage 3 needs $1+\log(k+1)$ steps and $n(k+1)$ processors. The overall cost, therefore, is given by $6\log n+\log(k+1)+7$ steps and $\frac{5}{2}n(k+1)$ processors.

Observe that $6\log n+\log(k+1)+4$ steps are sufficient for inverting A. From Proposition 3.3 we have that the parallel complexity of the inversion of a Toeplitz triangular matrix is reduced from $O(\log^2 n)$ to $O(\log n)$, leaving unchanged the order of the number of processors. In fact the best algorithm to date required $\log^2 n+2\log n-1$ steps and $n^2/4$ processors, whereas this algorithm requires $7\log n+4$ steps and $\frac{5}{2}n^2$ processors. Table 1 shows the number of steps of these two algorithms.

Note that the threshold value n_0, for which if $n>n_0$ the new algorithm is faster than the known one, is quite low, i.e. $n_0=32$.

It is worth pointing out that, if the matrix A_ε belongs to the algebra $\mathcal{T}_\varepsilon(\mathbf{C})$ generated by H_ε over $\mathbf{C}$, the system $A_\varepsilon x=b$ can be solved by using $6\log n+3$ parallel steps with $4n$ processors. In fact the vectors $u^{(1)}$ and $v^{(1)}$, at stage 1 of Proposition 3.1, are complex vectors, therefore $3\log n$ steps and $4n$ processors are needed for stage 2. Analogously it is possible to extend Corollary 3.1 ($6\log n+3$ steps, $2n$

TABLE 1

n	$\log^2 n+2\log n-1$	$7\log n+4$
17–32	34	39
33–64	47	46
65–128	62	53
129–256	79	60
257–1,024	98	67
1,025–2,048	119	74
2,049–4,096	142	81

processors), Proposition 3.2 ($6\log n+3$ steps, $4n$ processors) and Proposition 3.3 ($6\log n+\log(k+1)+4$ steps, $4n(k+3/2)$ processors) to the complex case. Moreover choosing $\varepsilon_i=\omega_{k+1}^i$ in Proposition 3.3, where ω_{k+1} is a primitive $(k+1)$st root of unity, implies $c=d$ and reduces the number of steps to $6\log n+\log(k+1)+3$.

Let us observe also that, if $\mathbf{F}$ is any field containing a primitive nth root of unity, then the system $A_\varepsilon x=b$, $A_\varepsilon\in\mathcal{T}_\varepsilon(\mathbf{F})$ can be solved by using $6\log n+3$ parallel steps. In fact Theorem 1, Proposition 2.1 as well as Proposition 2.2 still hold when the ground field is $\mathbf{F}$.

4. Further results. This section shows some applications of Proposition 3.2 of theoretical or computational interest. The first result is mainly theoretical and involves linear algebras generated by any $n\times n$ matrix A. It will be shown that a linear system with an associated matrix belonging to such a class can be solved in $O(\log n)$ parallel steps. The second result concerns the solution of systems with an unbalanced band Toeplitz matrix such as, for example, a Hessenberg matrix. In this case the complexity is given by $8\log n+O(1)$ with no restrictive conditions.

Let A be an $n\times n$ matrix and $\mathcal{A}$ the algebra generated by A over $\mathbf{C}$. Suppose that the distinct eigenvalues of A are $\lambda_1,\lambda_2,\cdots,\lambda_k$ and set $\rho_i=\operatorname{rank}(A-\lambda_i I)$ for the geometric multiplicity of λ_i. Consider the Jordan canonical form of A,

$$A=S\operatorname{Diag}(J_{1,1},J_{1,2},\cdots,J_{1,\rho_1},J_{2,1},J_{2,2},\cdots,J_{2,\rho_2},\cdots,J_{k,1},J_{k,2},\cdots,J_{k,\rho_k})S^{-1};$$

where $J_{i,j}$, of dimension $n_{i,j}\times n_{i,j}$ is a Jordan block, i.e.

$$J_{i,j}=\begin{bmatrix}\lambda_i & 1 & & & \\ & \lambda_i & 1 & & \\ & & \ddots & \ddots & \\ & & & \ddots & 1\\ & & & & \lambda_i\end{bmatrix};$$

and $n_{i,1}\geqq n_{i,j}$, and $\hat{n}=\max n_{i,1}$.

If $B\in\mathcal{A}$ then we can represent B in the following way:

(4.1)
$$B=S\operatorname{diag}(T_{1,1},T_{1,2},\cdots,T_{1,\rho_1},T_{2,1},T_{2,2},\cdots,T_{2,\rho_2},\cdots,T_{k,1},T_{k,2},\cdots,T_{k,\rho_k})S^{-1},$$

where $T_{i,j}$ is an upper triangular Toeplitz matrix such that $T_{i,j}$ is a principal submatrix of $T_{i,1}$, $j=2,3,\cdots,\rho_i$, $i=1,2,\cdots,k$. Therefore the linear system $Bx=b$ is equivalent to

(4.2) $$Ty = c, \quad c = S^{-1}b, \quad y = S^{-1}x,$$

where T is the block diagonal matrix in (4.1).

The parallel solution of (4.2) can be accomplished in the following stages:

1) compute $c = S^{-1}b$,
2) solve $Ty = c$,
3) compute $x = Sy$.

Stages 1 and 3 require $\log n$ steps and n^2 processors, unless the matrix S has some specific structure. In stage 2 we have to solve $\rho_1+\rho_2+\cdots+\rho_k$ linear systems whose associated matrix is a triangular Toeplitz matrix. From Proposition 3.3, $7 \log \hat{n} + 7$ parallel steps and $\frac{5}{2}\sum_{i,j} n_{i,j}^2$ processors would be needed if the blocks $T_{i,j}$ had real elements. Since, in general, $T_{i,j}$ has elements belonging to the complex field, we obtain $7\log \hat{n}+4$ steps and $4\sum_{i,j} n_{i,j}(n_{i,j}+\frac{1}{2})$ processors. We may conclude with the following:

PROPOSITION 4.1. *The solution of the system $Bx = b$, where $B \in \mathcal{A}$, can be computed in $2 \log n + 7 \log \hat{n} + 4$ parallel steps by using $\max(n^2, 4\sum_{i,j} n_{i,j}(n_{i,j}+\frac{1}{2}))$ processors.*

A second result of computational interest can be obtained from Proposition 3.3.

Consider the system

(4.3) $$Ax = b,$$

in which the Toeplitz $n \times n$ matrix A is such that $a_{i,j} = 0$ if $i-j>k$, or if $j-i>h$. Moreover, suppose that $k \leqq h$ and $a_{k,1} = a_1 \neq 0$. Let B be the $(n+k)\times(n+k)$ upper triangular Toeplitz matrix whose A is a submatrix. The following shows the relation between A and B where $n = 4$, $k = 2$, $h = 3$.

$$B = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 \\ & a_1 & a_2 & a_3 & a_4 & a_5 \\ & & a_1 & a_2 & a_3 & a_4 \\ & & 0 & a_1 & a_2 & a_3 \\ & & & & a_1 & a_2 \\ & & & & & a_1 \end{bmatrix}.$$

Note that x is a solution of (4.3) if, and only if, there exists a k-vector y such that the system

(4.4) $$B\begin{bmatrix} z \\ w \end{bmatrix} = \begin{bmatrix} b \\ y \end{bmatrix},$$

has a solution in which $z = 0$; in fact, in this case we have $w = x$.

Let us now solve (4.3) by solving (4.4). We must first clarify the relation between y and z.

Since $a_1 \neq 0$ we have $\det B \neq 0$ and

(4.5) $$\begin{bmatrix} z \\ w \end{bmatrix} = B^{-1}\begin{bmatrix} b \\ y \end{bmatrix},$$

where B^{-1} is still an upper triangular Toeplitz matrix ($\mathcal{T}_0$ is an algebra). Relation (4.5) can be rewritten as

(4.6) $$\begin{bmatrix} z \\ w \end{bmatrix} = \begin{bmatrix} S & R \\ U & T \end{bmatrix}\begin{bmatrix} b \\ y \end{bmatrix},$$

in which R, S, T and U are Toeplitz matrices and R is a square matrix of dimension k. It is easy to check that if $\det A \neq 0$ then $\det R \neq 0$; furthermore, from (4.6) we can clarify the relation between z and y; in fact, we have

$$z = Sb + Ry. \tag{4.7}$$

From (4.7) we have that the solution of (4.4) fulfills the condition $z = 0$ if and only if

$$R = -Sb. \tag{4.8}$$

Since $\det R \neq 0$ the solution of (4.8) exists.

We can compute the solution of (4.3) in the following stages:

	steps	processors
1) compute S and R by inverting R,	$6 \log(n+k) + \log(h+k) + 4$,	$\frac{5}{2}n(h+k)$,
2) compute $-Sb$	$1 + \log n$,	nk,
3) solve the system $Ry = -Sb$	$O(\log^2 k)$,	$O(k^4)$,
4) compute $B^{-1}\begin{bmatrix} b \\ y \end{bmatrix}$,	$6 \log(2n-1) + 4$	$4n - 2$.

Stage 4 is equivalent to multiplying an $n \times n$ triangular Toeplitz matrix by a vector. This is accomplished by bordering the matrix so that we obtain a circulant matrix of order $2n - 1$ and applying Theorem 1 together with Corollary 2.1. Note that stage 4 can be carried out by using $1 + \log n$ steps and $n(n+1)/2$ processors. The overall complexity is resumed by the following

PROPOSITION 4.2. *The solution of the system $Ax = b$, in which A is a $n \times n$ Toeplitz matrix such that $a_{i,j} = 0$ if $i - j > k$ or $j - i > h$ and $a_{k,1} \neq 0$, can be computed by using* $13 \log n + O(\log^2 k)$ steps *and* $\frac{5}{2}n(h+k)$ *processors, or by using* $8 \log n + O(\log^2 k)$ *steps and* $\max(n(n+1)/2, \frac{5}{2}n(h+k))$ *processors.*

We should point out that, in the case of a Hessenberg matrix, i.e., $h = n - 1$, $k = 1$, if $n \neq 2^m, 2^m - 1$ then the cost becomes $9 \log n + 7$ steps, $\frac{5}{2}n^2$ processors.

Furthermore, if $h = k$, in which case the band is balanced, we can make a comparison with the algorithms shown by Grcar and Sameh in [13]. These three algorithms function under certain restrictive conditions (mainly positive definiteness or nonsingularity of the leading principal submatrices). The computational cost of such algorithms is shown below, together with the cost of the algorithm shown above (new algorithm).

	steps	processors
Algorithm 1	$6 \log n + O(k \log k)$,	$4n$,
Algorithm 2	$3k \log n + O(\log^2 k \log n)$,	$4kn$,
Algorithm 3	$(10 + 6 \log k) \log n + O(k)$,	kn,
New algorithm	$13 \log n + O(\log^2 k)$,	$5kn$.

Note that the new algorithm functions without any restrictions, and, if $k \ll n$, the number of steps is quite unaffected by band-width, while the number of processors is slightly higher.

Remarks. The bordering technique used to reduce the solution of a banded Toeplitz system to a triangular Toeplitz system can also be used for certain banded matrices, even without a Toeplitz structure. In fact the five stages of the algorithm resumed in (4.9) still hold valid, the difference being that the inversion of B at stage 1 costs $O(\log(h+k) \log n)$ steps with $O((h+k)^2 n)$ processors [15], [8].

REFERENCES

[1] D. BINI, *Border rank of a $p \times q \times 2$ tensor and the optimal approximation of a pair of bilinear forms*, Lecture Notes in Computer Science, 85, Springer-Verlag, New York, 1980, pp. 98–108.

[2] ———, *Relations between exact and approximate bilinear algorithms, applications*, Calcolo, 17 (1980), pp. 87–97.

[3] D. BINI AND M. CAPOVANI, *Spectral and computational properties of band symmetric Toeplitz matrices*, T.R. B82-03, I.E.I. del C.N.R., Pisa, 1982.

[4] D. BINI, M. CAPOVANI, G. LOTTI AND F. ROMANI, *$O(n^{2.7799})$ for $n \times n$ approximate matrix multiplication*, Inform. Process. Lett., 8 (1979), pp. 234–235.

[5] D. BINI, G. LOTTI AND F. ROMANI, *Approximate solutions for the bilinear form computational problem*, this Journal, 9 (1979), pp. 692–697.

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

[7] S. CHEN, *Speedups of iterative programs in multiprocessing systems*, Ph.D. thesis, Dept. Computer Science, University of Illinois at Urbana-Champaign, 1975.

[8] S. CHEN AND D. KUCK, *Time and parallel processor bounds for linear recurrence systems*, IEEE Trans. Comput., C-24 (1975), pp. 701–717.

[9] D. COPPERSMITH AND S. WINOGRAD, *On the asymptotic complexity of matrix multiplication*, Tech. Rep., IBM T. J. Watson Research Center, Yorktown Heights, NY, 1981; this Journal, 11 (1982), pp. 472–492.

[10] L. CSANKY, *Fast parallel matrix inversion algorithms*, this Journal, 5 (1976), pp. 618–623.

[11] ———, *On the parallel complexity of some computational problem*, Ph.D. thesis, Univ. California, Berkeley, 1974.

[12] P. DAVIS, *Circulant Matrices*, John Wiley, New York, 1979.

[13] J. GRCAR AND A. SAMEH, *On certain parallel linear systems solvers*, SIAM J. Sci. Stat. Comput., 2 (1981), pp. 238–256.

[14] D. HELLER, *On the efficient computation of recurrence relations*, T.R. ICASE NASA Langley Res. Centre, Hampton, VA, 1974.

[15] S. ORCUTT, *Parallel solution methods for triangular linear systems of equations*, Tech. Rep. 77, Stanford Electronics Labs., Stanford, CA, 1974.

[16] V. YA. PAN, *Field extension and trilinear aggregating, uniting and cancelling for the acceleration of matrix multiplication*, Proc. 20th ACM Symposium on the Foundations of Computer Science, 1979, pp. 28–38.

[17] M. PEASE, *An adaptation of the fast Fourier transform for parallel processing*, J. Assoc. Comput. Mach., 15 (1968), pp. 252–264.

[18] A. SCHÖNHAGE, *Partial and total matrix multiplications*, this Journal, 10 (1981), pp. 433–455.

SIAM J. COMPUT.
Vol. 13, No. 2, May 1984

© 1984 Society for Industrial and Applied Mathematics
005

SELF-ORGANIZING HEURISTICS FOR IMPLICIT DATA STRUCTURES*

GREG N. FREDERICKSON†

Abstract. Self-organizing heuristics are presented for data structures containing elements with different weights (access probabilities). The structures use just a constant number of locations in addition to those required for the values of the elements, and support average access times that are within a constant multiplicative factor of optimal. Data structures and corresponding heuristics with essentially optimal average unsuccessful search times are also given for the case in which there are probabilities of access associated with the intervals between consecutive element values.

Key words. binary search tree, implicit data structure, probabilities of access, searching, self-organizing heuristics, unsuccessful search

1. Introduction. Additional space, beyond the minimum required to store key values, appears useful in the implementation of various data structures. Consider a data structure to hold a static set of n elements that are to be searched. The elements will be accessed independently according to weights (access probabilities) $p_1, \cdots, p_n$, and the $n+1$ intervening intervals will be accessed according to weights $q_0, \cdots, q_n$. If $2n+1$ pointers are available, then the elements can be stored in an optimal binary search tree [K1], [K2] or nearly optimal binary search tree [Ba], [Fm], [GW], [HT], [Me1], [Me2]. But without that much additional space, how well can one do? In this paper, we continue an examination of this question initiated in [Fs3]. Our answers yield insight into such issues as the difficulty of handling unsuccessful search as opposed to successful search and the amenability of certain data structures to self-organizing techniques.

We consider primarily data structures that have only a constant number of additional locations. We shall assume that the values are stored in the first n locations of a one-dimensional array. Such data structures have been termed *implicit data structures* [MS], [Fs1]. Suppose that we know the probabilities of access a priori. In an earlier paper [Fs3], we presented implicit data structures and associated search strategies that were considerable improvements over the obvious choice of ordering values by decreasing weight and using sequential search [K2].

Suppose that the elements will be accessed according to fixed probabilities, but these probabilities will not be known in advance. Thus, any structure that will have search times sensitive to the probabilities of access must be *self-organizing*. That is, the structure should arrange itself according to some function of its access history. The most accurate method is to maintain a count of the number of times each element is accessed, and arrange the elements based on these counts. Unfortunately, this requires substantial additional space.

Self-organizing heuristics for sequential search have been studied in [Bi], [GMS], [H], [K2], [Me], [R]. The best of these heuristics realize $\Theta(\min\{1/p_i, n\})$ time on average for a successful search. In this paper, we consider using self-organizing heuristics to approximate one of the implicit structures in [Fs3]. We show that two self-organizing heuristics may be used to approximate this structure and realize average successful search times of $O(\log \min\{1/p_i, n\})$.[1]

* Received by the editors June 23, 1982, and in revised form March 30, 1983. This research was supported in part by the National Science Foundation under grants MCS-7909259 and MCS-8201083.

† Department of Computer Science, The Pennsylvania State University, University Park, Pennsylvania 16802. Current address: Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907.

[1] All logarithms are to the base 2. When $O(\log x)$ is used, we shall mean $O(\max\{1, \log x\})$.

We also settle a question left open in [Fs2]. Specifically, we construct an implicit structure that is self-organizing and also handles unsuccessful search well. Since this structure and its self-organizing heuristic are somewhat complicated, we also present several structures in which a limited amount of additional space is allowed. Self-organizing heuristics for binary search trees, in which unsuccessful search can be handled, have been presented in [AM], [Bi]. We present a compressed version of a binary search tree, using $n+1$ pointers, and another structure that requires $O(n^{1/2})$ pointers. Finally, we give our fully implicit self-organizing structure, with average unsuccessful search times of $O(\log \min \{1/q_i, n\})$.

A preliminary version of this paper appeared as a portion of [Fs2].

2. Self-organizing heuristics for successful search. We first review an implicit structure from [Fs3] that realizes fast access times for successful search, given that the probabilities of access are known. The elements are partitioned into groups A_i, $i=0, 1, \cdots, s$, on the basis of their weights. Let group A_i, $i<s$, be of size $h(i)$, and group A_s be of size at most $h(s)$, where $h(i)=2^{2^i}-1$. The elements are partitioned among groups so that the weight of any element in group A_i is no smaller than the weight of any element in group A_{i+1}, for $i=0, 1, \cdots, s-1$. The elements are then arranged in sequential memory by increasing group number, and within group by increasing value. We shall call this structure I. Note that the failure probabilities have not been used in building this structure.

To search for an element in structure I, examine each group in order, starting with group A_0, until the element is found or the groups are exhausted. Within each group a binary search is performed. Individual values of $h(i)$ need not be stored, since they may be computed as needed during the search. By the choice of $h(\cdot)$, the search time in the ith group will be approximately equal to the sum of the search times in the previous groups.

Since we do not know the probabilities of access, we consider implicit structure $\hat{I}$, which will be an approximation to our structure I. As in structure I, the elements will be partitioned into groups, with values in each group arranged in increasing order. Our heuristics will rearrange elements to have the expected position of an element be close to its optimal position. We shall present two heuristics, and show that each performs quite well.

The first heuristic is an adaptation of the move-to-the-front heuristic [Mc] used in conjunction with sequential search. The *move-to-the-first-group* heuristic is as follows. When an element that is not in the first group is referenced, it is moved up to the first group. An element from each intervening group is chosen at random and moved down one group. The second heuristic is an adaptation of the transposition heuristic [Mc] also used in conjunction with sequential search. The *move-up-one-group* heuristic is as follows. When an element that is not in the first group is referenced, it is moved up one group, and an element at random in the group is moved down one group.

The effect of the heuristics is illustrated in Fig. 1. Suppose the current state of the structure is illustrated in Fig. 1a, and the value 13 is accessed. Figure 1b shows the result of a move-to-the-first-group reorganization, where the values 18 and 35 have been chosen at random to be moved down. Figure 1c shows the result if a move-up-one-group reorganization is used instead. To help in discussing the analysis, we make the following distinctions. The word *search* will indicate only the activity of looking for a desired element. The word *access* will indicate both searching for an element and performing the portion of a self-organizing move that immediately follows the search.

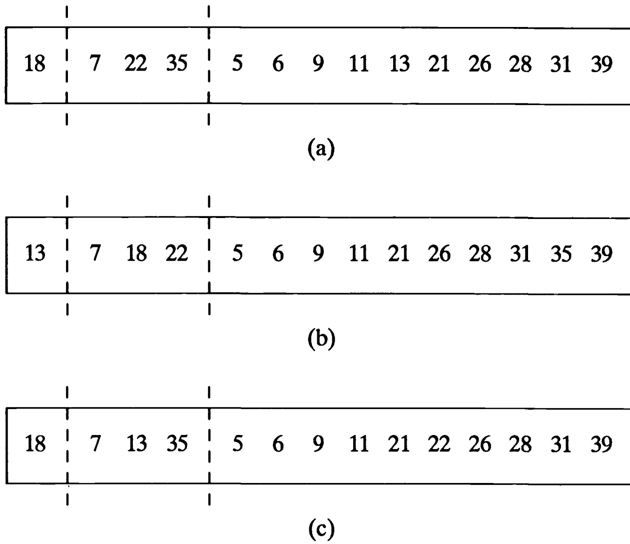


FIG. 1. *Implicit structure* $\hat{I}$: a) *before a self-organizing move,* b) *after a move-to-the-first-group rearrangement,* c) *after a move-up-one-group rearrangement.*

We shall show that, under an appropriate implementation, both heuristics realize expected access times that are $O(\log \min \{1/p_j, n\})$ for a successful search.

The data rearrangement required by these heuristics can be considerable. In worst case $\Omega(n)$ elements would have to be moved in a self-organizing move. If a self-organizing move were performed after every search, the access times would be degraded significantly, since search times are only $O(\log n)$ in worst case. Instead, a self-organizing move should be initiated only every so often, with the work distributed over succeeding accesses. A self-organizing move will be initiated every n time steps, where a time step will correspond to an element comparison in a search. A self-organizing move may always be completed during this interval.

Since a self-organizing move cannot in general be completed in one turn, the structure must be left in a state that is as close as possible to that already described. All but two groups should be in completely sorted order, and these two groups should be completely sorted except for one element. We discuss the implementation of the move-to-the-first-group heuristic in greater detail below.

The following procedure ensures that the conditions specified above will hold. Let x be the value that is being moved forward. While x is not in group A_0, do the following. Choose an element z at random from the next group up. Swap x and z. Now restore the order of the group that has received z, by a sequence of swaps. When x is in group A_0, restore the order of this group by a sequence of swaps. An example of a self-organizing move that is under way is shown in Fig. 2, which is intermediate

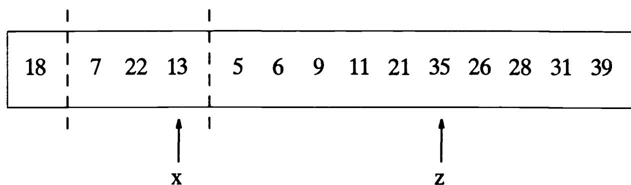


FIG. 2. *Implicit structure* $\hat{I}$ *with a move-to-the-first-group rearrangement under way.*

between Figs. 1a and 1b. Values $x=13$ and $z=35$ have been swapped, and 35 is currently in the process of being swapped into the correct position in group A_2. The search may still be performed efficiently in the partially reorganized structure.

It is possible to speed up convergence by storing the elements of each group in a recursively rotated list structure of [Fs1]. The time for an exchange in such a structure is $\Theta(2^{\sqrt{2\log n}}(\log n)^{3/2})$, which is a considerable improvement over the $\Theta(n)$ work necessary in worst case to exchange elements in a sorted list. Since exchanges may be implemented using only a constant number of pointers (see [Fs1]), the current state of the reorganization can be described implicitly.

3. Analysis of self-organizing heuristics for successful search. In this section, we analyze the move-to-the-first-group and the move-up-one-group heuristics, under the second implementation. We shall perform the analysis in two stages. First, we establish an upper bound on the asymptotic probability that the jth element is in some group A_L with $L \geqq i$, given that a self-organizing move is performed after each access. From these probabilities, we determine the expected search time for element j, under the same assumption. We then shift to the assumption that a self-organizing move is performed after every nth comparison, and apply the previous results to show that the expected access times are $O(\log 1/p_j)$.

We first consider the performance of the move-to-the-first-group heuristic, under the assumption that a self-organizing move is performed after every search. Let P_{ji} be the asymptotic probability that the jth element is in some group A_L, for $L \geqq i$. To find an upper bound on P_{ji}, $i>0$, we model our structure by a two-state Markov chain, shown in Fig. 3. The first state will roughly approximate the condition that the jth element is in one of the groups $A_0, \cdots, A_{i-1}$, and the second state will roughly approximate the complementary condition. A transition from state 2 to state 1 will occur with probability p_j, and a transition from state 1 to state 2 will occur with

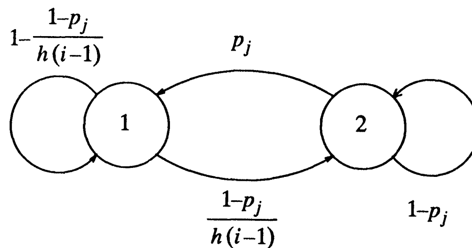


FIG. 3. *The Markov chain used to analyze the performance of the move-to-the-first-group heuristic.*

probability $(1-p_j)/h(i-1)$. Note that the latter probability overestimates the probability that an element will be transferred from some group A_L with $L<i$ to some group $A_{L'}$ with $L' \geqq i$, since it assumes that the element is in group A_{i-1} and that an access to an element in some group A_L, $L \geqq i$, has been made. Let M_{ji} be the asymptotic probability that the Markov process is in state 2. It can be seen from the previous discussion that $P_{ji} \leqq M_{ji}$. We use this bound to give the following result.

THEOREM 1. *Let the move-to-the-first-group heuristic be applied to implicit structure $\hat{I}$, with a self-organizing move performed after every search. Then,*

a. *For $i > \lceil \log\log 1/p_j \rceil$, the asymptotic probability that the jth element is in $\bigcup_{L=i}^{s} A_L$ is less than $(1/p_j)(1/2^{2^{i-1}})$.*

b. *The expected search time for the jth element is $O(\log 1/p_j)$.*

Proof. From elementary probability theory, we have that

$$M_{ji} = \frac{(1-p_j)/h(i-1)}{(1-p_j)/h(i-1)+p_j}.$$

Let r be the largest value of i such that M_{jr} is greater than $\frac{1}{2}$. It follows that $r = \lceil \log\log 1/p_j \rceil$. For $i > r$, we have $M_{ji} < 2^{2^r - 2^{i-1}}$. The first result then follows.

Let C_j be the expected cost to search for the jth element. From the given conditions, $r \geqq 0$. Thus

$$C_j \leqq \sum_{i=0}^{s} P_{ji} \log(h(i)+1) < \sum_{i=0}^{r} \log(h(i)+1) + \sum_{i=r+1}^{s} M_{ji} \log(h(i)+1)$$

$$< 2^{r+1} - 1 + \tfrac{1}{2}2^{r+1} + \sum_{i=r+2}^{\infty} 2^{2^r - 2^{i-1}+i}.$$

This expression assumes its largest value for $r = 0$. Thus

$$C_j < 10.13 \cdot 2^{r-1} < 10.13 \log 1/p_j.$$

The second result now follows. □

We now analyze the move-up-one-group heuristic, again under the assumption that a self-organizing move is performed after every search. Let P_{ji} be the asymptotic probability that the jth element is in group A_L, $L \geqq i$, under this heuristic. We again use a Markov chain, but this time one with $s+1$ states, $i = 0, 1, \cdots, s$, as shown in Fig. 4. Only the transitions between different states are shown. State i will roughly approximate the condition that element j is in group A_i of structure $\hat{I}$. A transition from state i to $i-1$, for $i > 0$, will occur with probability p_j. A transition from state si to $i+1$, for $i < s$, will occur with probability $(1-p_j)/h(i)$. Note that the latter probability overestimates the probability that an element will be moved down one group, since it assumes that an element in the next group down was accessed. Let t_{ji}

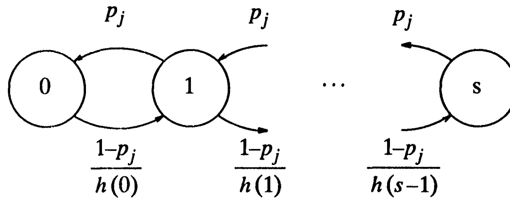


FIG. 4. *The Markov chain used to analyze the performance of the move-up-one-group heuristic.*

be the asymptotic probability that the Markov chain is in state i. Let T_{ji} be the asymptotic probability that the Markov chain is in some state $L \geqq i$. From the previous discussion, it can be seen that $P_{ji} \leqq T_{ji}$. We use this bound to derive the following result.

THEOREM 2. *Let the move-up-one-group heuristic be applied to implicit structure $\hat{I}$, with a self-organizing move performed after every search. Then*

a. *For $i > \lceil \log\log 1/p_j \rceil$, the asymptotic probability that the jth element is in $\bigcup_{L=i}^{s} A_L$ is less than $(1/p_j)(1/2^{2^{i-1}})$.*

b. *The expected search time for the jth element is $O(\log 1/p_j)$.*

Proof. From elementary probability theory, we have for $i = 1, 2, \cdots, s$

$$p_j t_{ji} = (1-p_j)/h(i-1)\, t_{j,i-1}.$$

Let r be the smallest integer such that $T_{j,r+1} \leqq \frac{1}{2}$. It follows that $p_j \geqq (1-p_j)/h(r)$ and $t_{jr} \geqq t_{j,r+1}$. Then, $(1-p_j)/h(r-1) > p_j \geqq (1-p_j)/h(r)$ and $2^{r-1} < \log 1/p_j \leqq 2^r$. Since $T_{j,r+1} \leqq \frac{1}{2}$, we get $t_{j,r+1} \leqq \frac{1}{2}$ and, for $i \geqq r+1$,

$$t_{ji} \leqq \tfrac{1}{2} \prod_{L=r+2}^{i} ((1-p_j)/p_j)/h(L-1) \leqq \tfrac{1}{2} \prod_{L=r+2}^{i} h(r)/h(L-1).$$

Thus $T_{jr} \leqq 1$, $T_{j,r+1} \leqq \frac{1}{2}$, and, for $i > r+1$,

$$T_{ji} = \sum_{L=i}^{s} t_{jL}.$$

Now, for $i > r+1$,

$$T_{ji} < \tfrac{1}{2} \sum_{L=i}^{s} \left(\frac{h(r)}{h(i-1)}\right)^{L-i+1} < \frac{(1/2)h(r)}{h(i-1)-h(r)} < 2^{2^r - 2^{i-1}}.$$

We note that the above bound also holds if $i = r+1$. The first result then follows.

Let C_j be the expected cost to access the jth element. The given conditions ensure that $r \geqq 0$. Thus,

$$C_j \leqq \sum_{i=0}^{s} P_{ji} \log(h(i)+1) < \sum_{i=0}^{r} \log(h(i)+1) + \sum_{i=r+1}^{s} T_{ji} \log(h(i)+1)$$

$$< 2^{r+1} - 1 + \tfrac{1}{2} 2^{r+1} + \sum_{i=r+2}^{\infty} T_{ji} 2^i.$$

This expression assumes its largest value for $r = 0$. Thus,

$$C_j < 5.52 \cdot 2^{r-1} < 5.52 \log 1/p_j.$$

The second result then follows. □

We now consider the performance of the heuristics under the second implementation, in which one self-organizing move is performed after every nth element comparison. We now wish to show that access times, as well as search times, are $O(\log 1/p_j)$. Let $H = \sum_{j=1}^{n} p_j \log 1/p_j$. From [Me1] we have that the expected search time in an optimal binary search tree is no smaller than $H/\log 3$. Since no arrangement of values in our search structure can have smaller expected search time than an optimal binary search tree, $H/\log 3$ is a lower bound for the expected search time in our structures also. Thus, in our structures, the expected number of searches that together use n comparisons is no greater than $n(\log 3)/H$, no matter which arrangement it is in.

If there were a fixed number of searches performed during each self-organizing move, then by Theorems 1 and 2, the expected search time for element j would be $O(\log 1/p_j)$. Under this condition, the expected search time overall would be less than cH, for some constant c. However, the number of searches per self-organizing move is not fixed: There are more searches on average for the arrangements with better average search time. Hence the average search time overall will be no longer if there are not a fixed number of searches per self-organizing move. Thus, the expected number of searches using a total of n comparisons is larger than $n/(cH)$. We may now prove that the self-organizing heuristics have the claimed performance.

THEOREM 3. *Let a self-organizing move be initiated after every nth element comparison in searches. Then the move-to-the-first-group and the move-up-one-group heuristics, when applied to structure* $\hat{I}$, *support expected access times of* $O(\log \min\{1/p_j, n\})$.

Proof. From the previous discussion, the expected number of searches for element j during a sequence of n element comparisons in steady state will be no smaller than $p_j \cdot n/(cH)$. Also, the expected number of searches for element j when element j is in $\bigcup_{L=i}^{s} A_L$, will be no greater than $p_j P_{ji} n(\log 3)/H$. Let r be the largest index such that at least half of the searches for element j will terminate in $\bigcup_{L=r}^{s} A_L$. Then

$$\frac{1}{2} p_j \frac{n}{cH} \leqq p_j P_{jr} \frac{n \log 3}{H}.$$

Thus

$$P_{jr} \geqq \frac{1}{2c \log 3}.$$

Let $w = \max\{r, \lceil \log\log 1/p_j \rceil\}$. From Theorems 1 and 2, we have

$$P_{jw} < \frac{1}{p_j} \frac{1}{2^{2^{w-1}}}.$$

Combining the two previous inequalities, we get

$$2^{w-1} < 1 + \log c + \log\log 3 + \log 1/p_j.$$

Thus the expected search time for element j will be no more than

$$C_j \leqq 2^{w+1} - 1 + \sum_{L=w+1}^{s} 2^L P_{jL}/P_{jr} < 2^{w+1} + 2c \log 3 \sum_{L=w+1}^{s} 2^L P_{jL} \leqq 2^{w+1} + c' \cdot 2^{w+1},$$

for some constant c'. Since searches are $O(\log n)$ in worst case, the claimed result then follows. □

4. Structures that use limited additional storage. We next consider self-organizing structures that handle unsuccessful search well. In § 5, we shall present a structure that accomplishes this and is also implicit. However, the structure is somewhat complicated. In this section, we consider simpler structures that use some pointers. We first consider binary search trees, since attractive self-organizing heuristics have been presented for them in [AM]. We propose a new, space-efficient representation that is amenable to these heuristics. A normal representation of a binary search tree would require $2n+1$ pointers: two pointers per node, plus a pointer to the root. This is quite inefficient, as there are $n+1$ null pointers. Our representation uses a total of just $n+1$ pointers.

Let the values from the tree be stored in ascending order in $v_1, v_2, \cdots, v_n$. In addition, there is a pointer array, $y_1, y_2, \cdots, y_{n-1}$, along with a pointer r to the root, and a pointer holding the value of n. For $j = 1, 2, \cdots, n-1$, y_j is a pointer to the right child of v_j in the tree, if there is one, and left child of v_{j+1}, otherwise. An example of a binary search tree is shown in Fig. 5a, and the corresponding space-efficient representation is shown in Fig. 5b, with the pointer and value-arrays interleaved.

Let the above representation of a binary search tree be called representation R.

LEMMA 1. *For every binary search tree, there is a valid representation R.*

Proof. The proof is a simple induction, using an inductive definition of a binary search tree. If the tree consists of a single node, then representation of R obviously exists. Otherwise, the tree consists of a root and either one or two nonempty subtrees. Assume that both subtrees are nonempty. By the induction hypothesis, there is a

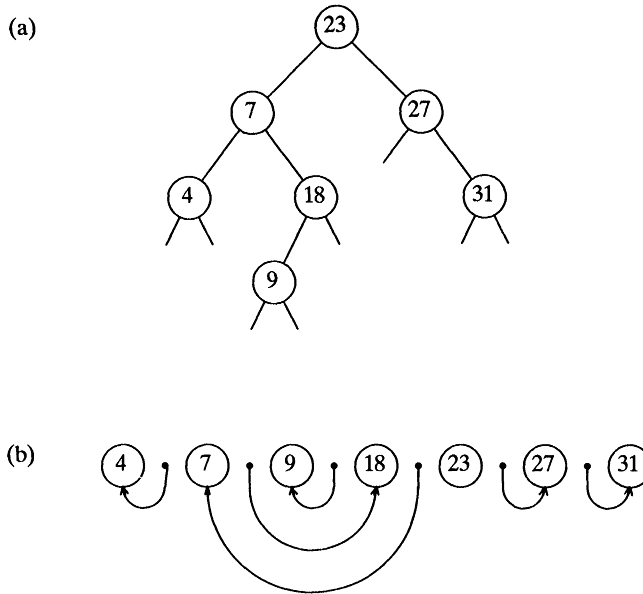


FIG. 5. *A binary search tree in*: a) *its normal representation,* b) *its representation R.*

representation for each nonempty subtree. Let these be $v_1(1: n_1)$, $y_1(1: n_1-1)$, r_1 and $v_2(1: n_2)$, $y_2(1: n_2-1)$, r_2, where $v_1(i) = v(i)$ and $v_2(i) = v(i+n_1+1)$. The full tree will have $y(i) = y_1(i)$ for $i = 1, \cdots, n_1-1$, $y(i+n_1+1) = y_2(i)$ for $i = 1, \cdots, n_2-1$, $y(n_1) = r_1$, $y(n_1+1) = r_2$ and $r = n_1+1$. The cases where either the left or the right subtree are empty can be handled similarly. □

A search in the structure proceeds in a straightforward fashion; when the left child of value v_j is desired, then a test $j = 0$ *or* $y_{j-1} \geqq j$ determines if there is a null left child. A similar test is performed if a right child is desired. The test for termination may be simplified to a single comparison if the value of n is in y_0, and the address of the root is in y_n. In this case, the left child of v_j will be null if and only if $y_{j-1} \geqq j$. A similar test is used for right children. Our representation is notable in that we make use of the numeric value of our pointers in our search. This contrasts with binary search trees, in which a weaker test of equality with null is made.

It is shown in [AM] that a move to the root heuristic, involving a sequence of simple rotations, yields expected access time $O(\log 1/p_j)$ and $O(\log 1/q_j)$. Our representation allows rotations to be performed quite easily. In fact, the heuristic may be applied from the top down, using only a constant number of extra pointers, in time proportional to the length of the search path. Insertions are, of course, not easy. Recently, Munro and Poblete [MP] have found another way to represent binary search trees with essentially n pointers, so that insertions are efficient. Unfortunately, for all of these representations, as is pointed out in [AM], worse-case search times for a self-organizing search tree are $\Omega(n)$. This may rule out some applications.

We next present a structure that is amenable to self-organization and also handles unsuccessful search well. The structure is *semi-implicit* in the sense of [MS], i.e., it uses $o(n)$ pointers. The particular instantiation, that we present, will use $O(n^{1/2})$ pointers, but it is possible to achieve fewer pointers at the expense of a larger constant factor in the expected access time.

The structure is organized in the following fashion. Element j and interval j are viewed as paired together. Let $p'_j = p_j + q_j$, $j = 1, \cdots, n-1$, and $p'_n = p_n + q_n + q_0$. Pair j is represented by element j, and the elements are partitioned into groups as the elements were in structure I, but on the basis of the new weights p'_j. The elements are ordered within a group by their values. The structure will consist of two arrays, $v_1, v_2, \cdots, v_n$ and $y_1, y_2, \cdots, y_m$, where v_j is the value of the element at position j, and, for $j \leqq m$, y_j is a pointer to the next larger value in the structure. For the largest value, the pointer would be to the smallest value. If we choose $m = \sum_{i=0}^{s-2} h(i)$, then the only values without corresponding pointers will be in the last two groups A_{s-1} and A_s. It may be seen that m is $O(n^{1/2})$. Let the above structure, with elements partitioned on the basis of p'_j as elements in structure I were, be called structure II.

A search for a value proceeds as before, with the groups examined in turn. When the search value x is determined not to be in a certain group, a check may be made, using a pointer to determine if x falls within an interval. The check will be made if $j \leqq m$, where v_j is the smallest value less than x in that group. The only searches that will suffer will be those for the intervals that would fall in group A_{s-1}. Since only one more group must be searched, the search time is at most doubled for an interval in group A_{s-1}.

Appropriate self-organizing heuristics are similar to those already presented. The data movement is the same, except that when an interval is accessed, the associated element is moved. The pointers must also be adjusted accordingly: Every time that a value is swapped into a new position, a search must be performed for the next smaller value, and its pointer must be updated. This will force an additional factor of $O(\log n)$ in the time to complete a self-organizing move. An example of our structure is shown in Fig. 6a. Suppose element 13 is accessed. Then, the resulting structure after a move-to-the-first-group self-organization is shown in Fig. 6b.

Let the approximation to structure II generated by using an appropriate self-organizing heuristic be denoted as structure $\hat{II}$.

THEOREM 4. *Structure* $\hat{II}$ *uses* $O(n^{1/2})$ *pointers and realizes* $O(\log 1/p_j)$ *and* $O(\log 1/q_j)$ *averages access times, respectively, for successful and unsuccessful search, under either of the indicated self-organizing heuristics.*

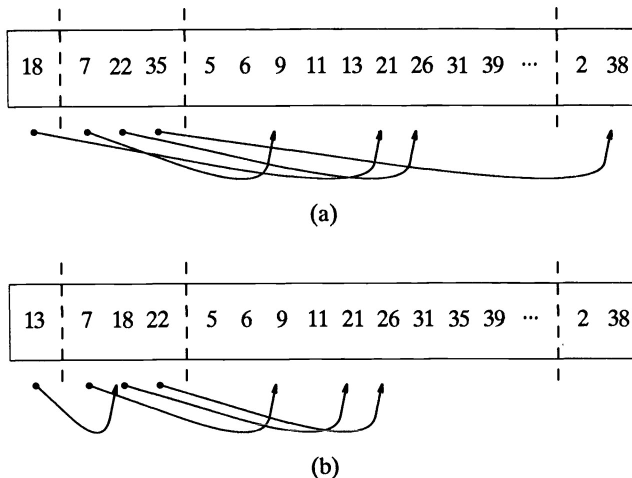


FIG. 6. *Semi-implicit structure* $\hat{II}$: a) *before and* b) *after a move-to-the-first-group rearrangement.*

Proof. By Theorem 3, the expected access time for an element with probability of access p'_j will be $O(\log 1/p'_j)$. Since $p_j \leqq p'_j$, the expected access time for element j will be $O(\log 1/p_j)$. Similarly, since $q_j \leqq p'_j$, and the expected access time for an interval j is at most twice that for an element j, the search time for interval j is $O(\log 1/q_j)$. □

As noted earlier, fewer pointers may be used. let $m = \sum_{i=0}^{s-c-1} h(i)$. Then, values in only groups $A_{s-c}, A_{s-c+1}, \cdots, A_s$ will not be paired with pointers. It may be seen that the average unsuccessful search time will be thus $O(2^c \log 1/q_j)$.

5. An implicit self-organizing structure for unsuccessful search. In this section, we present an implicit structure that is self-organizing and handles unsuccessful search well. Our structure *III* draws motivation from structure *II* in the previous section. Elements and intervals are once again viewed as paired. Thus, element j and interval j are viewed as paired together and have a combined probability of access of $p'_j = p_j + q_j$, for $j = 1, \cdots, n-1$, with $p'_n = p_n + q_n + q_0$. Elements are partitioned among groups in a manner similar to that described for structure *II*, except that the size of the groups will be described by the function $\tilde{h}(\cdot)$, to be defined later.

In structure *II* we used a pointer to identify to the next larger value in the structure. Since we can no longer use pointers, we must guarantee instead that the next larger value is nearby in the structure. Let $\pi(j)$ be the index of the next value larger than v_j in the set. (If v_j is largest, then $\pi(j)$ will index the smallest value.) Precisely stated, we require that if v_j is in group A_r, then $v_{\pi(j)}$ will be in one of the groups A_i, for $i \leqq r+1$. We term this the *patronage property*. If $v_{\pi(j)}$ is in group A_{r+1} and v_j is in group A_r, then v_j is termed the *patron* of $v_{\pi(j)}$, and $v_{\pi(j)}$ is termed the *protégé*. It does not appear to be convenient to be more selective about the location of $v_{\pi(j)}$. For instance, we cannot simply require $v_{\pi(j)}$ to be in group A_{r+1}, since $v_{\pi(j)}$ may have very large weight and thus deserve to be in some group A_i, with $i \leqq r$.

We shall describe two search procedures for structure *III*. The first will not be quite as simple, but its correctness will be obvious. The second will be quite simple, but its correctness will be more subtle. The first search is performed as follows. If the search value x is in A_0, then terminate the search with success; otherwise, continue to group A_1. To search group A_r, for $r > 0$, do the following. Let w be the largest value smaller than x in A_{r-1}. Search A_r for x. If found, terminate with success. Otherwise, let w' be the smallest value larger than w in $\bigcup_{i=0}^{r} A_i$. If $x < w'$, terminate the search with failure. Otherwise, continue to the next group. The correctness of this procedure is immediate, given that $v_j \in A_{r-1}$ implies that $v_{\pi(j)} \in \bigcup_{i=1}^{r} A_i$.

We simplify the search as follows. Instead of determining w', find w'', where w'' is the largest value smaller than x in A_r. Now terminate the search with failure if $w'' < w$. This termination condition is more convenient, since w'' can be identified as a byproduct in a search for x in A_r. We claim that this termination condition is equivalent to $x < w'$. First, we show that $x < w'$ implies $w'' < w$. Now, $x < w'$ and $w'' < x$ together imply $w'' < w'$. But, w' is chosen so that w and w' are consecutive elements in the set. Hence, $w'' < w$.

We next show that if $w'' < w$ is encountered in a search, then $x < w'$. We prove this by induction on the number of groups searched. As the basis, let w be in A_0, and let w'' be in A_1, with $w'' < w$. Then w', the next largest value greater than w, must be greater than x. For the induction, let $i > 0$ be the smallest value such that w is in A_i, and $w'' < w$ for w'' in A_{i+1}. Now, suppose $w' < x$. If w' is in one of $A_0, \cdots, A_{i-1}$, then the termination condition must have already been met in some earlier group and the search would not have progressed to group A_{i+1}. If w' is in A_i or A_{i+1}, then the choice of w or w'' was incorrect. Since none of these can happen, $w' > x$.

If probabilities of access are known, then structure *III* may be constructed in the following manner. Let the size of group A_i be $h(i)$. Group A_0 will consist of an element with largest weight p'_j. To construct A_i, given A_{i-1}, do the following. For every v_j in A_{i-1}, insert $v_{\pi(j)}$ in A_i if $v_{\pi(j)}$ is not in $\bigcup_{L=0}^{i-1} A_l$. Complete group A_i with the remaining elements of largest weights $\{p'_j\}$. It may be shown that the search time for an element or interval whose weight has rank k is $O(\log k)$. From the point of view of simplicity, this structure compares favorably with a static structure in section 4 of [Fs3]. However, the multiplicative factor is smaller for that structure.

In any event we are concerned here with self-organizing structures, so that we shall consider structure $\hat{III}$, which is an approximation to structure *III*. We consider the move-to-the-first-group and the move-up-one-group heuristics, modified to handle structure $\hat{III}$. The data movement in a self-organizing move will be somewhat more complicated for this structure. The element, that is the catalyst for the move, will be moved forward in the structure. In addition, it may be necessary to elevate a protégé into the next group down from its patron's group, in order to preserve the patronage property. But this protégé may have to bring along its own protégé, and so forth. (The situation may be roughly analogous to a promotion in an administrative hierarchy, where an appointee is allowed to bring along an associate to the next lower level.) Which elements will move up in the structure under the move-to-the-first-group heuristic may be determined by the following procedure. Start with j set to the index of the catalyst element, and i set to group index 0. While v_j is not in group A_i, v_j will be moved up, i will be incremented, and j will be set to $\pi(j)$. If i is initially set to the next group up, then this procedure may provide similar information for the move-up-one-group heuristic.

Of course, elements must be moved down to make space for the elements that are being moved up in the structure. Two restrictions appear necessary to make the self-organizing structure perform well. First, an element that is a protégé should not be demoted unless its patron is also demoted. If v_L is in A_r and $v_{\pi(L)}$ is in A_{r+1}, and $v_{\pi(L)}$ was demoted but not v_L, then the patronage property would be violated. Second, an element should not be moved down more than one group during all of the self-organizing move.

It is apparent that more than one element must be moved down from certain groups under the move-to-the-first-group heuristic. To determine how many elements must be moved down, do the following. If the catalyst is not in the first group, then one element will be moved down from the first group. If it has been determined that a elements in group A_{r-1} are to be moved down to group A_r, b elements are to be moved up out of group A_r, and c elements are to be moved up into group A_r, then $a-b+c$ elements should be moved from group A_r down to group A_{r+1}. It is simple to show that no more than $r+1$ elements from group A_r will be moved down.

The characterization of the self-organizing move given in the preceding paragraphs is sufficient to perform a self-organizing move. From § 2, we know that the self-organizing move cannot be performed all at once, but must be distributed over many subsequent accesses. This means that the self-organizing move must be performed in such a way that it can be interrupted at any point, and only a constant number of pointers will be needed to describe the state of the self-organizing move. Furthermore, the partially modified structure must be sufficiently well-formed so that searches may still be performed efficiently, given that the structure is in the middle of a self-organizing move.

The data movement for the move-up-one-group heuristic is simpler, and we consider it first. Let $x = v_j$ be the catalyst, with v_j in group A_L. While x is not in group A_{L-1}, do the following. Find the smallest r such that $v_{\pi^r(j)} \in A_{L+r}$. Thus, $y = v_{\pi^{r-1}(j)}$

will be the element to be promoted in A_{L+r-1}. Choose a nonprotégé element z at random from A_{L+r-2}. Swap y and z and restore the order in A_{L+r-1} by a sequence of swaps. When x is in A_{L-1}, restore the order of this group. An example of such a self-organizing move, with the inter-group swaps labeled in order of the execution, is shown in Fig. 7. The smallest value of r can be found by a fairly straightforward search. An element v_j can be tested for being a protégé by searching for the position of $v_{\pi^{-1}(j)}$.

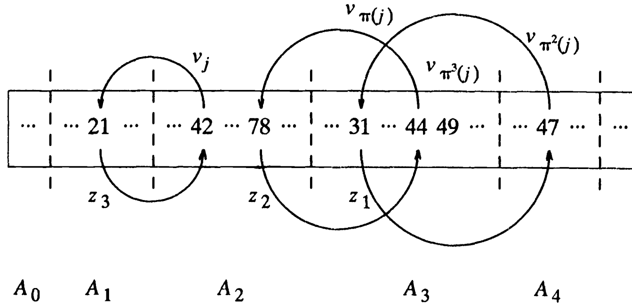


FIG. 7. *Inter-group data swaps for structure $\hat{III}$ under the move-up-one-group-heuristic.*

We consider searching the not completely well-formed structure while a self-organizing move is under way. Since a protégé is never moved down, at every point during the self-organizing move, the patronage property will hold for all elements in the structure. At any point, at most two elements (either y and z, or x and z) are out of order in the groups. As mentioned in § 2, one can search each group by first testing these elements, and then sidestepping them in the binary search.

We next present a procedure that will guarantee that the structure will be nearly well-formed during a move-to-the-first-group self-organizing move. Let $x = v_j$ be the catalyst. While x is not in group A_0, do the following. Find the smallest r such that $v_{\pi^r(j)} \in \bigcup_{L=0}^{r} A_L$. Let $y = v_{\pi^{r-1}(j)}$. While y is not in group A_{r-1}, do the following. Choose a bounceable nonprotégé element z at random from the previous group. Swap y and z, mark z as unbounceable, and restore the order in the group receiving z by a sequence of swaps. When y is in group A_{r-1}, restore the order of this group and mark y as unbounceable. When x is in group A_0, restore the order of this group. Now remove the unbounceable marks from all elements in the structure. An example of such a self-organizing move is shown in Fig. 8, again with the inter-group swaps labelled in order of occurrence.

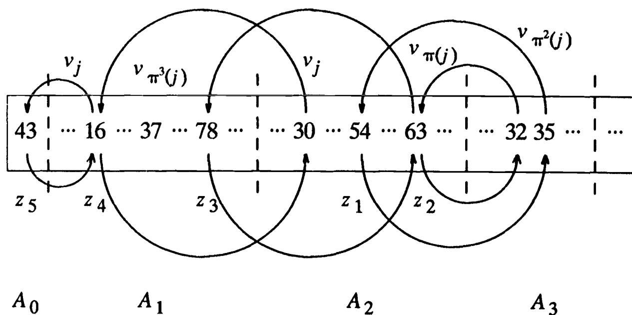


FIG. 8. *Inter-group data swaps for structure $\hat{III}$ under the move-to-the-first-group heuristic.*

It would be convenient to mark values by having an extra bit per value. Since this may not be possible (and we are interested in implicit structures anyway), the following trick from [Fs3] may be used. Each group may be viewed as partitioned into subgroups of size four. The four elements of a subgroup may be permuted to encode the value of the corresponding four-bit number. The search may handle these permuted subgroups by performing binary search on the first elements of subgroups and then performing a linear search through the two sub-groups identified by the binary search. The reader is referred to [Fs3] for a more complete description of this technique.

We now consider the choice of $\tilde{h}(\cdot)$ for structure $I\hat{I}I$. In the move-up-one-group heuristic, only elements that are not protégés may be chosen at random to be bounced down one group. Since every element in group A_{i-1} may have a protégé in group A_i, as few as $\tilde{h}(i)-\tilde{h}(i-1)$ elements in group A_i may be bounceable. To guarantee that there are $h(i)$ bounceable elements in group A_i, we choose $\tilde{h}(0)=h(0)$, and $\tilde{h}(i)=\tilde{h}(i-1)+h(i)$ for $i>0$.

For the move-to-the-first group heuristic, things are not so simple. In addition to the problem of protégés needing to be protected, in general, more than one element may be bounced out of a group. However, we know that at most $i+1$ elements will be bounced out of group A_i. Hence, to guarantee that the probability that a bounceable element j in group A_i is bounced is still no more than $(1-p_j)/h(i)$, we define $\tilde{h}'(0)=h(0)$ and $\tilde{h}'(i)=\tilde{h}'(i-1)+(i+1)h(i)$ for $i>0$. We now proceed to analyze the performance of these heuristics.

THEOREM 5. *Let the move-up-one-group heuristic be applied to implicit structure* $I\hat{I}I$, *with* $\tilde{h}(\cdot)$ *describing the size of the groups. If a self-organizing move is initiated after every* $n+cn^{1/2}\log n$ *element comparisons in searches, then the expected access times for successful and unsuccessful searches are* $O(\log\min\{1/p_j, n\})$ *and* $O(\log \min\{1/q_i, n\})$, respectively.

Proof. The performance of the heuristic may be modeled by the Markov chain in the proof of Theorem 2, with p_j replaced by p_j'. The probability p_j' that there is a transition from state i to state $i-1$ is an underestimate of the probability that an element is transferred from A_i to A_{i-1}, since an element may be promoted as a result of being a protégé of some element that is promoted. The probability of a transition from state $i-1$ to i remains the same, since it requires that some element other than itself be the catalyst, and that there are $h(i-1)$ elements in group A_{i-1} that can be bounced. Once again this probability is an overestimate, in part because element j may be a protégé and hence not bounceable.

A bound on P_{ji} similar to that in Theorem 2a may thus be established as

$$P_{ji}<\frac{1}{p_j'}\frac{1}{2^{2^{i-1}}}.$$

For the expected search time, given that a self-organizing move is performed after every search, we have

$$C_j \leqq \sum_{i=0}^{s} P_{ji}\log(\tilde{h}(i)+1).$$

From the definition of $\tilde{h}(\cdot)$, we have $\tilde{h}(i)\leqq\frac{4}{3}h(i)$, for $i\geqq 0$. Thus, the bound on the expected search time, given a self-organizing move after every search, will be $O(\log 1/p_j')$.

Now the number of swaps will be no greater than n. The work in testing for protégés is $O(n^{1/2}\log n)$, since there are $O(n^{1/2})$ protégés in groups A_i, for $i<s$.

Thus, a self-organizing move can be accomplished after the corresponding number of comparisons. Given these results, an analysis similar to that in Theorem 3 establishes that the expected successful search time is $O(\log 1/p'_j)$, which is $O(\log 1/p_j)$. Since the search time for interval j is of order the search time for element j, the expected unsuccessful search time is $O(\log 1/q_j)$. □

We now analyze the move-to-the-first-group heuristic.

THEOREM 6. *Let the move-to-the-first-group heuristic be applied to implicit structure* $\tilde{I}\hat{I}I$, *with* $\tilde{h}'(\cdot)$ *describing the size of the groups. If a self-organizing move is initiated after every* $c_1 n \log\log n + c_2 n^{1/2} \log n$ *element comparisons in searches, then the expected access times for the successful and unsuccessful searches are* $O(\log \min\{1/p_j, n\})$ *and* $O(\log \min\{1/q_j, n\})$, *respectively.*

Proof. The proof is similar to that of Theorem 5. The performance of this heuristic may be modeled by the Markov chain in the proof of Theorem 1, with p_j replaced p'_j. The probability p'_j that there is a transition from state 2 to state 1 is an underestimate of the probability that an element is promoted from $\bigcup_{L=i}^{s} A_L$ to $\bigcup_{L=0}^{i-1} A_L$, since an element may be promoted as a result of being a protégé. The probability of a transition from state 1 to state 2 is an overestimate of the probability that an element is bounced from $\bigcup_{L=0}^{i-1} A_L$ to $\bigcup_{L=i}^{s} A_L$, since at least $i \cdot h(i-1)$ elements in A_{i-1} are bounceable, and at most i elements will be bounced from A_{i-1}.

A bound on P_{ji} similar to that in the proof of Theorem 5 may be derived. For the expected search time and given that a self-organizing move is performed after every search, we have

$$C_j \leqq \sum_{i=0}^{s} P_{ji} \log(\tilde{h}'(i)+1).$$

From the definition of $\tilde{h}'(\cdot)$, we have $\tilde{h}'(i) \leqq (h(i))^2$. Thus, the expected search time, given that a self-organizing move is performed after every search, will be $O(\log 1/p'_j)$.

Since no more than s elements are swapped into any group, and $s < 1 + \log\log n$, the number of swaps is no greater than $n(1+\log\log n)$. The work in testing for protégés is similar to that in the proof of Theorem 5. Thus, a self-organizing move will be completed within the allotted amount of work. The remainder of the proof is similar to that of Theorem 5. □

6. Conclusion. We have identified implicit data structures and self-organizing heuristics that allow for fast access times in dictionaries of elements that have probabilities of access. The heuristics are natural generalizations of the move-to-the-front and transposition heuristics, and yield expected access times of $O(\log 1/p_j)$. We have also settled the question of whether additional space is needed to handle unsuccessful search well in a self-organizing environment. The data structure that we have designed seems particularly appropriate and potentially useful. The expected access times for unsuccessful search are $O(\log 1/q_j)$.

We have not pursued in this paper the problem of identifying a structure and self-organizing heuristic for which the average access times are a function of the rank of the weight of an element, rather than the weight itself. In [Fs2], we proposed a variant of the notion of rank, termed the *near-rank* $\tilde{k}$ of the weight p_j of an element, which is the number of elements of weight greater than $\frac{1}{2}p_j$. A data structure, along with a partial analysis, appears in [Fs2], and it appears that the analysis can probably be extended in a fashion similar to the analysis in this paper to yield average access times that are $O(\log \tilde{k})$.

There are a number of more interesting problems that we have not addressed here. We have not analyzed the rate of convergence of our structures to steady state. Certainly the implementation of the heuristics that require a self-organizing move after every nth element comparison would have reasonably slow convergence. It would be interesting to analyze expected access times and rate of convergence for the more natural implementation of the heuristics. Finally, we note that it would be quite interesting to obtain sharp performance bounds by which to compare the move-to-the-first-group and move-up-one-group heuristics. The multiplicative constants in the bounds derived in the proofs of Theorems 1 and 2, as well as intuition, might support the conjecture that move-up-one-group gives better average performance. However, the bounds derived in these arguments are quite loose, and care should be taken in drawing conclusions.

Acknowledgment. I would like to thank Mike Paterson and Leo Guibas for several helpful observations.

REFERENCES

[AM] B. Allen and I. Munro, *Self-organizing binary search trees*, J. Assoc. Comput. Mach., 25 (1978), pp. 526–535.

[Ba] P. Bayer, *Improved bounds on the costs of optimal and balanced binary search trees*, Dept. Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1978.

[Bi] J. R. Bitner, *Heuristics that dynamically organize data structures*, this Journal, 8 (1979), pp. 82–110

[Fs1] G. N. Frederickson, *Implicit data structures for the dictionary problem*, J. Assoc. Comput. Mach., 30 (1983), pp. 80–94.

[Fs2] ———, *Implicit data structures for the weighted dictionary problem*, Proc. 22nd Symposium on Foundations of Computer Science, Nashville, TN, 1981, pp. 133–139.

[Fs3] ———, *Implicit data structures for weighted elements*, CS-82-04, Dept. Computer Science, Pennsylvania State University, University Park, 1982.

[Fm] M. L. Fredman, *Two applications of a probabilistic search technique: sorting $X+Y$ and building balanced search trees*, Proc. 7th Symposium on Theory of Computing, 1975, pp. 240–244.

[GMS] G. H. Gonnet, J. I. Munro and H. Suwanda, *Exegesis of self-organizing linear search*, this Journal, 10 (1981), pp. 613–637.

[GW] C. C. Gotlieb and W. A. Walker, *A top-down algorithm for constructing nearly optimal lexicographic trees*, Graph Theory and Computing, Academic Press, New York, 1972, pp. 303–323.

[H] W. J. Hendricks, *An account of self-organizing systems*, this Journal, 5 (1976), pp. 715–723.

[HT] 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.

[K1] D. E. Knuth, *Optimum binary search trees*, Acta Informatica, 1 (1971), pp. 14–25.

[K2] ———, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[Mc] J. McCabe, *On serial files with relocatable records*, Oper. Res., 12 (1965), pp. 609–618.

[Me1] K. Mehlhorn, *Nearly optimal binary search trees*, Acta Informatica, 5 (1975), pp. 287–295.

[Me2] ———, *Best possible bounds on the weighted path length of optimum binary search trees*, this Journal, 6 (1977), pp. 235–239.

[MP] J. I. Munro and P. V. Poblete, *A discipline for robustness or storage redundancy in binary search trees*, Proc. ACM Symposium on Principles of Database Systems, Atlanta, March 1983.

[MS] J. I. Munro and H. Suwanda, *Implicit data structures for fast search and update*, J. Comp. Sys. Sci., 21 (1980), pp. 236–250.

[P] W. W. Peterson, *Addressing for random access storage*, IBM J. Res. Dev, 1 (1957), pp. 131–132.

[PR] Y. Perl and E. M. Reingold, *Understanding the complexity of interpolation search*, Inf. Proc. Lett., 6 (1977), pp. 219–222.

[R] R. L. Rivest, *On self-organizing sequential search heuristics*, Comm. ACM, 19 (1976), pp. 63–67.

SIAM J. COMPUT.
Vol. 13, No. 2, May 1984

© 1984 Society for Industrial and Applied Mathematics
006

VERIFICATION OF PROBABILISTIC PROGRAMS*

MICHA SHARIR†, AMIR PNUELI‡ AND SERGIU HART§

Abstract. A general method for proving properties of probabilistic programs is presented. This method generalizes the intermediate assertion method in that it extends a given assertion on the output distribution into an invariant assertion on all intermediate distributions, too. The proof method is shown to be sound and complete for programs which terminate with probability 1. A dual approach, based on the expected number of visits in each intermediate state, is also presented. All the methods are presented under the uniform framework which considers a probabilistic program as a discrete Markov process.

Key words. program verification, probabilistic programs, Markov chains

CR categories. 5.24, 5.5

Introduction. In this work we examine the possibility of developing verification methodology for probabilistic programs. The need for analysis of probabilistic programs arises in two main situations. The first is when we analyze a deterministic program whose inputs are drawn out of a space with some known probability distribution, and we wish to infer some statistical property of the program, such as its average running time, the expected value of some output variable, the probability of program termination, etc. Another situation is that of a nondeterministic program where the decision in nondeterministic forks in the program is made according to some known distribution. We could have, of course, a combination of the two where both the input values and nondeterministic choices within the program are chosen at random according to known distributions.

With the recent emergence of probabilistic algorithms, such as primality testing [RB] and synchronization between concurrent processes [LR], and the more conventional problem of average behavior of deterministic algorithms, the need for tools for probabilistic verification becomes increasingly urgent.

One possible approach to the probabilistic analysis of programs, which must certainly be the first step towards any coherent theory of the subject, is the definition of the probabilistic semantics of programs. Such an approach is taken for example in [KO] where a probabilistic program is regarded as a distribution transformer, transforming an input distribution into an output distribution. The output distribution then tells us the probability for the program to terminate in any of its terminal states. In principle, once we know how to compute the probability of each terminal state, we have captured the complete (input/output) behavior of the program, and each specific question can be settled by referring to the output distribution. In practice, however, when we are interested in a specific question, the computation of the complete distribution transformation is often a formidable and unnecessary task. This is why, in the nonprobabilistic case, the disciplines of semantic assignment and verification are closely related but still separate. The first seeks to define the mathematical interpretation of programs in a given language. The latter tries to offer methods by which specific questions about a program can be vigorously settled by extracting from

* Received by the editors October 28, 1981, and in revised form January 21, 1983.

† Department of Computer Science, Tel Aviv University, Tel Aviv, Israel. The research of this author was supported partly by the Office of Naval Research under grant N00014-75-C-0571, while the author was visiting Courant Institute, New York University, and partly by the Bat-Sheva Fund, Israel.

‡ Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel.

§ Department of Statistics, Tel Aviv University, Tel Aviv, Israel.

the program just the *minimal* amount of information which is required in order to settle the question. In one sense the theory of verification can be regarded as the theory of semantic approximation.

Taking as our starting point the probabilistic semantics of programs, as defined for example in [KO], we set out to see whether the verification methods that proved successful in the deterministic case, such as the intermediate assertion method [FL], computational induction [PA] and subgoal induction [MW], can be generalized to the probabilistic case.

The salient features of all these methods are:

a) They are *goal oriented*; i.e., the verification conditions to be solved depend on the property to be proved, and we only work so hard as is needed in order to establish the particular property.

b) The verification conditions are *local* in the sense that they connect two consecutive instants in the execution of the program.

c) If we insist on the *minimal* solution to the verification conditions we come up with the full semantics of the program, or an equivalent characterization. These are for example the minimal invariant predicates in Floyd's method.

As will be shown below, we suggest two generalizations. The first is an extension of the intermediate assertion method with some of the flavor of subgoal induction. Starting with an assertion on the terminal states which is supposed to hold upon termination, we seek to extend it into an assertion on all the states which holds continuously throughout the execution. The second method is similar to computational induction. We form equations which express changes in the distribution due to a single program step. The minimal solution to these equations gives exactly the terminal distribution. Consequently every solution, not necessarily the minimal, provides an upper bound to the terminal distribution.

We show that these two approaches are dual in the sense that they are both derived from the same matrix describing the program, and both obey certain duality relationships which allow us to combine the information yielded by each approach separately.

The methods are presented in a uniform framework which considers programs as global state transformations. It should not be too difficult to adapt them to more structured representations of programs. Specifically: The basic approach treats a probabilistic program as a Markov process that goes by a chain of transitions through the program states. At each step, depending on the current state, there are known probabilities for the next state, and the process chooses the next state according to their distribution.

These probabilities depend on the nature of the program statement about to be executed. If this statement is not a random draw, then there is a unique next state; otherwise, there may be several succeeding states, depending on the outcome of the draw made by the program. Thus, in such a model the state-transition probabilities can be assumed to be given a priority.

Note that random distribution of the program's input will manifest itself in similar distribution of the program states, but not in the transition probabilities among them.

Since, in principle, Markov chains model infinite processes, our approach can therefore also assign semantics to nonterminating probabilistic programs. This generalizes other approaches based on input–output semantics, which ignores nonterminating executions (see [KO] for example). To deal with terminating programs, we regard the terminal program states as absorbing states which once in them the process can never escape.

This paper is organized as follows. In § 1, we define probabilistic programs as Markov chains and assign to them semantics defined in terms of certain well-known quantities associated with such chains. We show that these semantics coincide with the semantics defined by Kozen [KO] and generalize it to nonstructured programs. In § 2, we describe our first method of probabilistic verification, which is based on invariant functionals on the program states' distribution. In § 3, we describe a second verification method based on the expected number of visits in the nonterminating states. Essentially this approach had been formerly suggested by Ramshaw [RA], but under a different framework. When cast into the framework of Markov chains, the approach becomes greatly simplified and much of the theory developed by Ramshaw turns out to be straightforward consequences of Markov chain theory. We also establish some "duality" relationships between our two approaches. In § 4, we demonstrate our verification methods in a series of examples.

When considering probabilistic programs as Markov chains, it is important to bear in mind that this representation is faithful only if all the data that can affect program execution is incorporated into the program's (or rather the chain's) states. Thus, if the program execution depends largely on its input, which in turn is drawn from some complex distribution, then incorporating the whole input data into the program states may make its analysis as a Markov chain rather difficult. (For example, if the program itself is fully deterministic, it may well be the case that its Markov chain representation decomposes into many disjoint chains, one per each input value.) The Markov chain representation is most favorable in cases where the probabilistic nature of the program arises from random draws made by the program itself.

Quite surprisingly, very few researchers have used the Markov chain model to represent probabilistic programs, although a completely static treatment of programs as Markov chains (with states being program locations only) has long been suggested by [RM]. Saheb-Djahromi [SD] uses Markov chains to define the operational semantics of a probabilistic version of the language LCF. The works of Kozen [KO] and Ramshaw [RA] mentioned above do not use Markov chains, although most of their results can be easily interpreted as standard results in Markov chain theory. Probabilistic analysis of programs is also studied by Wegbreit [WE].

Recently, the authors have generalized analysis of probabilistic programs to the case of concurrent programs [HSP], [HS]. Their program execution can be described as a certain cooperation of several Markov chains, and its analysis requires special techniques, unlike the classical Markov chain theory used in this paper. Another interesting and recent direction of research is the development of probabilistic logics for reasoning about properties of probabilistic programs (cf. [RE], [LS], [HS2]).

1. Probabilistic programs and their semantics. In our framework, a program is considered as a (probabilistic) transformation operating on a set of states. Let S denote the set of program states which may be infinite but countable. (See, however, § 4 for a treatment of uncountably many states.) We assume that the action of a single step of the program is represented by a given matrix of transition probabilities $P=\{P_{ij}\}$. Thus P_{ij} is the probability of going from state $i \in S$ to state $j \in S$ in one step. Let $\bar{\mu}^0$ be the initial distribution vector which specifies for each state $i \in S$ the probability $\mu_i^0 \geqq 0$ that initially the program is in this state. Based on the assumption that the probability for a transition from state i to state j depends only on i and j (and not on the time or any other nonlocal entity), an execution of a probabilistic program can be regarded as a Markov process which goes through a chain of discrete S states [CH], [RV], [KSK].

To illustrate these concepts, consider the following program:

$i := 0;$
l_1: **while** random $(p\delta_0 + q\delta_1) = 0$ **do** $i := i + 1;$
l_2: **halt**.

In general, the probabilistic expression "random (λ)" chooses a random value according to the distribution λ. In this case δ_i is a unit distribution concentrated at i, $i = 0, 1$. Thus, for $p + q = 1$, random$(p\delta_0 + q\delta_1)$ chooses 0 with probability p and 1 with probability $q = 1 - p$. Ignoring the initializing step, the set of states for this program is:

$$S = \{(l_1, i), (l_2, i) | i \geqq 0\}.$$

Note that states include the location in the program as well as values for all the program variables.

The initial distribution is given by:

$$\mu^0_{(l_1,i)} = \delta_{i,0}, \qquad \mu^0_{(l_2,i)} = 0.$$

That is: with certainty the initial state is $(l_1, 0)$. The transition probabilities are given by:

$$P_{(l_1,i)(l_2,i)} = q, \qquad P_{(l_1,i)(l_1,i+1)} = p, \qquad i \geqq 0.$$
$$P_{(l_2,i)(l_2,i)} = 1,$$

All other transitions have probability 0.

We partition our state space $S = I \cup T$. The set T is the set of all terminal states (absorbing states); for each $t \in T$, $P_{ts} = \delta_{t,s}$; i.e., with certainty we remain at t for the next stage and hence forever. The set $I = S - T$ is the set of intermediate states. Thus in the example above,

$$I = \{(l_1, i) | i \geqq 0\}, \qquad T = \{(l_2, i) | i \geqq 0\}.$$

Let us define:

$$P^{(n)}_{ij} = \{\text{Probability of reaching state } j \text{ from state } i \text{ in exactly } n \text{ steps}\}.$$

Obviously $P^{(n)}_{ij} = \{P^n\}_{i,j}$ where P^n is the nth power of the (infinite) transition probability matrix P. This can also be written as:

$$P^{(n)}_{ij} = \sum P_{ii_1} \cdot P_{i_1 i_2} \cdot \cdots \cdot P_{i_{n-1} j}$$

where the summation extends over all $(n-1)$-tuples $(i_1, \cdots, i_{n-1})$. Corresponding to an initial distribution $\bar{\mu}^0$, we can also define:

$$\mu^{(n)}_j = \{\text{Probability of being in state } j \text{ after } n \geqq 0 \text{ steps}\}.$$

Obviously $\mu^{(n)}_j = \sum_{i \in S} \mu^0_i P^{(n)}_{ij}$, or in matrix notation $\bar{\mu}^{(n)} = \bar{\mu}^0 P^n$. Note that, since each $j \in T$ is an absorbing state, the sequence $\{\mu^{(n)}_j\}_{n \geqq 0}$ is nondecreasing for each $j \in T$.

We also define:

$$f^{(n)}_{ij} = \{\text{Probability of reaching state } j \text{ from state } i \text{ \textit{for the first time} in exactly } n \text{ steps}\}.$$

These quantities satisfy:

$$f^{(n)}_{ij} = \sum P_{ii_1} \cdot \cdots \cdot P_{i_{n-1} j}$$

where the summation extends over all $(n-1)$-tuples $(i_1, \cdots, i_{n-1})$ of states all of which are different from j.

Clearly $f_{ij}^{(n)}$ or $\mu_j^{(n)}$, given an initial distribution, fully describe the behavior of the program. However, they are too detailed, and we would like to take out the dependence on the step counter n. One such integrated measure is given by:

$$f_{ij}^* = \sum_{n=1}^{\infty} f_{ij}^{(n)}$$

where f_{ij}^* is the probability of *ever* getting to state j from state i.

Similarly we define μ_j^* as the probability of ever getting to state j, given that the initial distribution is $\bar{\mu}^0$. Obviously:

$$\mu_j^* = \sum_{i \in S} \mu_i^0 f_{ij}^*. \tag{1}$$

If we restrict ourselves to terminal states $j \in T$, then since j is an absorbing state it follows that:

$$f_{ij}^* = \lim_{n\to\infty} P_{ij}^{(n)} \quad \text{and} \quad \mu_j^* = \lim_{n\to\infty} \mu_j^{(n)} \quad \text{for } j \in T.$$

The f_{ij}^* for $j \in T$ can be considered as the input–output semantics of the program viewed as a distribution transformer, in that given an initial distribution $\bar{\mu}^0$ the terminal distribution is given by $\bar{\mu}^* = \bar{\mu}^0 F^*$ where $F^* = \{f_{ij}^*\}$.

We will therefore regard the program as being fully specified when the matrix F^* is given. It should be noted that this is a generalization of Kozen's semantics, provided that one restricts oneself to discrete distributions. Note that Kozen defines the semantics of only a restricted class of structured programs, whereas our interpretation does not impose any such restriction. Let us indeed compare the two approaches for a while loop of the form

while $x \in B$ **do** Q.

For simplicity, let us identify the program states with values of x. The terminating states are then elements of B^c. Suppose that the subprogram Q has a transition probability matrix $\tilde{Q}$, where $\tilde{Q}$ specifies probabilities of transitions from B to $S = B \cup B^c$. Then the matrix associated with the whole program is easily seen to be

$$P = \left(\frac{\tilde{Q}}{0 \mid I}\right) = \begin{pmatrix} Q_1 & Q_2 \\ 0 & I \end{pmatrix}$$

where the states in B precede those in B^c, so that Q_1 is a transition (substochastic) matrix from B to B, and Q_2 is a similar matrix from B to B^c. A direct calculation shows that

$$P^n = \begin{pmatrix} Q_1^n & (Q_1^{n-1} + Q_1^{n-2} + \cdots + Q_1 + I)Q_2 \\ 0 & I \end{pmatrix}.$$

Hence, it follows from preceding remarks that if $i \in B$, $j \in B^c$ we have

$$f_{ij}^* = \sum_{k \geq 0} (Q_1^k Q_2)_{ij}.$$

But, in Kozen's notation, Q_1 is the matrix defining the linear operator $e_B \circ T_Q \circ e_B$ on the space of measures on S, whereas Q_2 is the matrix defining the operator $e_{B^c} \circ T_Q \circ e_B$, so that

$$f_{ij}^* = \left(\left[\left(\sum_{k \geqq 1} e_{B^c} \circ (T_Q \circ e_B)^k\right)(\delta_i)\right]\right.(\{j\}) = T(\delta_i)(\{j\})$$

where T is the distribution-transforming operator associated by Kozen with the while loop. This and (1) show that the two approaches indeed coincide for the above program.

Returning to the example program, it can be checked that

$$f^*_{(l_1,i),(l_2,j)} = p^{j-i}q \quad \text{if } j \geqq i,$$

and that

$$\mu_j^* = p^j q.$$

Unfortunately, in the general case, the quantities f_{ij}^* and μ_j^* may be difficult to compute explicitly, and we may be interested only in a partial property of the program. For example, we might only be interested in determining the expected number of steps till a 1 is chosen. This is the same as determining the expected value of i upon termination, which is

$$\sum_{i \geqq 0} i\mu^*_{(l_2,i)}.$$

We therefore would like to find methods for the calculation of such quantities without having to compute explicitly $\bar{\mu}^*$. This is done in the following sections.

2. Probabilistic verification by invariants. In this section we present our first probabilistic verification method. Motivated by the concluding remarks of the preceding section, we set out to find a way to compute a linear functional over $\bar{\mu}^*$, having the general form

$$\psi(\bar{\mu}^*) = \sum_{j \in T} \beta_j \mu_j^*.$$

This is a probabilistic analogue of an assertion on the terminal program states. We will assume, henceforth, that $\beta_j \geqq 0$, $j \in T$.

Our approach is to try to extend the coefficients $\{\beta_j\}_{j \in T}$ to a vector $\bar{\beta} = \{\beta_i\}_{i \in S}$ such that $\beta_i \geqq 0$, $i \in S$, and such that its restriction to T gives the original coefficients of ψ. Furthermore, we require $\bar{\beta}$ to be a right-characteristic vector of P, i.e., $P\bar{\beta} = \bar{\beta}$, or, in expanded form

$$\sum_{j \in S} P_{ij}\beta_j = \beta_i, \qquad i \in S.$$

(We assume also that all the infinite sums involved converge.) Such a $\bar{\beta}$ is known in Markov chain theory as a *P-regular* or *P-harmonic* function (cf. [RV], [KSK]). Note that, unless the vector $\{\beta_i\}_{i \in T}$ is bounded, some components β_i, for $i \in I$, may be $+\infty$.

If such a $\bar{\beta}$ exists, then define the linear functional $\varphi(\bar{\mu}) = \sum_{i \in S} \mu_i \beta_i$ which, given as an argument a distribution $\bar{\mu}$ over the program states, computes a real number (possibly ∞).

Then if $\varphi(\bar{\mu}^0)$ is finite, so is $\varphi(\bar{\mu}^{(n)})$, i.e. the value of the functional after step n, and we have the following invariance relation:

$$\varphi(\bar{\mu}^0) = \varphi(\bar{\mu}^{(n)}), \qquad n = 0, 1, \cdots.$$

This invariance is a consequence of the computation:

$$\varphi(\bar{\mu}^{(n)}) = (\bar{\mu}^{(n)} \cdot \bar{\beta}) = \bar{\mu}^0 P^n \bar{\beta} = (\bar{\mu}^0 \cdot \bar{\beta}) = \varphi(\bar{\mu}^0)$$

since $\bar{\beta}$ is a characteristic vector of P.

Obviously this gives a method for deriving invariance relations for general programs. We may now rewrite this invariance as:

$$\sum_{i \in I} \mu_i^{(n)} \beta_i + \sum_{i \in T} \mu_i^{(n)} \beta_i = \varphi(\bar{\mu}^0).$$

If we let now n go to ∞, the second term in the sum keeps increasing (because, for each $i \in T$, $\mu_i^{(n)}$ increases and $\beta_i \geqq 0$), and is bounded by $\varphi(\bar{\mu}^0)$ so that it must converge to a limit. Consequently, so must the first term, leading to:

$$R + \sum_{i \in T} \mu_i^* \beta_i = \varphi(\bar{\mu}^0)$$

where $R = \lim_{n \to \infty} R_n = \lim_{n \to \infty} \sum_{i \in I} \mu_i^{(n)} \beta_i \geqq 0$. Thus in the general case, we can conclude that

$$\psi(\bar{\mu}^*) \leqq \varphi(\bar{\mu}^0).$$

In the case where we can show that $R = 0$, we have an exact equality,

$$\psi(\bar{\mu}^*) = \varphi(\bar{\mu}^0).$$

Thus, this method allows us to compute the desired "output assertion" directly from the input distribution. Let us consider, for the simple example program above, the functional given by:

$$\beta_{(l_1, i)} = i + \frac{p}{q}, \qquad \beta_{(l_2, i)} = i$$

(i.e. a nonnegative functional that extends the desired functional ψ). The invariance verification condition $P\bar{\beta} = \bar{\beta}$ amounts in this case to

$$p\beta_{(l_1, i+1)} + q\beta_{(l_2, i)} = \beta_{(l_1, i)}, \quad \text{or} \quad p\left(i + 1 + \frac{p}{q}\right) + qi = i + \frac{p}{q},$$

which is easily verifiable, and

$$\beta_{(l_2, i)} = \beta_{(l_2, i)},$$

which is immediate.

Accepting the easily verifiable $\sum_i \mu_{(l_1, i)}^{(n)} \ (i + p/q) \to 0$ (since $\mu_{(l_1, i)}^{(n)} = \delta_{i,n} p^n$, $R_n = p^n(n + p/q) \to 0$), we therefore conclude:

$$\psi(\bar{\mu}^*) = \sum_{i=0}^{\infty} \mu_{(l_2, i)}^* i = \sum \mu_{(l_1, i)}^0 \left(i + \frac{p}{q}\right) = \frac{p}{q}.$$

This, of course, establishes that the expected value of i on termination of the program is p/q. (Here we have explicitly checked that $\lim_n R_n = 0$; in the sequel we will suggest several simple conditions that imply the vanishing of this limit.)

Summarizing the conditions for applicability of this method we have:

(V1) $$\sum_{i \in S} \mu_i^0 \beta_i < \infty,$$

(V2) $$\bar{\beta} \geqq 0, \qquad P\bar{\beta} = \bar{\beta}.$$

Then, under these two conditions we are assured of

$$\psi(\bar{\mu}^*) = \sum_{i \in T} \mu_i^* \beta_i \leqq \sum_{i \in S} \mu_i^0 \beta_i = \varphi(\bar{\mu}^0).$$

In fact, in order to ensure this result it is sufficient to have $P\bar{\beta} \leqq \bar{\beta}$ in (V2).

If in addition we also have

(V3) $$\sum_{i \in I} \mu_i^{(n)} \beta_i \underset{n \to \infty}{\to} 0 \qquad (\beta\text{-termination}),$$

then we may conclude that

(C) $$\psi(\bar{\mu}^*) = \sum_{i \in T} \mu_i^* \beta_i = \sum_{i \in S} \mu_i^0 \beta_i = \varphi(\bar{\mu}^0).$$

In order to emphasize the similarity between this method and the method of intermediate assertions [FL], we point out that (V1) is analogous to saying that φ is true initially, while (V2) is analogous to the local verification conditions. Thus (V1) and (V2) imply (V3)$\Rightarrow$(C), but this is the analogue of partial correctness. It states that if the program converges then the value on convergence is equal to $\varphi(\bar{\mu}^0)$. Only here, we have to require an appropriate rate of convergence as well (i.e. β-termination).

Note that if the β_i, $i \in T$, are uniformly bounded, then (V3) will hold if the program terminates with probability 1. Indeed, then one has

$$\lim_n \sum_{i \in I} \mu_i^{(n)} \beta_i \leqq \sup_{i \in I} |\beta_i| \cdot \lim_{n \to \infty} \sum_{i \in I} \mu_i^{(n)},$$

but the right-hand-side limit is precisely the probability of the program not to terminate, which, by assumption, is 0. Similar sufficient conditions for the β-termination of the program can be given for other kinds of vectors $\bar{\beta}$ (see § 3 where such a general condition is given).

The main question concerning this approach is: Can we always extend a given functional ψ to an invariant functional φ in the manner described above (in other words, is this method complete)? To see that this is indeed the case, we proceed as follows: Let $\{\beta_j\}_{j \in T}$ be given, with $\beta_j \geqq 0$ for each $j \in T$. For each $i \in I$ define

(2) $$\beta_i = \sum_{j \in T} \beta_j f_{ij}^*.$$

Each sum exists in an extended sense (β_i can be $+\infty$). Let $\tilde{\psi}$ be the functional $\tilde{\psi}(\bar{\mu}) = \sum_{i \in S} \beta_i \mu_i$.

THEOREM 1. *If $\tilde{\psi}$ satisfies* (V1), *then it also satisfies* (V2) *and* (V3). *On the other hand, if $\tilde{\psi}$ does not satisfy* (V1), *then $\psi(\bar{\mu}^*) = +\infty$.*

Proof. We first note that (V2) always holds. Clearly $\beta_i \geqq 0$ for each $i \in I$, and

$$\sum_{k \in S} P_{ik}\beta_k = \sum_{k \in S} P_{ik} \sum_{j \in T} f^*_{kj}\beta_j.$$

Interchanging the order of summation, we obtain

$$= \sum_{j \in T} \beta_j \left(\sum_{k \in S} P_{ik} f^*_{kj} \right).$$

But by the monotone convergence theorem,

$$\sum_{k \in S} P_{ik} f^*_{kj} = \sum_{k \in S} P_{ik} \lim_{n \to \infty} P^n_{kj} = \lim_{n \to \infty} \sum_{k \in S} P_{ik} P^n_{kj} = \lim_{n \to \infty} P^{n+1}_{ij} = f^*_{ij}.$$

Hence

$$\sum_{k \in S} P_{ik}\beta_k = \sum_{j \in T} \beta_j f^*_{ij} = \beta_i.$$

For $i \in T$, $P_{ik} = \delta_{ik}$, so certainly $\sum_{k \in S} P_{ik}\beta_k = \beta_i$. (Equality also holds when both sides are $+\infty$.) Hence (V2) is satisfied.

Next, suppose that (V1) holds for $\tilde{\psi}$. Again, by substituting the value of β_i, $i \in I$, and interchanging the order of summation, we obtain

$$\sum_{i \in S} \mu^0_i \beta_i = \sum_{j \in T} \beta_j \left(\sum_{l \in S} \mu^0_i f^*_{ij} \right) < \infty.$$

That is,

$$\sum_{i \in S} \mu^0_i \beta_i = \sum_{j \in T} \beta_j \mu^*_j < \infty.$$

This already shows that (C) holds, but it also follows from this that

$$R = \lim_{n \to \infty} \sum_{i \in I} \mu^{(n)}_i \beta_i = \tilde{\psi}(\bar{\mu}^0) - \psi(\bar{\mu}^*) = 0.$$

Hence (V3) also holds. If (V1) does not hold, we still have the above equality $\tilde{\psi}(\bar{\mu}^0) = \psi(\bar{\mu}^*) = +\infty$. Q.E.D.

This, of course, establishes the completeness of our verification method theoretically. That is, given a partial vector $\beta_j \geqq 0$, $j \in T$, there always exists a completion of it to a full vector β_i, $i \in S$, which satisfies (V2) and satisfies either (V1), (V3) and (C), or else the value $\psi(\bar{\mu}^*) = \sum_{j \in T} \beta_j \bar{\mu}^*_j = +\infty$.

In practice, of course, there are some difficulties. First, in order to obtain the above completion of $\bar{\beta}$, we need to know the matrix f^*_{ij}, which is generally unavailable. Similarly, the establishment of (V3) (for any completion of $\bar{\beta}$) requires the knowledge of $\bar{\mu}^{(n)}$ for every $n \geqq 0$ which is also unavailable.

A partial solution to these problems is given by the following characterization of the specific completion of $\bar{\beta}$ given by (2).

PROPOSITION 1. *Let $\beta_j \geqq 0$, $j \in T$, be given. Then the completion of $\bar{\beta}$ given by* (2) *is the smallest nonnegative P-regular* (*i.e. invariant*) *extension of the given β_j's.*

Proof. Let $\bar{\gamma} = \{\gamma_i\}_{i \in I}$ be such that $\gamma_i \geqq 0$ for all $i \in I$, and such that, together with the given $\bar{\beta}^0 \equiv \{\beta_j\}_{j \in T}$, $\bar{\gamma}$ is P-regular. Let us decompose P into blocks as follows:

$$P = \underbrace{\begin{pmatrix} Q & R \\ 0 & I \end{pmatrix}}_{I \quad T} \begin{matrix} \}\, I \\ \}\, T \end{matrix}.$$

The invariance of γ then can be written as

$$Q\bar{\gamma} + R\bar{\beta}^0 = \bar{\gamma}.$$

Since $\bar{\gamma} \geqq 0$, we have $Q\bar{\gamma} \geqq 0$ and hence $\bar{\gamma} \geqq R\bar{\beta}^0$. Continuing inductively in this manner, we obtain

$$\bar{\gamma} \geqq \sum_{k \geqq 0} Q^k R \bar{\beta}^0.$$

A computation completely analogous to the one performed in the preceding section yields

$$\left(\sum_{k \geqq 0} Q^k R \bar{\beta}^0\right)_i = \lim_{n \to \infty} \left(\sum_{j \in T} P_{ij}^{(n)} \beta_j\right) = \sum_{j \in T} f_{ij}^* \beta_j = \beta_i.$$

(Interchanging the limit and the sum is justified by the monotone convergence theorem.) Hence $\gamma_i \geqq \beta_i$ for each $i \in I$. Q.E.D.

Proposition 1 therefore yields a practical method for probabilistic verification. Starting with the given coefficients $\beta_j \geqq 0$, $j \in T$, we find the general solution to the invariance equations $P\bar{\beta} = \bar{\beta}$, that coincides with the given partial vector on T, and then choose the smallest nonnegative solution from which the value of $\psi(\bar{\mu}^*)$ can be readily obtained.

As an illustration, consider our example program with the coefficients $\beta_{(l_2,i)} = i$.

Example 1. The recurrence equations implied by $P\bar{\beta} = \bar{\beta}$ are

$$p\beta_{(l_1,i+1)} + qi = \beta_{(l_1,i)},$$

whose general solution is readily found to be

$$\beta_{(l_1,i)} = i + \frac{p}{q} + \frac{A}{p^i}.$$

By the requirement of minimality (and nonnegativity) we must choose $A = 0$ and are ensured, by the above two propositions, that conditions (V1)–(V3) are satisfied, so that calculation of $\psi(\bar{\mu}^*)$ can proceed as before.

Thus, we may summarize: In order to compute $\sum_{j \in T} \mu_j^* \beta_j$, find a completion β_i, $i \in I$, such that (V1) and (V2) are satisfied and either:

a) β_i, $i \in I$, is the minimal such solution; or

b) The program terminates with probability 1 and the β_i, $i \in I$, are uniformly bounded; or

c) It is possible to verify (V3) by some other means.

(More about such means will be discussed in the next section.) The desired value is then $\sum_{i\in S}\beta_i\mu_i^0$. In case a), if this latter value is $+\infty$, then so is the desired value of the "output assertion". Note that the family of nonnegative linear functionals enables us to express a very rich variety of program properties upon termination, such as:

(i) The probability of terminating at a particular state $j \in T$ (take $\beta_k = \delta_{kj}$).
(ii) The probability of termination (take $\beta_j \equiv 1$).
(iii) The expected value of some variable (as in our running example).
(iv) The expected running time of the program. (One way of doing this is to add a special "step-counter" variable to the program, and then find its expected value (cf. [RA]).)
(v) Higher moments of some variables, such as variance, etc.

Remark 1. If one knows that the program terminates almost surely (i.e. with probability 1), then one can generalize the second approach mentioned above to the case where the β_i's are not bounded. This is simply done by defining a sequence of partial vectors

$$\beta_j^{(N)} = \min(\beta_j, N), \qquad j \in T, \quad N = 1, 2, \cdots.$$

For each N, $\beta_j^{(N)} \leqq N$, $j \in T$. Hence, the smallest completion of the $\beta_j^{(N)}$'s, given by (2), is also uniformly bounded (by N).

We can therefore consider any bounded invariant completion $\bar{\beta}^{(N)}$ of the coefficients $\beta_j^{(N)}$, $j \in T$, and obtain

$$\psi^{(N)}(\bar{\mu}^*) = \sum_{j\in T} \mu_j^* \beta_j^{(N)} = \sum_{i\in S} \mu_i^0 \beta_i^{(N)} = \varphi^{(N)}(\mu^0).$$

By the monotone convergence theorem, $\lim_{N\to\infty} \psi^{(N)}(\bar{\mu}^*) = \sum_{j\in T}\mu_j^*\beta_j$. Hence, this sum is also equal to $\lim_{N\to\infty}\varphi^{(N)}(\bar{\mu}^0)$.

Remark 2. The techniques used in this section are rather standard in Markov chain theory (cf. [RV], [KSK]). It is pleasing to find out that the adaptation of Markov chain theory to the realm of program verification yields a natural and straightforward generalization of existing verification methods for deterministic programs.

We conclude this section with two additional illustrations of our method

Example 2. Consider the program

$x := 0;$
l_1: **while** $(t := \text{random}(\frac{1}{3}\delta_0 + \frac{1}{3}\delta_1 + \frac{1}{3}\delta_2)) \neq 2$ **do** $x := x + t;$
l_2: **halt**.

We associate states with the location in the program and with the value of x. Hence, we can write

$$I = \{(l_1, n) | n \geqq 0\}, \qquad T = \{(l_2, n) | n \geqq 0\}.$$

The nonzero transition probabilities for this program are

$$P_{(l_1,n),(l_1,n)} = \tfrac{1}{3}, \quad P_{(l_1,n),(l_1,n+1)} = \tfrac{1}{3}, \quad P_{(l_1,n),(l_2,n)} = \tfrac{1}{3}, \quad P_{(l_2,n),(l_2,n)} = 1.$$

Let us compute the terminal distribution $\bar{\mu}^*$. We thus fix $n \geqq 0$, and put

$$\beta_{(l_2,j)} = \delta_{j,n}.$$

We then want to extend $\bar{\beta}$ over the nonterminal state as well, so that it is invariant. The requirement $P\bar{\beta} = \bar{\beta}$ then becomes

$$\tfrac{1}{3}\beta_{(l_1,i)} + \tfrac{1}{3}\beta_{(l_1,i+1)} + \tfrac{1}{3}\delta_{i,n} = \beta_{(l_1,i)}, \quad i \geqq 0, \quad \text{or} \quad \beta_{(l_1,i+1)} = 2\beta_{(l_1,i)} - \delta_{i,n},$$

whose general solution is

$$\beta_{(l_1,i)}=\begin{cases}K\cdot 2^i, & i\leqq n,\\ K\cdot 2^i-2^{i-n-1}, & i>n.\end{cases}$$

To obtain the smallest nonnegative solution, we must choose $K=2^{-n-1}$. Hence, we conclude that

$$\mu^*_{(l_2,n)}=\sum_{i=0}^{n}2^{i-n-1}\mu^0_i.$$

In particular, since $\mu^0_{(l_1,i)}=\delta_{i,0}$, we obtain $\mu^*_{(l_2,n)}=1/2^{n+1}$. (Note that in this example we have actually computed the matrix f^*_{ij}, in a somewhat roundabout way.)

Example 3 (Gambler's ruin or Drunkard's walk). Consider the following program

$x:=n$; /* n is some positive integer */
l_1: **while** $x\neq 0$ **do** $x:=x+\text{random}\,(p\delta_{-1}+q\delta_1)$;
l_2: **halt**.

This program simulates a random walk on the nonnegative integers with 0 as an "absorbing barrier." This describes a process in which a gambler with an initial fortune n plays indefinitely against a house with unlimited fortune. In each game the player has a chance p of losing and a chance $q=1-p$ of winning. The process stops when the gambler loses all its money.

States are defined as in the preceding example, but in this case T contains only the state $(l_2,0)$. The nonzero transition probabilities are

$$P_{(l_1,j),(l_1,j-1)}=p,\qquad P_{(l_1,j),(l_1,j+1)}=q\quad\text{for } j>0,$$
$$P_{(l_1,0),(l_2,0)}=P_{(l_2,0),(l_2,0)}=1,$$

and the initial distribution is $\mu^0_{(l_1,j)}=\delta_{j,n}$. Let us compute the probability of the program termination, i.e. taking $\beta_{(l_2,0)}=1$. Extending $\bar{\beta}$ as usual, we are led to the following recurrence equations:

$$\beta_{(l_1,0)}=1,\qquad \beta_{(l_1,j)}=p\beta_{(l_1,j-1)}+q\beta_{(l_1,j+1)},\qquad j>0.$$

Solution of these recurrence equations amounts to solving the equation

$$q\lambda^2-\lambda+p=0$$

whose roots are 1 and p/q. If $p\neq\frac{1}{2}$, the roots are distinct and the general solution is

$$\beta_{(l_1,j)}=1+K\left[\left(\frac{p}{q}\right)^j-1\right].$$

If $p>\frac{1}{2}$, then $p/q>1$ and the minimality requirement forces us to choose $K=0$. Hence, $\beta_{(l_1,j)}=1$ for all $j\geqq 0$ so that the termination probability is $\beta_{(l_1,n)}=1$ (the gambler will almost surely be ruined).

If $p<\frac{1}{2}$, then $p/q<1$, and we choose

$$K=\inf\frac{1}{1-(p/q)^j}=1.$$

Hence, $\beta_{(l_1,j)}=(p/q)^j$, so that the termination probability is $(p/q)^n$.

Finally, if $p=\frac{1}{2}$, the λ-equation has two identical roots, and the general solution for the β's is

$$\beta_{(l_1,j)}=1+Kj.$$

The minimality condition implies $K=0$ so that $\beta_{(l_1,n)}=1$ and the program terminates almost surely in this case.

As a related example, consider the case $p=\frac{1}{2}$ in the above program. Although the program terminates almost surely in this case, it is well known [CH] that it is not expected to terminate. We will establish this fact using our method. To do so, we need to introduce an additional step-counter variable c, and modify the program as follows:

$[x, c] := [n, 0]$;
l_1: **while** $x \neq 0$ **do** $[x, c] := [x + \text{random}(\frac{1}{2}\delta_{-1} + \frac{1}{2}\delta_1), c+1]$;
l_2: **halt**.

The program states are now

$$I = \{(l_1, c, n) | c \geqq 0, n \geqq 0\}, \qquad T = \{(l_2, c, 0) | c \geqq 0\}.$$

The nonzero transition probabilities are

$$P_{(l_1,c,j),(l_1,c+1,j+1)} = P_{(l_1,c,j),(l_1,c+1,j-1)} = \tfrac{1}{2}, \qquad j > 0$$

$$P_{(l_1,c,0),(l_2,c,0)} = P_{(l_2,c,0),(l_2,c,0)} = 1.$$

The initial distribution is $\mu^0_{(l_1,0,n)} = 1$, and zero elsewhere. We wish to compute the expected value of c upon termination, so that we begin with the partial vector $\beta_{(l_2,c,0)} = c$, $c \geqq 0$, and we wish to extend it to an invariant vector $\bar{\beta}$. Instead of doing so directly, we use the limiting approach suggested in Remark 1 above. That is, let $M \geqq 0$ be an integer, and define

$$\beta^M_{(l_2,c,0)} = \begin{cases} c, & c \leqq M, \\ 0, & c > M. \end{cases}$$

Since β^M is uniformly bounded on T, and the program is already known to terminate almost surely, *any* invariant completion $\bar{\beta}^M$ can be used to compute the desired functional. We claim that the following is such an invariant completion:

$$\beta^M_{(l_1,c,j)} = \sum_{k=0}^{\lfloor (M-c-j)/2 \rfloor} \frac{c+j+2k}{2^{j+2k}} \left[\binom{j+2k-1}{k} - \binom{j+2k-1}{k-1} \right], \qquad j \geqq 1,$$

(note that the sum vanishes if $c+j > M$), and

$$\beta^M_{(l_1,c,0)} = \begin{cases} c, & c \leqq M, \\ 0, & c > M. \end{cases}$$

To verify that $\bar{\beta}^M$ has the desired properties, one has to check that the following equations hold:

$$\beta^M_{(l_1,c,0)} = \beta^M_{(l_2,c,0)} \quad \text{and} \quad \beta^M_{(l_1,c,j)} = \tfrac{1}{2}\beta^M_{(l_1,c+1,j+1)} + \tfrac{1}{2}\beta^M_{(l_1,c+1,j-1)}, \qquad j \geqq 1.$$

The first equation is immediate, and the second can easily be checked. Since β^M is uniformly bounded on I (it is zero on all but a finite number of components), we conclude that

$$E^M(c) = \sum_{c=0}^{M} c\mu^*_{(l_2,c,0)} = \beta^M_{(l_1,0,n)} = \sum_{k=0}^{\lfloor (M-n)/2 \rfloor} \frac{n+2k}{2^{n+2k}} \left[\binom{n+2k-1}{k} - \binom{n+2k-1}{k-1} \right].$$

Hence, if we let $M \to \infty$, we find that the expected value of c is the sum of the infinite series appearing above. Using Stirling's formula, we find that the kth term of this series is of the order of $k^{-1/2}$, so that the series diverges and the expected value of c is infinite. (This method can be used to show that the αth moment of c is finite for $\alpha < \frac{1}{2}$ and infinite for $\alpha > \frac{1}{2}$.)

3. A dual approach—expected number of visits. A recent work of Ramshaw [RA] suggests an alternative approach to probabilistic program verification. Although he does not use Markov chains in his approach, it turns out that his approach can be easily and naturally described in terms of the Markov chain model that we have been using. This leads to a much more compact description of his method, helps to explain the problems that it faces and the (partial) solutions to these problems suggested by Ramshaw, and also makes it easier to generalize this approach and to connect it with our first approach as given in the preceding section. All this will be done in this section.

Intuitively, the approach that we have taken in the first section was to record the program behavior by taking "snapshots" of the distribution of all program states, at different times during execution. An invariant functional is thus a linear "assertion" about this distribution that does not change from one snapshot to another. The approach that Ramshaw takes is orthogonal to ours, in the sense that he takes an "infinite-exposure" picture, of each program state separately, throughout the program execution. His approach can be formally explained in terms of the Markov chain model as follows:

Let P be the transition probability of the program. Decomposing it into blocks as we did in the previous section, we obtain

$$P = \underbrace{\begin{pmatrix} Q & R \\ 0 & I \end{pmatrix}}_{I \quad T} \begin{matrix} \}I \\ \}T \end{matrix}$$

(i.e., Q_{ij} is the probability of going from $i \in I$ to $j \in I$, and R_{ij} is the probability of going from $i \in I$ to $j \in T$). Consider a modified transition matrix defined as

$$\tilde{P} = \begin{pmatrix} Q & R \\ 0 & 0 \end{pmatrix}.$$

This matrix corresponds to a (substochastic) process in which, once the process reaches a terminal state, it stops right there.

Let $\tilde{\mu}^{(n)}$ be the distribution of program states after n steps of the revised process, i.e. $\tilde{\mu}^{(n)} = \bar{\mu}^0 \tilde{P}^n$. Define a vector $\bar{v}$ over S as follows:

$$\bar{v} = \sum_{n=0}^{\infty} \tilde{\mu}^{(n)}. \tag{3}$$

For each $i \in S$, v_i is the expected number of visits at state i in an execution, given the initial distribution $\bar{\mu}^0$. Following Markov chain terminology, $\bar{v}$ is a pure potential measure induced by the charge $\bar{\mu}^0$. Note that $\bar{v}$ is always defined in an extended sense, but need not be finite. However, if $i \in T$, v_i is always finite and in fact we have $v_i = \mu_i^*$. This follows from the fact that terminal states are visited at most once in the revised process.

From the definition of $\bar{v}$ we have immediately the following

CLAIM. *$\bar{v}$ is the smallest nonnegative solution of the equation*

$$\bar{v} = \bar{v}\tilde{P} + \bar{\mu}^0. \tag{4}$$

Ramshaw's approach is to consider the quantities v_i, and to introduce assertions about them having the following restricted form:

$$\sum_{i \in A} v_i = e$$

where $A \subseteq S$ is a subset of states all having the same program location (he refers to these assertions as "vanilla" assertions). His method is to verify that these assertions are consistent with (4), i.e., to show that any vector $\bar{v}$ satisfying (4) also satisfies the assertions. This is done by a generalization of the standard inductive assertions method, but may not always work. In fact, the assertions must be of a special structure to allow his inference rules to be applicable.

Ramshaw shows that under certain conditions (roughly amounting to requiring that $\bar{v}$ be finite) this proof method is sound and yields some information about $\bar{\mu}^*$, given by those assertions that are planted at the program termination point. Ramshaw does not bother to actually solve (4), and so the main problem that he faces is to show that his assertions, even when consistent with (4), do actually describe the smallest solution of (4). This creates the possibility of obtaining nonminimal solutions, such as his so-called "time bombs", which may not yield the desired $\bar{\mu}^*$.

Having stated the basic nature of Ramshaw's approach, we will not follow his method of estimating $\bar{v}$. Rather, we view the solution of (4) as the main goal of this approach. In this regard, there are several additional possibilities for estimating $\bar{v}$. For example,

A) Let $\bar{u}$ be any nonnegative solution of

$$\bar{u} = \bar{u}\tilde{P} + \bar{\mu}^0$$

or even of

$$\bar{u} \geqq \bar{u}\tilde{P} + \bar{\mu}^0.$$

Then for every i, $v_i \leqq u_i$, so that $\bar{u}$ is an approximation from above to the desired $\bar{v}$.

B) Define the sequence of vectors

$$\bar{v}^0 = \bar{\mu}^0, \qquad \bar{v}^{n+1} = \bar{v}^n\tilde{P} + \bar{\mu}^0, \qquad n \geqq 0.$$

Then for every i and $n \geqq 0$, $v_i^n \leqq v_i$, so that $\bar{v}^n$ is an approximation from below for $\bar{v}$.

To illustrate this approach, let us return to our running example of the program that searches for a first appearance of 1 in an infinite sequence of independent draws of 0 and 1. Equation (4) then has the following form:

$$v_{(l_1,i)} = \mu^0_{(l_1,i)} + pv_{(l_1,i-1)}, \qquad i \geqq 1,$$

$$v_{(l_1,0)} = \mu^0_{(l_1,0)},$$

$$v_{(l_2,i)} = qv_{(l_1,i)}, \qquad i \geqq 0,$$

and the (unique) solution is easily found to be

$$v_{(l_1,i)} = p^i, \qquad v_{(l_2,i)} = p^iq = \mu^*_{(l_2,i)}, \qquad i \geqq 0.$$

Drawing an analogy to the standard verification techniques, we can compare this second approach to the computational induction method [PA], where information about intermediate program states is derived inductively from information about its input states. Comparing the two methods presented in this paper, we may view the first one as being *goal-directed*, in that it draws from the program output requirements conditions that should hold at the intermediate and input program states, and only then checks them against the input information about the program. On the other hand, the second method is *input-directed*, in that it draws information about intermediate and terminal program states from the input information and then computes the output data from this information.

There are two main disadvantages of the second approach. The first is that it is, as just pointed out, input-dependent. Hence, if the program input distribution is not fixed then we may have to recompute the vector $\bar{v}$ afresh for each new distribution $\bar{\mu}^0$. By contrast, it is more natural to assume that the questions about the program terminal states are fixed, and we can process each of them using our first method independently of any input distribution. Then for each input distribution, we can compute the required output quantities immediately.

A second disadvantage is that the expected number of visits at an intermediate state need not be finite even if, say, the program terminates almost surely. Hence in solving (4), we may find that certain v_i's are $+\infty$. This is not a major obstacle, since (4) holds even in such a case. This means that each finite component v_i, $i \in I$, cannot depend on any infinite components, so that these infinite components correspond to states from which the program almost surely diverges and cannot reach a terminal state with a positive probability.

Aside from independent computation of the $\bar{v}$ or approximations thereof, we can connect solutions to (4) to linear invariant functionals. Let $\bar{u}$ be any solution to $\bar{u} \geqq \bar{u}\tilde{P} + \bar{\mu}^0$ and $\bar{\beta}$ any solution to $P\bar{\beta} = \bar{\beta}$. By partitioning $\bar{u} = (\bar{u}_1, \bar{u}_2)$ and $\bar{\beta} = (\bar{\beta}_1, \bar{\beta}_2)$ according to the partitioning of S into I, T, we obtain the following equations satisfied by each:

$$\bar{u}_1 \geqq \bar{u}_1 Q + \bar{\mu}_1^0, \quad \bar{u}_2 \geqq \bar{u}_1 R + \bar{\mu}_2^0, \quad \bar{\beta}_1 = Q\bar{\beta}_1 + R\bar{\beta}_2.$$

Obviously $\bar{\mu}_1^{(n)} = \bar{\mu}_1^0 Q^n$. Thus $\bar{v}_1 = \sum_{n=0}^{\infty} \bar{\mu}_1^{(n)}$.

Consequently $(\bar{u}_1 \cdot \bar{\beta}_1) \geqq (\bar{v}_1 \cdot \bar{\beta}_1) = \sum_{n=0}^{\infty} (\bar{\mu}_1^{(n)} \cdot \bar{\beta}_1) = \sum_{n=0}^{\infty} R_n$ where R_n is the remainder such that $\lim_{n\to\infty} R_n = 0$ is the required (V3) condition. Hence, we have

PROPOSITION 2. *If for some $\bar{u}_1$ satisfying*

$$\bar{u}_1 \geqq \bar{u}_1 Q + \bar{\mu}_1^0$$

and some $\bar{\beta} = (\bar{\beta}_1, \bar{\beta}_2)$ satisfying

$$\bar{\beta}_1 = Q\bar{\beta}_1 + R\bar{\beta}_2$$

the product $(\bar{u}_1 \cdot \bar{\beta}_1) < \infty$, then (C) *holds; i.e.,*

$$\bar{\mu}_2 \cdot \bar{\beta}_2 = \bar{\mu}_1^0 \cdot \bar{\beta}_1 + \bar{\mu}_2^0 \cdot \bar{\beta}_2.$$

Note, however, that this is only a sufficient condition which implies the $\bar{\beta}$-termination of the program (i.e. condition (V3)), but is not necessarily equivalent to it. To illustrate this point, suppose that $\bar{\beta} \equiv 1$. This is an invariant vector whose restriction to T yields a functional that computes the probability of termination. If $\bar{\beta}$ satisfies (V1)–(V3), and therefore also (C), then it follows that the program almost surely terminates. On the other hand, the above condition, even for $\bar{u} = \bar{v}$, reads

$$\sum_{i \in I} v_i < \infty.$$

However, we have

LEMMA 1. *$\sum_{i \in I} v_i < \infty$ if and only if the program has a finite expectation of termination (i.e., the expected length of stay in I is finite), which is then equal to that sum.*

Proof. The expectation of program termination is

$$\sum_{n\geq 0} n\cdot \text{Prob (the program terminates in exactly } n \text{ steps)}$$

$$= \sum_{n\geq 1} \text{Prob (the program terminates only after } n \text{ or more steps)}$$

$$\leqq \sum_{n\geq 0} \text{Prob (the program is not yet in } T \text{ after } n \text{ steps)}$$

$$= \sum_{n\geq 0}\sum_{i\in I} \tilde{\mu}_i^{(n)} = \sum_{i\in I} v_i.$$

Hence $\sum_{i\in I} v_i<\infty$ implies a finite expectation of termination.

Conversely, if the program has a finite expectation to terminate, then it terminates almost surely, which makes the inequality in the above formulae into an equality. hence $\sum_{i\in I} v_i$ is equal to the expectation to terminate, which is finite. Q.E.D.

Hence the above condition requires that the program have a finite expected execution length, which in general is stronger than the requirement that it terminate almost surely.

Nevertheless, we have the following alternative approach to verification:

(i) Find any nonnegative solution $\bar{u}$ of (4) (even with an inequality).

(ii) Find an invariant nonnegative completion $\bar{\beta}$ of the coefficients of the given functional on T.

(iii) Check that $(\bar{u}_1\cdot\bar{\beta}_1)<\infty$, where $\bar{\beta}_1=\bar{\beta}|_I$, $\bar{u}_1=\bar{u}|_I$.

If so, $\bar{\beta}$-termination, and hence also (C), are assured.

An example of this procedure will be given in the following section. We will conclude this section with an example of a straightforward calculation of $\bar{v}$. Consider the Gambler's Ruin program given in Example 3, with $p=q=\frac{1}{2}$. Following the notation of § 2, equations (4) have the form

$$\begin{aligned}
&v_{(l_1,0)}=\tfrac{1}{2}v_{(l_1,1)},\\
&v_{(l_1,1)}=\tfrac{1}{2}v_{(l_1,2)},\\
&v_{(l_1,j)}=\tfrac{1}{2}v_{(l_1,j-1)}+\tfrac{1}{2}v_{(l_1,j+1)}, \qquad j\neq n,\quad j>1,\\
&v_{(l_1,n)}=\tfrac{1}{2}v_{(l_1,n-1)}+\tfrac{1}{2}v_{(l_1,n+1)}+1,\\
&v_{(l_2,0)}=v_{(l_1,0)}.
\end{aligned}$$

Putting $v_{(l_1,0)}=a$, we have the following general solution:

$$v_{(l_1,j)}=\begin{cases} a, & j=0,\\ 2aj, & j=1,\cdots,n,\\ 2aj-2(j-n), & j>n.\end{cases}$$

Since we want the smallest nonnegative solution of (4), we must take $a=1$, and we obtain the solution

$$v_{(l_1,j)}=\begin{cases} 1, & j=0,\\ 2j, & j=1,\cdots,n,\\ 2n, & j>n.\end{cases}$$

Hence $\mu^*_{(l_2,0)} = v_{(l_2,0)} = v_{(l_1,0)} = 1$. This means that the program terminates almost surely. However, $\sum_{i\in I} v_i = +\infty$, so that the program is not expected to terminate, in accordance with the results obtained, in a much more complicated manner, in the preceding section.

4. Additional examples. In this section we will illustrate the verification methods developed in the two preceding sections, as applied to two nontrivial example programs.

Example 4. Consider the following program ($0<\alpha<1$ is fixed):

```
y := 0; n := 1;
while y < α do
    y := y + 1/2^n random (½δ_0 + ½δ_1);
    n := n + 1;
od.
```

For simplicity, we identify states only by the values of y and n. Thus, $S=\{(y, n): n\geqq 1, y\in D_{n-1}\}$, where D_j is the set of all dyadic fractions having j binary digits. The terminating states are

$$T=\{(y, n): n\geqq 1, y\in D_{n-1}, y>\alpha \text{ and the } (n-1)\text{st digit of } y \text{ is } 1\},$$

and the transition probabilities are

$$\left.\begin{array}{l} P_{(y,n),(y,n+1)}=\frac{1}{2} \\ P_{(y,n),(y+1/2^n,n+1)}=\frac{1}{2} \end{array}\right\} \quad y\in D_{n-1}, \quad y\leqq\alpha \quad (\text{i.e. } (y, n)\in I),$$

$$P_{(y,n),(y,n)}=1 \qquad y\in D_{n-1}, \quad y>\alpha \quad (\text{i.e. } (y, n)\in T).$$

Let us compute the termination probability of this program; that is, we wish to compute

$$\psi(\bar{\mu}^*)=\sum_{(y,n)\in T}\mu^*_{(y,n)}.$$

Let us extend ψ to an invariant nonnegative functional φ:

$$\varphi(\bar{\mu})=\sum_{(y,n)\in S}\beta_{(y,n)}\mu_{(y,n)}, \quad \text{where } \beta_{(y,n)}=1 \quad \text{for } (y, n)\in T.$$

The invariance of φ implies that for each $(y, n)\in I$ we must have

$$\beta_{(y,n)}=\tfrac{1}{2}\beta_{(y,n+1)}+\tfrac{1}{2}\beta_{(y+(1/2^n),n+1)}.$$

Of course, these equations have the solution $\beta_{(y,n)}\equiv 1$, but this is too large. (Note in general that such an extension implies that $\varphi(\bar{\mu}^0)=1$ for any initial distribution $\bar{\mu}^0$. Hence, this extension is the smallest nonnegative invariant extension of ψ if and only if the program terminates almost surely, regardless of the initial distribution.) In our case, we have a smaller solution:

$$\beta_{(y,n)}=\begin{cases} 0, & 2^{n-1}(\alpha-y)>1, \\ 1, & 2^{n-1}(\alpha-y)<0, \\ 1-2^{n-1}(\alpha-y), & \text{otherwise.} \end{cases}$$

Indeed, let us verify that these coefficients satisfy the above recurrence equation. Suppose first that $\beta_{(y,n)}=1$. This happens when $y\geqq\alpha$, and then both $\beta_{(y,n+1)}=\beta_{(y+1/2^n,n+1)}=1$, so the equation is satisfied. Next suppose $\beta_{(y,n)}=0$. This happens when $y\leqq\alpha-1/2^{n-1}$. Then obviously $y\leqq\alpha-1/2^n$ so that $\beta_{(y,n+1)}=0$, and also

$y+1/2^n \leqq \alpha - 1/2^n$, so that $\beta_{(y+1/2^n,n+1)}=0$. Hence the equation is satisfied in this case, too. (Note that this corresponds to the case where the program never terminates if it reaches the state (y, n).) Finally, assume that

$$0<2^{n-1}(\alpha-y)<1, \quad \text{i.e.,} \quad \alpha-\frac{1}{2^{n-1}}<y<\alpha.$$

Two subcases are possible. If $\alpha-1/2^n \leqq y<\alpha$ then $y+1/2^n \geqq \alpha$ and so $\beta_{(y+1/2^n,n+1)}=1$, whereas $\beta_{(y,n+1)}=1-2^n(\alpha-y)$. Hence

$$\tfrac{1}{2}\beta_{(y+1/2^n,n+1)}+\tfrac{1}{2}\beta_{(y,n+1)}=\tfrac{1}{2}+\tfrac{1}{2}-2^{n-1}(\alpha-y)=1-2^{n-1}(\alpha-y)=\beta_{(y,n)}.$$

If $\alpha-1/2^{n-1}<y<\alpha-1/2^n$, then $\alpha-1/2^n<y+1/2^n<\alpha$, so that

$$\beta_{(y+1/2^n,n+1)}=1-2^n\left(\alpha-y-\frac{1}{2^n}\right),$$

whereas $\beta_{(y,n+1)}=0$. Hence

$$\frac{1}{2}\beta_{(y,n+1)}+\frac{1}{2}\beta_{(y+1/2^n,n+1)}=0+\frac{1}{2}-2^{n-1}\left(\alpha-y-\frac{1}{2^n}\right)=1-2^{n-1}(\alpha-y)=\beta_{(y,n)}.$$

Note that we have not shown that these $\beta_{(y,n)}$'s yield the smallest invariant extension of ψ (although this is indeed the case). However, we can apply the duality principle stated in § 3 by computing the vector $\bar{v}_1$ and checking that $(\bar{v}_1 \cdot \bar{\beta}_1)<\infty$. As it turns out, computation of $\bar{v}_1$ in this case is simpler, because each nonterminating state (y, n) can be reached from only one preceding state. Specifically, we have

$$v_{(0,1)}=1, \qquad v_{(y,n)}=\tfrac{1}{2}v_{(y',n-1)}, \qquad n>1, \quad y \in D_{n-1}, \quad y \leqq \alpha,$$

where $y' \in D_{n-2}$ consists of the first $n-2$ digits of y. Thus, for each $(y, n) \in I$, we have

$$v_{(y,n)}=\frac{1}{2^{n-1}}.$$

Now we have

$$\sum_{(y,n)\in I} v_{(y,n)}\beta_{(y,n)}<\infty$$

because for each n there are at most two $y \in D_{n-1}$ for which $(y, n) \in I$ and $\beta_{(y,n)} \neq 0$. It therefore follows that

$$\psi(\bar{\mu}^*)=\varphi(\bar{\mu}^0)=\varphi(\delta_{(0,1)})=\beta_{(0,1)}=1-\alpha.$$

Expectation of y upon termination. Here we consider the following functional:

$$\psi(\bar{\mu}^*)=\sum_{(y,n)\in T} y\bar{\mu}^*_{(y,n)}$$

which is extended to

$$\varphi(\bar{\mu})=\sum_{(y,n)\in I} \beta_{(y,n)}\bar{\mu}_{(y,n)}+\sum_{(y,n)\in T} y\bar{\mu}_{(y,n)},$$

which has to satisfy the same invariance equations

$$\tfrac{1}{2}\beta_{(y,n+1)}+\tfrac{1}{2}\beta_{(y+1/2^n,n+1)}=\beta_{(y,n)}, \qquad (y, n) \in I.$$

Let us guess the following solution:

$$\beta_{(y,n)} = \begin{cases} 0, & y < \alpha - \dfrac{1}{2^{n-1}}, \\ A_n, & \alpha - \dfrac{1}{2^{n-1}} \leqq y < \alpha, \\ y, & \alpha \leqq y. \end{cases}$$

(Note that $\beta_{(y,n)} = A_n$ for exactly one $y \in D_{n-1}$.) Let us check the recurrence equations: If $\beta_{(y,n)} = 0$, $y < \alpha - 1/2^{n-1}$, then both y and $y + 1/2^n$ are less than $\alpha - 1/2^n$, so that the left-hand side is also 0. Suppose then that $\alpha - 1/2^{n-1} \leqq y < \alpha$. Two cases are possible.

Case I. $\alpha - 1/2^n \leqq y < \alpha$, which happens if the nth digit of α is 0 (assume an infinite binary representation of α). Then $\beta_{(y,n+1)} = A_{n+1}$ and $\beta_{(y+1/2^n,n+1)} = y + 1/2^n$. Hence we have

$$\frac{1}{2} A_{n+1} + \frac{1}{2}\left(y + \frac{1}{2^n}\right) = A_n \quad \text{if } \alpha_n = 0.$$

Case II. $\alpha - 1/2^{n-1} \leqq y < \alpha - 1/2^n$, which happens if $\alpha_n = 1$. Then $\beta_{(y,n+1)} = 0$ and $\beta_{(y+1/2^n,n+1)} = A_{n+1}$. Hence

$$\tfrac{1}{2} A_{n+1} = A_n \quad \text{if } \alpha_{n=1}.$$

Combining both cases, we can write

$$A_{n+1} = 2A_n - (1 - \alpha_n)\left(y_n + \frac{1}{2^n}\right).$$

where y_n is the binary fraction represented by the first $(n-1)$ digits of α, i.e.

$$y_n = \sum_{j=1}^{n-1} \frac{\alpha_j}{2^j}.$$

The solution of the general equation

$$A_{n+1} = 2A_n - C_n, \qquad n \geqq 1,$$

is given by

$$A_n = 2^{n-1}\left[A_1 - \sum_{k=1}^{n-1} \frac{C_k}{2^k}\right].$$

Hence, as usual, we have to choose

$$A_1 = \sum_{k=1}^{\infty} \frac{C_k}{2^k}.$$

Hence, the expectation of y is $\beta_{(0,1)} = A_1$ (note that $(\bar{v}_1 \cdot \bar{\beta}_1) < \infty$ in this case too)

$$= \sum_{k=1}^{\infty} \frac{1}{2^k}(1 - \alpha_k)\left(y_k + \frac{1}{2^k}\right) = \underbrace{\sum_{k=1}^{\infty} \frac{1 - \alpha_k}{2^{2k}}}_{Q} + \underbrace{\sum_{k=1}^{\infty} \frac{1 - \alpha_k}{2^k} \sum_{j=1}^{k-1} \frac{\alpha_j}{2^j}}_{S}.$$

To compute this sum, we use the following equality:

$$(1-\alpha)^2 = \sum_{k,j=1}^{\infty} \frac{(1-\alpha_j)(1-\alpha_k)}{2^j 2^k} = 2\sum_{j<k} \frac{(1-\alpha_k)(1-\alpha_j)}{2^k 2^j} + \sum_{k=1}^{\infty} \frac{1-\alpha_k}{2^{2k}}$$

$$= -2S + Q + 2\sum_{k=1}^{\infty} \frac{1-\alpha_k}{2^k}\left(\sum_{j=1}^{k-1} \frac{1}{2^j}\right)$$

$$= -2S + Q + 2(1-\alpha) - 4Q = -2S - 3Q + 2(1-\alpha).$$

Hence

$$\beta_{(0,1)} = S + Q = \frac{2(1-\alpha)-(1-\alpha)^2 - Q}{2} = \frac{1-\alpha^2}{2} - \frac{Q}{2}.$$

Thus the derived expectation of y is less than the expectation of y under uniform distribution of y in $(\alpha, 1]$ by $Q/2$, where

$$Q = \sum_{k=1}^{\infty} \frac{1-\alpha_k}{2^{2k}}.$$

Example 5. *Maximum component in a random sequence.* Finally we give an example of an average-case analysis of a nonprobabilistic program, using our methods. This example will require that we deal with continuous distributions; however, our methods can be easily extended to the continuous case, as will be demonstrated below, although we will not justify this extension formally. This example is considered by Knuth [KN] and is analyzed by Ramshaw [RA] by his system of "frequentistic" assertions. Consider the following program, which finds the maximum among n random elements, all drawn independently from a uniform distribution on $[0, 1]$ (λ denotes the Lebesgue measure on $[0, 1]$; note that here we allow draws out of a continuous distribution):

```
M := random (λ); C := 0;
for J := 2 to n do
  if (t := random (λ)) > M then C := C+1; M := t; fi
od.
```

C is a counter variable added to the program in order to measure the number of assignments to M. This number, the number of "left-to-right" maxima occurring in the sequence, is the performance parameter that we wish to estimate. The program states can be compactly represented by the values c, m, j of C, M, J respectively at entrance to the loop. (We will use the convention that $J = n+1$ designates terminal states.) Furthermore, since m varies over a continuous distribution, we will regard $\bar{\mu}^0$ and $\bar{\mu}^*$ as density functions in m, and as distributions in c and j. (This is the first example of a continuous distribution; the results obtained so far can be easily generalized to this case, by simply replacing sums by the appropriate integrals.)

We thus wish to compute

$$\psi(\bar{\mu}^*) = \int_0^1 \sum_{c=0}^{n-1} c\mu^*_{c,n+1}(m)\, d\lambda(m).$$

We extend ψ to an invariant functional φ over S, so that

$$\varphi(\bar{\mu}) = \sum_{j=2}^{n} \sum_{c=0}^{n-1} \int_0^1 \gamma_{c,j}(m)\mu_{c,j}(m)\, d\lambda(m) + \sum_{c=0}^{n-1} c \int_0^1 \mu_{c,n+1}(m)\, d\lambda(m).$$

In a continuous model, the transition probability matrix has to be written as a kernel representing "transition density" in the continuous parameter m (see Revuz [RV] for details). Thus, we have the following nonzero entries:

$$\left.\begin{aligned} P_{(c,j,m),(c,j+1,u)} &= m\delta_m(u), \\ P_{(c,j,m),(c+1,j+1,u)} &= \chi_{(m,1]}\cdot\lambda(u), \end{aligned}\right\} \quad j \leqq n,$$
$$P_{(c,n+1,m),(c,n+1,u)} = \delta_m(u).$$

The first line describes an iteration step of the loop at which C has not been incremented, so that m does not change; the total weight of this transition is m. The second line describes an iteration step at which C is incremented, and the new value (u) of m is greater than the old value. The third line describes terminating "transitions".

The invariance of φ requires the following recurrence equations to hold:

$$\gamma_{c,j}(m) = m\gamma_{c,j+1}(m) + \int_m^1 \gamma_{c+1,j+1}(u)\,d\lambda(u), \qquad j \leqq n,$$

where $\gamma_{c,n+1}(m) = c$. We will prove that

$$\gamma_{c,n+1-r}(m) = c + \sum_{i=1}^{r} \frac{1-m^i}{i},$$

by induction on r. The equality holds for $r = 0$. Assume it holds for some r. Then

$$\begin{aligned} \gamma_{c,n-r}(m) &= m\gamma_{c,n+1-r}(m) + \int_m^1 \gamma_{c+1,n+1-r}(u)\,d\lambda(u) \\ &= m\left[c + \sum_{i=1}^{r} \frac{1-m^i}{i}\right] + \int_m^1 \left[c+1+\sum_{i=1}^{r}\frac{1-u^i}{i}\right] du \\ &= mc + m\sum_{i=1}^{r}\frac{1-m^i}{i} + (1-m)c + (1-m) + \sum_{i=1}^{r}\frac{(1-m)-(1-m^{i+1})/(i+1)}{i} \\ &= c + \sum_{i=1}^{r}\frac{1-m^{i+1}}{i+1} + 1 - m = c + \sum_{i=1}^{r+1}\frac{1-m^i}{i}. \end{aligned}$$

Note that here φ is uniquely determined, and so must satisfy (V1)–(V3). Hence $\varphi(\bar{\mu}^0) = \psi(\bar{\mu}^*)$ where $\bar{\mu}^0$ is the initial distribution on entry to the loop. This initial distribution, however, is concentrated on $c = 0$ and $j = 2$ and is uniformly distributed in m. Hence we have

$$\begin{aligned} \psi(\bar{\mu}^*) = \int_0^1 \gamma_{0,2}(m)\,d\lambda(m) &= \int_0^1 \sum_{i=1}^{n-1}\frac{1-m^i}{i}\,d\lambda(m) \\ &= \sum_{i=1}^{n-1}\frac{1-1/(i+1)}{i} = \sum_{i=2}^{n}\frac{1}{i} = H_n - 1, \end{aligned}$$

which is the result in [KN, § 1.2.10].

REFERENCES

[CH] K. L. CHUNG, *Markov Chains with Stationary Transition Probabilities*, Springer-Verlag, New York, 1967.

[FL] R. W. FLOYD, *Assigning meaning to programs*, in Proc. Mathematical Aspects of Computer Science, J. T. Schwartz, ed., American Mathematical Society, New York, 1967, pp. 19–32.

[HS] S. HART AND M. SHARIR, *Concurrent probabilistic programs, or how to schedule if you must*, Technical Report, School of Mathematical Sciences, Tel Aviv Univ., Tel Aviv, Israel, 1982.

[HS2] ——, *Propositional probabilistic temporal logics*, Technical Report, School of Mathematical Sciences, Tel Aviv Univ., Tel Aviv, Israel, 1983.

[HSP] S. HART, M. SHARIR AND A. PNEULI, *Termination of probabilistic concurrent programs*, Proc. 9th ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, 1982, pp. 1–7; ACM Trans. Prog. Languages and Systems, 5 (1983).

[KSK] J. G. KEMENY, J. L. SNELL AND A. W. KNAPP, *Denumerable Markov Chains*, 2nd ed., Springer-Verlag, New York, 1976.

[KN] D. E. KNUTH, *The Art of Computer Programming*, Vol. I, Addison-Wesley, Reading, MA, 1968.

[KO] D. KOZEN, *Semantics of probabilistic programs*, in 20th IEEE Symposium on the Foundations of Computer Science, Institute of Electronics and Electrical Engineers, New York, 1979, pp. 101–114.

[LR] D. LEHMANN AND M. O. RABIN, *On the advantages of free choice*, in Proc. 8th ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, 1981, pp. 133–138.

[LS] D. LEHMANN AND S. SHELACH, *Reasoning with time and chance*, Technical Report, The Hebrew Univ., Jerusalem, Israel, 1982.

[MW] J. MORRIS AND B. WEGBREIT, *Subgoal induction*, Comm. ACM, 20 (1977), pp. 209–222.

[PA] D. PARK, *Fixpoint induction and proofs of program properties*, in Machine Intelligence, B. Meltzer and D. Michie, eds., Vol. 5, Edinburgh Univ. Press, Edinburgh, 1969, pp. 59–78.

[RA] L. H. RAMSHAW, *Formalizing the analysis of algorithms*, Ph.D. thesis, Report STAN-CS-79-742, Department of Computer Science, Stanford Univ., Stanford, CA, June, 1979.

[RB] M. O. RABIN, *Probabilistic algorithms*, in Algorithms and Complexity—New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York, 1976, pp. 21–40.

[RE] J. H. REIF, *Logics for probabilistic programs*, in Proc. 12th ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1980, pp. 8–13.

[RM] C. V. RAMAMOORTHY, *Discrete Markov analysis of computer programs*, in Association for Computing Machinery, 20th National Conference, Cleveland, Association for Computing Machinery, New York, pp. 386–392.

[RV] D. REVUZ, *Markov Chains*, North-Holland, Amsterdam, 1975.

[SD] N. SAHEB-DJAHROMI, *Probabilistic* LCF, in Proc. Conference on the Mathematical Foundations of Computer Science, Springer-Verlag, New York, 1978, pp. 442–451.

[WE] B. WEGBREIT, *Verifying program performance*, J. Assoc. Comput. Mach., 23 (1976), pp. 691–699.

SIAM J. COMPUT.
Vol. 13, No. 2, May 1984
© 1984 Society for Industrial and Applied Mathematics
007

CONSTRAINED OPTIMUM COMMUNICATION TREES AND SENSITIVITY ANALYSIS*

SUNITA AGARWAL†, A. K. MITTAL‡ AND P. SHARMA§

Abstract. Consider n cities with specified communication requirements between all pairs of cities. An optimum communication tree has the property that among all the spanning trees connecting the n cities, the sum of its costs of communication for the $n(n-1)/2$ pairs of cities is minimum. The cost of communication between a pair of nodes, with respect to a spanning tree, is the product of the communication requirement and the length of the path between the two cities. We construct constrained optimum communication trees when (i) certain specified cities are required to be the outer nodes of the communication tree, and when (ii) it is required that certain pairs of cities be connected directly in the communication tree. We further analyse changes in the structure of an optimum communication tree when the communication requirement between a pair of cities is subject to changes. We show that for the whole range $[0, \infty)$, of the communication requirement, there exist at most $(n-1)$ optimum communication trees and we construct all of them in $O(n^4)$ computational effort.

Key words. communication spanning trees, cut-tree, critical value

1.1. Introduction. Given a set N of n nodes and a set of $n(n-1)/2$ nonnegative integers $\{r_{ij}\}$, where r_{ij} is considered as the communication requirement between the ith and jth nodes, and taking the distance between every pair of nodes as unity, cost of communication over a spanning tree T is defined as

$$C(T) = \sum_{i \in N} \sum_{j \in N} r_{ij} \times (\text{number of links in the path form } i \text{ to } j \text{ in } T).$$

Hu [5] considered the problem of determining the tree for which the cost of communication is minimum and has shown that the optimum tree is a cut-tree for an undirected network G, whose vertex set is N and arc (i, j), has capacity of r_{ij} units. Gomory and Hu [4] and Schnorr [6] have given algorithms for constructing a cut-tree in polynomial time (operation count).

In this paper we develop algorithms for constructing optimum communication trees for the following cases:

(1) Constructing a tree such that it contains specified nodes in its outernodes and is of minimum communication cost among all such trees.

Some of the situations where such a need may arise are:

(i) Certain cities are the outposts and therefore cannot act as transshipment points in the tree.

(ii) In computer communication networks, input or output devices are represented as outernodes.

(iii) In military communication, these nodes represent the stations which are not privileged to receive the information from other stations.

(iv) These nodes have a very high cost of constructing a central facility and hence are not desired to be the inner nodes.

(2) Constructing a tree such that it contains a specified set of links and is of minimum communication cost among all such trees. Some of the situations where such

* Received by the editors March 18, 1981, and in revised form June 29, 1983.
† Moti Lal Nehru Regional Engineering College, Allahabad, India.
‡ Industrial and Management Engineering Department, Indian Institute of Technology, Kanpur, India.
§ Mathematics Department, Indian Institute of Technology, Kanpur, India.

a need may arise are:

(i) These links are already existing and therefore have to be used in the communication tree.

(ii) Information between a pair of cities is classified and hence a direct link between the two nodes is required as part of the communication tree.

(3) Constructing all cut-trees when the communication requirement between a specified pair of nodes varies in the interval $[0, \infty)$, Elmaghraby [3] has pointed out some interesting applications of such sensitivity analysis.

1.2. Notation and definitions.

$G[N, \{r_{ij}\}]$: network with N as the the node set and r_{ij} as the capacity of the arc (i, j).

Let T denote a spanning tree for $G[N, \{r_{ij}\}]$.

$(X_i(T), X_j(T))$: partition of N, obtained by removing link (i, j) from T, with $i \in X_i(T)$ and $j \in X_j(T)$.

$P_{ij}(T)$: path between i and j in T.

$R_{ij}(T)$: capacity of the cut $(X_i(T), X_j(T))$

$$= \sum_{p \in X_i(T)} \sum_{q \in X_j(T)} r_{pq} \text{ if } (i, j) \in T$$

$$= \min \sum_{(p,q) \in P_{ij}(T)} \{R_{pq}(T)\} \text{ if } (i, j) \notin T.$$

$C(T)$: cost of communication over T

$$= \sum_{(i,j) \in G} r_{ij} \times (\text{number of links in } P_{ij}(T))$$

$$= \sum_{(i,j) \in T} R_{ij}(T), \text{ ([5])}.$$

v_{ij}: max-flow between nodes i and j in $G[N, \{r_{ij}\}]$.

S: cut-tree for $G[N, \{r_{ij}\}]$, and

$R_{ij}(S) =$ $v_{ij}\ \forall (i, j) \in S$, hence,

$C(S) =$ $\sum_{(i,j) \in S} v_{ij}$.

1.3. Construction of optimum communication tree with specified set of nodes as subset of its outer nodes. Let $M \subset N$ denote the subset of nodes required to be the outer nodes of the optimum communication tree. In the proposed algorithm, we start with the tree S, which is an optimum communication tree without any constraints. At each iteration of the algorithm we identify a node which is not an outer node of the current tree but is required to be in M. Let it be node k. We select a node q, adjacent to k but not in M, such that the max-flow between nodes k and q is maximum among all nodes adjacent to k. Node k is then converted into an outer node of the current tree by making all nodes incident on k, except q, to be incident on q. This process is repeated till all nodes in M are outer nodes of the current tree. This algorithm will require at most $O(n)$ operation at each iteration and hence has an overall complexity of $O(n^2)$. We give a formal description of the algorithm in Pidgin Algol [2].

procedure *SCT* (n, r, M, S, V, S^*):

comment n is the number of nodes, r is an $n \times n$ array whose element r_{ij} is the communication requirement between node i and node j,

M is the list of nodes required to be the outer nodes,

S is the link list of the cut-tree for the graph $G[N, \{r_{ij}\}]$,
V is an $n \times n$ array whose element v_{ij} is the minimum cut-value between nodes i and j.

begin
$B \leftarrow$ {set of outer nodes of S};
$Q \leftarrow M \cap B$;
while $|M| \neq |Q|$ **do**
begin $k \in M - Q$;
comment k is an inner node in S, and is required to be an outer node.
$H \leftarrow$ {set of adjacent nodes of k in T};
$H1 \leftarrow \{H - (H \cap Q)\}$;
Val $\leftarrow 0$;
comment Select the node q, which is adjacent to k in S, such that it is not required to be an outer node and has maximum value of v_{kq} among all the nodes in $H1$.
for each i **in** $H1$ **do**
if val $< v_{ki}$ **then**
begin val $\leftarrow v_{ki}$; $q \leftarrow i$;
end;
comment Delete all links incident on k in S and add all these links to q to get the new tree.
for each j **in** $H - \{q\}$ **do**
$S \leftarrow (S \cup \{(j, q)\}) - \{(k, j)\}$;
$Q \leftarrow Q \cup \{k\}$;
end;
$S^* \leftarrow S$;
write S^*;
end.

We now prove that S is an optimum communication tree for the construction in § 1.3. At any iteration m, let

$T(m)$: denote the current tree,
$Q(m)$: the set of outer nodes of $T(m)$ which are also in M,
k and q: nodes of $T(m)$ as selected in the procedure *SCT*.

We make the following observation. For any link (i, j) not incident on k, such that (i, j) is a link in both $T(m+1)$ and $T(m)$,

$$R_{ij}(T(m+1)) = R_{ij}(T(m)).$$

This follows from the fact that

$$(X_i(T(m+1)), X_j(T(m+1))) = (X_i(T(m)), X_j(T(m))).$$

In Lemma 1 below, we show that for all pairs of nodes i, j, such that $i, j \notin M$, S^* gives the minimal cuts.

LEMMA 1. *For $i \notin M$ and $j \notin M$, $R_{ij}(S^*) = R_{ij}(S) = v_{ij}$.*

Proof. By induction. Let the induction hypothesis be

$$\text{For } i \notin Q(m), j \notin Q(m), R_{ij}(T(m)) = v_{ij}.$$

For $m = 1$, it is trivially true, since $T(1) = S$ and $R_{ij}(T(1)) = v_{ij}\ \forall i, j \in N$.

Let the induction hypothesis be true for the mth iteration. We show that it is true for the $(m+1)$st iteration also. Consider the following cases:

Case 1. (i, j) is a link in $T(m+1)$ as well as in $T(m)$ and is not incident on k. Then by the observation made above and the induction hypothesis,

$$R_{ij}(T(m+1)) = R_{ij}(T(m)) = v_{ij}.$$

Case 2. $i \notin Q(m+1)$, $j \notin Q(m+1)$, and path from i to j in $T(m)$ does not contain the link (k, q). Then the same path connects i and j in $T(m+1)$ also (see Fig. 1), i.e., $P_{ij}(T(m+1)) = P_{ij}(T(m))$. Therefore,

$$(1) \qquad R_{ij}(T(m+1)) = \min_{(l,p) \in P_{ij}(T(m))} \{R_{lp}(T(m+1))\}.$$

By the observation made above, and the induction hypothesis being true for the mth iteration, it follows for all the links $(l, p) \in P_{ij}(T(m))$ that

$$(2) \qquad R_{lp}(T(m+1)) = R_{lp}(T(m)) = v_{lp}.$$

From (1) and (2) it now follows that

$$R_{ij}(T(m+1)) = \min_{(l,p) \in P_{ij}(T(m))} \{v_{lp}\} = v_{ij}.$$

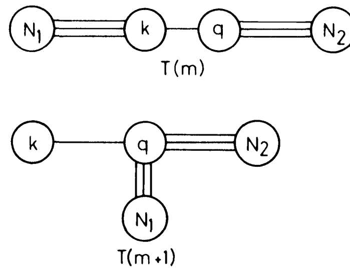


FIG. 1

Case 3. $i \notin Q(m+1)$, $j \notin Q(m+1)$ and the path from i to j in $T(m)$ contains the link (k, q). Let this path be denoted by $(i, i_l, \cdots, i_f, k, q, i_q, \cdots, i_e, j)$. Then the corresponding path in $T(m+1)$ is $(i, i_l, \cdots, i_f, q, i_q, \cdots, i_e, j)$. Since all the links on this path, except for (i_f, q), satisfy the conditions for Case 1,

(i) $R_{lp}(T(m+1)) = R_{lp}(T(m)) = v_{lp} \ \forall (l, p) \in \{P_{ij}(T(m+1)) - (i_f, q)\}$.

Also, at the mth iteration, $k \notin Q(m)$ and the induction hypothesis holds; hence,

$$R_{kq}(T(m)) = v_{kq} \quad \text{and} \quad R_{i_f k}(T(m)) = v_{i_f k}.$$

But $v_{kq} \geqq v_{i_f k}$ (from the selection criterion for q). Therefore,

(ii) $R_{kq}(T(m)) \geqq R_{i_f k}(T(m))$.

Using the configurations in Fig. 1, (ii) implies

$$(X_{i_f}(T(m)), X_k(T(m))) = (X_{i_f}(T(m+1)), X_q(T(m))),$$

i.e.

(iii) $v_{i_f k} = R_{i_f k}(T(m)) = R_{i_f q}(T(m+1))$.

We can now write

$$R_{ij}(T(m+1)) = \min\{R_{ii_l}(T(m+1)), \cdots, R_{i_fq}(T(m+1)), \cdots, R_{i_ej}(T(m+1))\}$$

$$= \min\{R_{ii_l}(T(m)), \cdots, R_{i_fk}(T(m)), \cdots, R_{i_ej}(T(m))\} \quad \text{(from (i) and (iii))}$$

$$= \min\{R_{ii_l}(T(m)), \cdots, R_{i_fk}(T(m)), R_{kq}(T(m)), \cdots, R_{i_ej}(T(m))\} \quad \text{(from (ii))}$$

$$= R_{ij}(T(m))$$

$$= V_{ij}.$$

We have thus shown that the induction hypothesis is true for the $(m+1)$st iteration if it is true for the mth iteration. Hence it holds for the last iteration. But Q (last iteration) $= M$. This proves Lemma 1.

We now prove the following theorem.

THEOREM 1. *S^* is the optimum communication tree among all trees which have all the nodes in M as its outer nodes.*

Proof. It is obvious from the procedure SCT that S^* contains M in its set of outer nodes. Proof that $C(S^*)$ is minimum among all such trees is by contradiction. Let T be a tree which has all the nodes of M as its outer nodes and for which $C(T) < C(S^*)$.

Let

$$A(T) = \{(i, j) \in T / i \in M\} \quad \text{and} \quad A(S^*) = \{(i, e) \in S^* / i \in M\}.$$

Remove all the links in $A(T)$ and $A(S^*)$ from the corresponding trees and let the trees so obtained be denoted by T_M and S^*_M respectively. For each link $(i, j) \in S^*_M$, using the mapping by Adolphson and Hu [1], we associate a unique link $([i], [j]) \in T_M$, which lies in the path from i to j in T_M. Since the link $([i], [j])$ is also in $P_{ij}(T)$,

$$R_{([i],[j])}(T) \geqq R_{ij}(T) \geqq v_{ij} = R_{ij}(S^*) \quad \text{(by Lemma 1).}$$

Hence,

$$\sum_{(i,j) \in S^*_M} R_{ij}(S^*) \leqq \sum_{(i,j) \in S^*_M} R_{([i],[j])}(T).$$

Also $\forall (i, e) \in A(S^*)$, $R_{ie}(S^*) = \sum_{k \in N} r_{ik}$ and $\forall (i, j) \in A(T)$, $R_{ij}(T) = \sum_{k \in N} r_{ik}$. Using the definition of the cost of communication over a tree, it follows that,

$$C(S^*) \leqq C(T), \quad \text{contradicting } C(T) < C(S^*).$$

Example. Consider the following network, where the numbers on the arcs are the corresponding r_{ij}. Arcs not shown are assumed to have $r_{ij} = 0$ (Fig. 2). Figure 3 gives the cut-tree for this network, and the numbers on the links represent the corresponding maximum flows. We now require that all the nodes of $M = \{3, 5\}$ should be the outer nodes. Then $Q = \{3\}$, and $M - Q = \{5\}$. Hence $H1 = \{4, 6\}$.

We determine the node q such that

$$v_{5q} = \max_{j \in H1} \{v_{5j}\} = v_{54}$$

i.e. $q = 4$, and the modified tree is obtained by making nodes 3 and 6 incident on node 4, instead of on node 5. Since 3 and 5 both are outer nodes in this tree (Fig. 4) the algorithm stops. This is the required optimum tree.

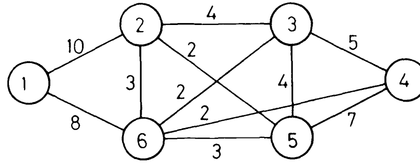


FIG. 2

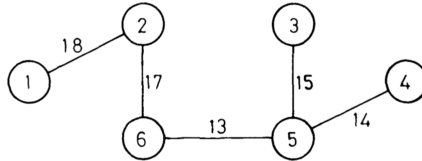


FIG. 3

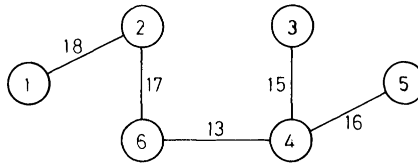


FIG. 4

1.4. Construction of an optimum communication tree having some specified links. We first prove a simple lemma and then give the algorithm *SLT*, for constructing the above tree.

LEMMA 2. *If for a pair of nodes i, j*

$$r_{ij} > \min\left\{\sum_{k\in N-\{j\}} r_{ik}, \sum_{k\in N-\{i\}} r_{kj}\right\},$$

then (i, j) *is a link in the cut-tree S.*

Proof. By contradiction. Let

$$r_{ij} > \sum_{p\in N-\{i\}} r_{pj}. \tag{3}$$

Let us assume that (i, j) is not a link in S and k is the first node on the path from i to j in S. Then $(i, j) \in (X_i(S), X_k(S))$ and since this is the minimum cut between i and k, link (i, j) is used to its full capacity in the max-flow between i and k. Thus by conservation of flow at j, it follows that

$$r_{ij} \leqq \sum_{p\in N-\{i\}} r_{pj} < r_{ij} \quad \text{(from (1))}.$$

This proves the lemma.

Algorithm *SLT* simply makes repeated use of this lemma, for constructing the optimal tree in § 1.4.

procedure *SLT* (n, r, J, S);
comment n is the number of nodes in N, r is an $n \times n$ matrix whose element r_{ij} is the communication requirement between nodes i and j,
J is the list of the links required to be in the optimal communication tree,
S^* is the tree constructed by this procedure,
$G[J]$ is the graph formed by the arcs in J.
begin
  for each $(i, j) \in J$, i is an outer node of $G[J]$ **do**

$$\bar{r}_{ij} \leftarrow \sum_{\substack{k \in N \\ k \neq j}} r_{ik} + 1;$$

  for each $(i, j) \notin J$ **do** $\bar{r}_{ij} \leftarrow r_{ij}$; compute the cut-tree S using Gomory and Hu's procedure for the network with N as its node set and $\bar{r}_{ij}$ as the capacity of arc (i, j);
  write S^*;
end.

THEOREM 2. *S^* is the optimal communication tree for $G[N, \{r_{ij}\}]$ containing all the links of J.*

Proof. For the modified communication requirements $\{\bar{r}_{ij}\}$, all the links in J satisfy the condition for Lemma 2, and hence S^* contains all the links in J. Let T be another tree which contains all the links in J and for which $C(T) < C(S^*)$. Let $\bar{R}_{ij}(S^*)$ denote the value of $(X_i(S^*), X_j(S^*))$ for the network $G[N, \{\bar{r}_{ij}\}]$. Since $\bar{r}_{ij} = r_{ij}\ \forall (i, j) \notin J$ and S^* and T contain all the links in J, it follows that for $(i, j) \in J$,

$$\bar{R}_{ij}(S^*) = R_{ij}(S^*) + \bar{r}_{ij} - r_{ij} \quad \text{and} \quad \bar{R}_{ij}(T) = R_{ij}(T) + \bar{r}_{ij} - r_{ij}.$$

Since S^* is a cut-tree for $G[N, \{\bar{r}_{ij}\}]$, it follows that for $\forall (i, j) \in J$, $R_{ij}(S^*) \leqq R_{ij}(T)$. For $(i, j) \in S^* - J$, let $([i], [j])$ be the corresponding link in T, ([1]). Then

$$R_{ij}(S^*) = \bar{R}_{ij}(S^*) \leqq \bar{R}_{ij}(T) \leqq \bar{R}_{[i][j]}(T).$$

Since $\bar{r}_{ij} = r_{ij}\ \forall (i, j) \notin J$ and all the links in J are links in T, $\bar{R}_{[i][j]}(T) = R_{[i][j]}(T)$, hence $R_{ij}(S^*) \leqq R_{([i],[j])}(T)$. We thus obtain $C(T) \geqq C(S^*)$, a contradiction. Hence S^* is the required tree.

Example. Consider the network of Fig. 2. Let

$$J = \{(2, 5), (5, 4)\}. \quad \text{Then } G[J] = ②—⑤—④.$$

Since 2 is an outer node of $G[J]$, we replace r_{25} by $\bar{r}_{25} = 1 + r_{21} + r_{23} + r_{26} = 18$ and delete the arc from $G[J]$. Next replace r_{54} by $\bar{r}_{54} = 18 + 4 + 3 + 1 = 26$. Figure 5 shows the network with modified requirements. Figure 6 gives the corresponding cut-tree, it is also the optimum tree containing links $(2, 5)$ and $(5, 4)$. Numbers corresponding to the links are the cut values w.r.t. the requirements $\{r_{ij}\}$.

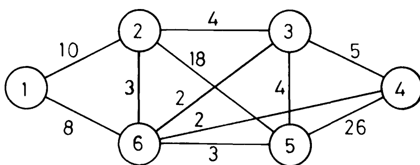


FIG. 5

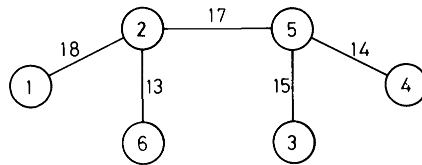


FIG. 6

2.0. Sensitivity analysis w.r.t. changes in the level of communication between a specified pair of nodes. It is evident that with any change in the capacity of an arc, the minimum cuts between the nodes of G will change, thus altering the structure of the optimum communication tree. We shall analyse such changes with respect to a pair of specified nodes p and q. Elmaghraby in [3] has pointed out some interesting applications of such sensitivity analysis.

Let λ denote the value of the communication requirement r_{pq}. Each value of λ for which the cut-tree changes when $r_{pq} \geqq \lambda$ we shall call a critical value and the interval between two consecutive critical values a critical interval. Let $[\lambda^{k-1}, \lambda^k]$ denote the kth critical interval, with $\lambda^0 = 0$. A cut-tree will remain optimum for all values of λ in this interval. Elmaghraby [3] has given an algorithm for constructing such intervals but in his case it is possible that the same spanning tree is a cut-tree for more than one interval as these intervals are not necessarily critical. Furthermore, at each iteration his algorithm requires $O(n^4)$ computational effort and the number of iterations may be more than n.

We show that there are at most $(n-1)$ critical values, and give an algorithm to construct all the cut-trees corresponding to the $(n-1)$ critical intervals using at most $O(n^4)$ operations.

2.1. Bound on the number of critical values. We make the following observations in order to arrive at an upper bound on the number of critical values for a specified pair of nodes (p, q).

(i) If $r_{pq} > \min\{\sum_{i\in N-\{p\}} r_{iq}, \sum_{i\in N-\{q\}} r_{ip}\}$, then (p, q) is a link in the cut-tree S. This is the statement of Lemma 2.

(ii) If link $(p, q) \in S(m)$, the cut-tree for the mth critical interval $[\lambda^{m-1}, \lambda^m]$, then $\lambda^m = \infty$. For any increase in the value of r_{pq} beyond λ^{m-1}, all the cuts between p and q will also go up by the same amount and the cut value of any other link in $S(m)$ will not be affected. Hence $S(m)$ will continue to be the cut-tree for all $r_{pq} \geqq \lambda^{m-1}$.

(iii) All the critical values are in the interval $[0, \min\{\sum_{i\in N-\{p\}} r_{iq}, \sum_{i\in N-\{q\}} r_{ip}\})$. This follows from the observations (i) and (ii).

We now prove that there are at most $(n-1)$ critical values.

LEMMA 3. *There are at most $(n-1)$ critical values as r_{pq} varies in the interval* $[0, \infty)$.

Proof. Let $S(k)$ denote the cut-tree for the kth critical interval $[\lambda^{k-1}, \lambda^k]$. We prove the lemma by showing that the number of links in $P_{pq}(S(k+1))$ is less than the number of links in $P_{pq}(S(k))$, i.e. $|P_{pq}(S(k+1))| < |P_{pq}(S(k))|$.

Consider a fixed value $r_{pq}^0 (< \lambda^k)$ of r_{pq} in the interval $[\lambda^{k-1}, \lambda^k]$, and let $C(S(k))$ and $C(S(k+1))$ denote the costs of communication for the corresponding trees for this value of r_{pq}.

For $r_{pq} = \lambda^k$, the increased cost on $S(k)$ is

(i) $C(S(k)) + |P_{pq}(S(k))|(\lambda^k - r_{pq}^0)$,

and on $S(k+1)$,

(ii) $C(S(k+1)) + |P_{pq}(S(k+1))|(\lambda^k - r_{pq}^0)$.

Since $r_{pq}^0 < \lambda^k$ and $S(k)$ is the cut-tree for the critical interval $[\lambda^{k-1}, \lambda^k]$, $C(S(k)) < C(S(k+1))$. Since, for $r_{pq} = \lambda^k$, $S(k+1)$ is also the alternate cut-tree, the costs in (i) and (ii) are equal.

Comparing the expressions in (i) and (ii), it follows that

$$|P_{pq}(S(k))| > |P_{pq}(S(k+1))|.$$

Now, for the first critical interval, $[0, \lambda^1]$,

$$|P_{pq}(S(1))| \leqq n-1.$$

Thus there can be at most $(n-1)$ critical values before the path between nodes p and q reduces to the link (p, q). In the following section we present an algorithm which will identify all the critical values and will also construct a cut-tree for each critical interval.

2.2. Algorithm for constructing all the critical values. We begin by putting $r_{pq}=0$ and constructing the corresponding cut-tree. Values of r_{pq} is progressively increased to identify its critical values. At each critical value, there exists at least one link on the path from p to q, for which the incident pair of nodes have an alternate minimum cut whose value is not dependent on the value of r_{pq}. We identify a consecutive set of such links on the path from p to q, and collect all the nodes incident on these links in a set B. To obtain these alternate minimum cuts for each pair of nodes in B, we work with a reduced network. Each component of the current cut-tree which is incident on a node of B is shrunk into a single node and the communication requirement between this node and any other node i is the sum of the communication requirement between i and all the nodes of this component. Using Gomory and Hu's procedure, nodes in B are connected by links which form part of the links of the new tree. Shrunk nodes are expanded by adding the corresponding components to the new tree. For the current value of r_{pq}, the new tree is an alternate cut-tree, such that all the links incident on both the nodes of B have been removed from the path from p to q, and number of links in this path has also been reduced by $|B|-1$ links. This procedure requires solving at most $|B|-1$ max-flow problems. We repeat this procedure, until no link is a candidate for being removed from the path from p to q. Value of r_{pq} is further increased and the whole procedure repeated again, until (p, q) becomes a link in the cut-tree.

It can be immediately observed that obtaining all the critical values will involve solving at most $|P_{pq}(T1)|$ max-flow problems, (i.e. equal to the number of links in the path from p to q in $T1$), and hence the complexity of the algorithm is $O(n^4)$, i.e., same as that of constructing a cut-tree on a network with n nodes. Below we give a formal description of the algorithm. For keeping the description short, we omit details of labelling and record keeping etc. Before describing the main procedure, we describe two other procedures which are required for executing the main procedure.

procedure SHRINK $(m, n, T, \bar{N}, C, r)$:
comment This procedure shrinks the component of the tree T, which is rooted at node m, and which is obtained by removing link (m, n), into node m,
$\bar{N}$ is the node list,
C is the matrix whose element C_{ij}, $i, j \in \bar{N}$, is the capacity of arc (i, j),
r is the communication requirement matrix.
begin
  $CN \leftarrow$ {set of nodes of the component rooted at m and obtained by removing link (m, n) from T};
  comment m does not belong to CN.
  $\bar{N} \leftarrow \bar{N} - CN$;
  for each i **in** $\bar{N}$ **do**
    begin $C_{im} \leftarrow \sum_{k \in CN \cup \{m\}} r_{ki}$;
      for each $j \in \bar{N}-\{m\}$ **do** $C_{ij} \leftarrow r_{ij}$;
    end;
end.

procedure CUT-TREE (Y, X, C, T, V):
comment Y and X are the subsets of the node set of the network for which a cut-tree is to be constructed,
C is the matrix whose element C_{ij} is the capacity of arc (i, j), $i, j \in XUY$,
T is the cut-tree constructed by this procedure,
V is the matrix of max-flows for the given network,
(X_i, X_j) denotes the minimum cut between nodes i and j.
begin
Find minimum cuts between the nodes of X with respect to the network whose node set is XUY and C is the matrix of its arc capacities;
Use Gomory and Hu's procedure to connect nodes in X by links which form part of the link set of T. Remaining links of T are obtained without solving any max-flow problems;
for each i **in** X **do**
begin
if $|X_i \cap X| = 1$ **then** $T \leftarrow T \cup_{j \in Y \cap X_i} (i, j)$;
for each node k **in** T, such that (i, k) is a link in T, **do**
$v_{ik} \leftarrow \sum_{m \in X_i, n \in X_k} C_{mn}$;
end;
for $1 \leqq i \leqq n, 1 \leqq j \leqq n$ **do**
if (i, j) is not a link in T, **then**
$v_{ij} \leftarrow \min_{(k,1)} \{v_{k1}\}$; is a link on the path between i to j in T;
end.

We now begin the procedure for computing all the critical values and constructing the corresponding cut-trees.

procedure CRITICAL VALUES (n, p, q, r, LAMBDA, N, SC):
comment n is the number of nodes, p, q is the pair of nodes with respect to which sensitivity analysis is done,
r is the communication requirement matrix,
LAMBDA is an array whose element LAMBDA$[k]$ is the kth critical value,
SC is an array, where $SC[k]$ gives the link list of the cut-tree for the kth critical value, N is the node list.
begin
$IR \leftarrow \min (\sum_{i \in N-\{p\}} r_{ip}, \sum_{i \in N-\{q\}} r_{iq})$;
$r_{pq} \leftarrow IR$; $Y \leftarrow \{\ \}$; $X \leftarrow \{1, 2, \cdots, n\}$;
CUT-TREE ($Y, X, r, T2, V2$);
comment cut-tree $T2$ has the property that (p, q) is a link in $T2$, and hence the max-flow between any other pair of nodes is independent of the value of r_{pq}.
$r_{pq} \leftarrow 0$;
CUT-TREE ($Y, X, r, T1, V$);
$CT \leftarrow T1$; $k \leftarrow 1$;
comment CT is the current cut-tree.
while number of links in the path from p to $q > 1$ **do**
begin
comment Identify the next critical value by increasing the value of r_{pq}.
$d \leftarrow IR$;
for each (i, j) **in** the path from p to q, **do**
begin

```
        d_ij ← v2_ij − v_ij; d ← min {d, d_ij};
        v_ij ← v_ij + d;
    end;
  r_pq ← r_pq + d; LAMBDA [k] ← r_pq;
  for each subpath P in the path between p and q in CT, such that for all
  links (i, j) in P, d_ij = d, i.e. for all links in P, the max-flow between the pair
  of nodes incident on these links will no longer be dependent on the value
  of r_pq, do
  begin
    a ← the first node in P;
    b ← the last node in P;
    if (a = p) ∧ (b = q) then
    begin SC[k] ← T2;
      write (LAMBDA [k], SC[k], V);
    end;
    N̄ ← N;
    else begin
      for each node i in P do
        begin
          NB_i ← {set of nodes adjacent to i in CT, except those in the path
                  from p to q};
          for each j in NB_i do
          SHRINK (j, i, CT, N̄, C, r);
        end;
      p(a) ← node preceding a in the path from p to q in CT;
      s(b) ← node succeeding b in the path from p to q in CT;
      r̄ ← C;
      if a ≠ p then SHRINK (p(a), a, CT, N̄, C, r̄);
      r̄ ← C;
      if b ≠ q then SHRINK (s(b), b, CT, N̄, C, r̄);
      B ← {nodes in P};
      comment We now define a condensed network Ḡ with N̄ as its node
      set and C as the matrix of its arc capacities. It is understood that if
      some C_ij = 0, then the corresponding arc is not in the network. We
      construct a cut-tree for Ḡ, such that for each pair of nodes in B, the
      nodes p and q are in the same partition of the minimum cut. To ensure
      this we set C_pq equal to a large number.
      C_pq ← IR + 1; Y ← N̄ − B;
      CUT-TREE (Y, B, C, T, V);
      for each i in Y do
        begin
          comment Expand the node i into the component shrunk into
          node i.
          T ← T ∪ {set of links of the component of CT rooted at i};
          N̄ ← N̄ ∪ {set of nodes of the component of CT rooted at i};
          for each link (i, j) in {set of links of the component of CT rooted
                                  at i} do
          v_ij ← v_ij; CT ← T;
        end;
```

```
        end;
          SC[k] ← {links of CT};
          write (LAMBDA [k], SC[k]);
          k ← k+1;
        end;
      end;
end.
```

To show that two consecutive values of the array LAMBDA form a critical interval and the corresponding links in the array SC, the cut-tree for that critical interval, no formal proof is necessary. It is enough to say that for any cut-tree only for the pair of nodes on the path between p and q, the cut values are dependent on the value of r_{pq}. Also, when the value of r_{pq} is increased until at least for one pair of nodes the cut value becomes equal to its cut value with respect to the tree $T2$, it is necessary to obtain for this pair an alternate minimum cut, which has both the nodes p and q in the same partition. That is, it is necessary to remove this link from the path from p to q and hence change the structure of the cut-tree. In the above procedure we simply obtain the alternate minimum cuts for all pairs of nodes which should no longer be on the path from p to q and keep all the other minimum cuts intact. Hence the resulting tree is a cut-tree for the next critical interval. We illustrate the various procedures of this algorithm by solving an example in detail.

Example. Consider the network given in Fig. 7 with communication requirements between nodes indicated along the corresponding arcs. Suppose we want to compute all the critical values of r_{25}.

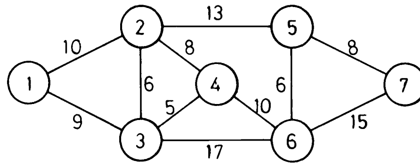


FIG. 7

$r_{25}=0$, cut-tree $T1$.

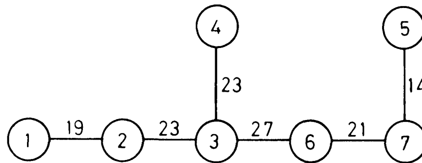


FIG. 8

cut-tree $T2$, $r_{25}=IR=14$.

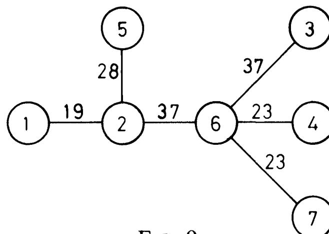


FIG. 9

Iteration 1. Path from 2 to $5=\{(2,3),(3,6),(6,7),(7,5)\}$. $IR=14$, $d=2$, $r_{25}=2$, $B=\{6,7\}$, $a=6$, $b=7$ and $Y=\{3,5\}$. The component rooted at node 3 is shrunk into node 3, and the condensed graph with the corresponding arc capacities is shown in Fig. 10.

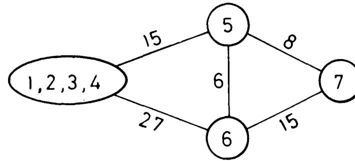


FIG. 10

Minimum cut between nodes 6 and 7 is the cut $(\{3,5,6\},\{7\})$, which is obtained by solving a max-flow problem on the network in Fig. 10, and (6, 7) is a link in the new cut-tree with $v_{67}=23$.

Since nodes 5 and $3\in Y\cap X_6$ and $|X_6\cap B|=1$, we connect nodes 3 and 5 with 6 and expand the component shrunk into node 3, according to the cut-tree in Fig. 8. The alternate cut-tree for $r_{25}=2$ is shown in Fig. 11, with the max-flows between the nodes indicated along the links, and link (6, 7) out of the path from 2 to 5, and LAMBDA [1] = 2.

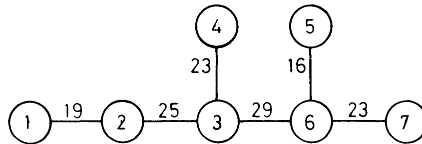


FIG. 11

Iteration 2. $d=8$, $r_{25}=10$, $B=\{3,6\}$ and $Y=\{2,4,5,7\}$. The condensed graph with arc capacities is shown in Fig. 12. To obtain a minimum out between nodes 3 and 6 such that both the nodes 2 and 5 are in the same partition of the cut, we solve a max-flow problem on the network in Fig. 12. The cut obtained is $(\{3\},\{2,4,5,6,7\})$. Thus (3, 6) is a link in the new cut-tree with $v_{36}=37$.

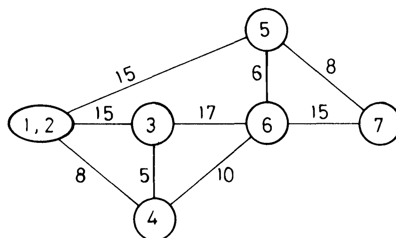


FIG. 12

Since nodes $2,4,6,7\in Y\cap X_6$ and $|X_6\cap B|=1$, we connect nodes 2, 4, 7 with node 6 and expand the component shrunk into node 2. The alternate cut-tree for $r_{25}=10$, i.e. for LAMBDA [2] = 10 is given in Fig. 13 with the max-flows between the nodes indicated along the links.

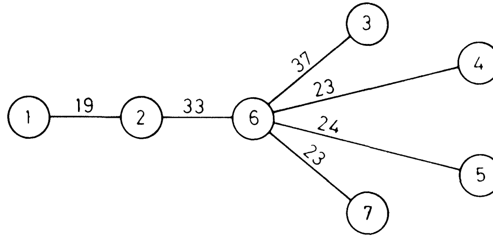


FIG. 13

Iteration 3. $d=4$, $r_{25}=14$ and $B=\{2, 6, 5\}$, $a=2$, $b=5$. since $a=p$ and $b=q$, no more max-flows to be solved and the cut-tree for $r_{25}=14$ is the cut-tree $T2$, LAMBDA [3] = 14, and no more critical values for r_{25}. Thus the four critical intervals are $[0, 2]$, $[2, 10]$, $[10, 14]$ and $[14, \infty)$.

REFERENCES

[1] D. ADOLPHSON AND T. C. HU, *Optimal linear ordering*, SIAM J. Appl. Math., 25 (1973), pp. 403–423.
[2] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
[3] S. E. ELMAGHRABY, *Sensitivity analysis of multiterminal flow networks*, Oper. Res., 12 (1964), pp. 680–688.
[4] R. E. GOMORY AND T. C. HU, *Multi-terminal network flows*, SIAM J. Appl. Math., 9 (1961), pp. 551–570.
[5] T. C. HU, *Optimal communication spanning tree*, this Journal, 3 (1974), pp. 189–195.
[6] C. P. SCHNORR, *Bottlenecks and connectivity in unsymmetrical networks*, Technical report (Oct. 1977), Fachbereich Mathematik, Universität Frankfurt.

SIAM J. COMPUT.
Vol. 13, No. 2, May 1984

© 1984 Society for Industrial and Applied Mathematics
008

IMMUNITY, RELATIVIZATIONS, AND NONDETERMINISM*

UWE SCHÖNING† AND RONALD V. BOOK‡

Abstract. It is known [8], [13] that there is a recursive set A such that NP (A) contains a set that is P (A)-immune; that is, there is an infinite set $L \in \mathrm{NP}(A)$ such that no infinite subset of L is in P (A). The first result generalizes this fact to situations where the running times of the machines specifying the "larger" class bound the size of the sets of strings queried in the computation trees of the machines specifying the "smaller" class. The second result is of a different type. For relativized complexity classes specified by bounds on the number of oracle queries and the number of nondeterministic steps allowed in computations, it is known [14] that one can describe a set A such that there is an infinite hierarchy of classes relative to A where each new class in the hierarchy is obtained by increasing the amount of nondeterminism. Here it is shown that the conditions allowing infinite hierarchies to exist also allow for each $i \geqq 0$, the existence of a set in the $(i+1)$st class which has no infinite subset in the ith class.

Key words. complexity classes, relativizations, nondeterminism, bounded queries, immunity, P, NP

1. Introduction. Does every infinite set in NP have an infinite subset in P? An infinite set with no infinite subset in P is called P-*immune* [12].

Bennett and Gill [4] showed the existence of a set A such that NP (A) has a P (A)-immune set, that is, an infinite set in NP (A) with no infinite subset in P (A). Recently, Schöning [13] and, independently, Homer and Maass [8] showed that the set A can be taken to be recursive.

The proof technique used in [13] can be applied in a variety of other circumstances. The purpose of this paper is to establish two general settings in which this technique can be used to establish "immunity" results. The First Immunity Theorem generalizes the result noted above by giving conditions on a set **F** of running times such that there exists a set A and a set L in NTIME $(\mathbf{F}, A)$ that is immune with respect to the class of languages accepted relative to A by any class of oracle machines such that the set of strings queried in all computations on an input is bounded in size by a function in **F**.

Baker, Gill, and Solovay [1] showed that there exists a set A such that $\mathrm{P}(A) \neq \mathrm{NP}(A)$. Kintala and Fischer [10], [11] showed that the construction of the oracle set could be "spread" to show the existence of an infinite hierarchy $\mathrm{P}(A) \subsetneq \mathrm{P}(A)_{\log^2 n} \subsetneq \mathrm{P}(A)_{\log^3 n} \subsetneq \cdots \subsetneq \mathrm{P}(A)_n$ and an infinite hierarchy $\mathrm{P}(B) \subsetneq \mathrm{P}(B)_n \subsetneq \mathrm{P}(B)_{n^2} \subsetneq \cdots \subsetneq \mathrm{NP}(B)$ where the function in the subscript indicates a bound on the number of nondeterministic steps in any computation. Thus, one can "refine" the use of nondeterminism in relativized computation to obtain infinite hierarchies.

The proofs of the separation theorems in [1] and [10], [11] do not depend on time as such. Rather time bounds serve to bound the number of nondeterministic steps allowed in any computation and also to bound the number of oracle queries allowed in any computation. Consider classes of oracle machines such that it is these two parameters that are bounded during computations. In this context Xu, Doner, and Book establish in [14] a very general theorem which allows one to describe infinite hierarchies of relativized complexity classes by allowing the amount of nondeterminism

* Received by the editors October 27, 1982, and in revised form July 12, 1983. This research was supported in part by the National Science Foundation under grants MCS80-11979 and MCS83-12472 and by the Deutsche Forschungsgemeinschaft.

† Institut für Informatik, Universität Stuttgart, 7000 Stuttgart 1, West Germany.
‡ Department of Mathematics, University of California, Santa Barbara, California 93106.

used to grow at each level. The hypotheses specify how the nondeterminism can grow and how the amount of nondeterminism is related to the bound on the number of oracle queries.

In the present paper we extend the immunity result cited above to the situation studied by Xu, Doner, and Book. The Second Immunity Theorem shows the existence of an oracle set such that at each level of an infinite hierarchy there is a set that is immune with respect to the previous level. Again, the parameters involve only the amount of nondeterminism and the number of oracle queries allowed in any computation, and the hypotheses specify the conditions on these parameters. The Second Immunity Theorem is applicable to a wide variety of classes. Just five examples are given here.

One may wish to interpret the value of these general theorems, and there is one interpretation that arises immediately. These results add evidence to the thesis that separation theorems about relativized classes say little about the difference between determinism and nondeterminism in ordinary computation but rather illustrate the power of nondeterminism to generate a large set of strings to be queried. This is strongly emphasized here since the Second Immunity Theorem is established in the context of limited nondeterminism and not just determinism vs. full nondeterminism.

The study of immunity with respect to complexity classes was initiated by Flajolet and Steyaert [7]. The notion of "simplicity" is essentially dual to that of immunity. Balcázar [2] has established results on simplicity that parallel the results of this paper. The notion of both a set and its complement being **C**-immune for a complexity class **C** has been studied by Balcázar and Schöning [3].

2. The basic proof. The Immunity Theorems are essentially invariant under a variety of changes in the model of computation with the exception that inputs are assumed to be strings and the oracle queries are of the form "is the string on the query tape in the oracle set?" However, the examples given are in terms of classes specified by various types of restricted oracle Turing machines.

An *oracle machine* is a multitape Turing machine M with a distinguished work tape, the *query* tape, and three distinguished states, QUERY, YES, and NO. At some step of a computation on an input string w, M may transfer into the state QUERY. In state QUERY, M transfers into the state YES if the string currently appearing on the query tape is in an *oracle set* A; otherwise, M transfers into the state NO; in either case the query tape is instantly erased. The set of strings *accepted by M relative to the oracle set* A is $L(M, A) = \{w \mid \text{there is an accepting computation of } M \text{ on input } w \text{ when the oracle set is } A\}$.

Oracle machines may be deterministic or nondeterministic. An oracle machine may operate within some time bound T, where T is a function of the length of the input string, and the notion of operation within a time bound for an oracle machine is just the same as that notion for an ordinary Turing machine. An oracle machine may operate within some space bound S, where S is a function of the length of the input string, and here we require that the query tape as well as the ordinary work tapes be bounded in length by S.

We assume that every machine has nondeterministic fan-out at most two.

We assume that the reader is familiar with the elements of machine-based complexity theory at the level of a textbook such as [9] and with relativized complexity classes such as P (A), NP (A), PSPACE (A), etc.

For a string w, $|w|$ denotes the length of w.

The formal definition of "immunity" is made in a general context.

DEFINITION 1. Let $\mathbf{C}$ be a (possibly, relativized) complexity class. A set L is $\mathbf{C}$-*immune* if L is infinite and no infinite subset of L is in $\mathbf{C}$.

Now we have the first result.

PROPOSITION. *There exists a recursive set A such that* NP (A) *has a set that is* P (A)*-immune.*

Proof. We construct a recursive set A such that $L(A)=\{0^m|\text{there exists a word of length } m \text{ in } A\}$ does not have an infinite subset in P (A). Clearly, $L(A)$ is in NP (A).

Let $P_0, P_1, \cdots$, be an effective enumeration of the deterministic oracle machines that run in polynomial time. For each i, let q_i be a polynomial that bounds the running time of P_i. The set A will be constructed in stages so that for each m at most one string of length m will be put into A. For each $n \geqq 0$, A_n is the set of strings put into A at stages $0, \cdots, n$, and R_n is the set of indices of machines P_i, $i \leqq n$, that are candidates for diagonalization at stage $n+1$.

Stage 0.
$A_0 := \varnothing$.
$\mu(0) := 0$.
$R_0 := \{0\}$
Stage n $(n \geqq 1)$.
Let $\mu(n)$ be the least integer k such that the following conditions hold:
(i) $2^k > \sum_{i \leqq n} q_i(k)$;
(ii) $k > \max\{q_i(\mu(n-1)) | i < n\}$.
Search for the smallest $j \in R_{n-1}$ such that $0^{\mu(n)} \in L(P_j, A_{n-1})$.

Case 1. No such j is found. Search for a string w, $|w| = \mu(n)$, such that no machine P_i, $i \leqq n$, on input $0^{\mu(n)}$ queries its oracle for "$w \in A_{n-1}$?" Such a string exists since on input $0^{\mu(n)}$, P_i can query its oracle at most $q_i(\mu(n))$ times, the number of strings in $\{0,1\}^*$ of length $\mu(n)$ is $2^{\mu(n)}$, and $\mu(n)$ was chosen so that condition (i) is satisfied.

Let w be the least such string in some effective enumeration of strings and let $A_n := A_{n-1} \cup \{w\}$. Let $R_n := R_{n-1} \cup \{n\}$.

Case 2. Some such j is found. Let $R_n := (R_{n-1} - \{j\}) \cup \{n\}$ and let $A_n := A_{n-1}$.

End of construction.

Define A as $A := \bigcup_{n \geqq 0} A_n$.

Notice that condition (ii) implies that k is larger than the length of any string queried at any previous stage. Thus, for any $n > 0$ the behavior of machines P_i, $i \leqq n$, on $0^{\mu(n)}$ relative to A_{n-1} is precisely the same as their behavior on $O^{\mu(n)}$ relative to A_{n+j} for any $j \geqq 0$ or relative to A.

Clearly, $L(A)$ is infinite if and only if A is infinite. Suppose that A is finite. Then there exists n_0 such that at each stage $n \geqq n_0$, Case 2 occurs so that an index is cancelled from R_n. Thus, at each stage $n \geqq n_0$, the size of R_n remains constant, the size of R_{n_0}. But there are infinitely many i with the property that for all sets B, $L(P_i, B) = \varnothing$, so that no index in this set is cancelled from R_n in an occurrence of Case 2. Hence, A must be infinite.

Suppose there exists an infinite subset C of $L(A)$ such that for some j, $C = L(P_j, A)$. Since C is infinite, $C \subseteq L(A)$, and $L(A) \subseteq \{0^{\mu(n)} | n \geqq 0\}$, there are infinitely many n such that $0^{\mu(k)} \in C = L(P_j, A)$. The index j was put into R_j at the end of stage j. Since there are only finitely many indices less than j, there must be a stage $k > j$ such that Case 2 with index j occurs so that $0^{\mu(k)} \in L(P_j, A_{k-1})$ and, hence, $0^{\mu(k)} \in L(P_j, A) = C$. But then A_k is chosen so that $0^{\mu(k)} \notin L(A_k)$, and, hence, $0^{\mu(k)} \notin L(A)$ since no string of length $\mu(k)$ is put into A at a stage greater than k because condition (ii) is satisfied when choosing $\mu(n)$. Thus, $0^{\mu(k)} \notin C$ since $C \subseteq L(A)$, contradicting $0^{\mu(k)} \in C$.

Hence, $L(A)$ is P (A)-immune. □

The existence of a set A such that NP (A) has a P (A)-immune set was first shown by Bennett and Gill [4]. That the set A can be taken to be recursive was shown by Homer and Maass [8] and, independently, by Schöning [13]. The proof of the proposition is from [13] and serves as the outline of the proofs of our more general results.

For any set D, let NPB (D) be the class of languages L such that $L \in \mathrm{NP}(D)$ is witnessed by a machine M with the property that there is a polynomial q such that for all inputs x there are at most $q(|x|)$ accepting computations of M on x (relative to any oracle). One can trivially alter the proof of the proposition to show that there is a set A such that $L(A)$ is NPB $(A) \cap$ co-NPB (A)-immune.

3. The First Immunity Theorem. The First Immunity Theorem is a straightforward generalization of the proposition to settings other than deterministic and nondeterministic polynomial time. It illustrates how nondeterminism is used to generate more strings than a deterministic machine can query when the number of strings that the deterministic machine can query is restricted.

DEFINITION 2. Let M be an oracle machine. For each set B and each input string x of M, let $Q(M, B, x)$ be the set of strings y such that in some computation of M relative to B on input x, the oracle is queried about y. Let $\|Q(M, B, x)\|$ be the cardinality of $Q(M, B, x)$.

Consider the proof of the proposition. Since each P_i is deterministic and runs in time q_i, for every set B and input string x, $\|Q(P_i, B, x)\| \leqq q_i(|x|)$. Thus, condition (i) implies that $2^{|x|} > \sum_{i \leqq n} \|Q(P_i, B, x)\|$ for all sets B and x, and there are $2^{|x|}$ strings in $\{0, 1\}^*$ of length $|x|$. Hypotheses forcing this to happen allow one to consider more general classes of oracle machines. This leads to the First Immunity Theorem.

FIRST IMMUNITY THEOREM. *Let* $\mathbf{M} = \{M_i | i \geqq 0\}$ *be a class of nondeterministic oracle machines, and let* $\mathbf{F}$ *be a class of nondecreasing functions (on the natural numbers) that are running times. Suppose that the following conditions hold:*

(i) *for each* i *there exists* $f \in \mathbf{F}$ *such that for every set* B *and every string* x, $\|Q(M_i, B, x)\| \leqq f(|x|)$;

(ii) *for each* $f \in \mathbf{F}$ *and integer* $c \geqq 0$, *there exists* $g \in \mathbf{F}$ *such that for all but finitely many* n, $cf(n) \leqq g(n)$;

(iii) *for every* f *and* g *in* $\mathbf{F}$, $\log f = o(g)$;

(iv) *for infinitely many* i *and all sets* B, $L(M_i, B) = \varnothing$.

Then there exists a set A *and a set* L *in* NTIME $(\mathbf{F}, A)$ *such that* L *is* $\{L(M_i, A) | i \geqq 0\}$*-immune.*

Sketch of the proof. Let t be in $\mathbf{F}$. For each set B, define $L(B, t) = \{0^m |$ there exists $w \in B$ such that $|w| = t(m)\}$. Clearly, for every set B and every t in $\mathbf{F}$, $L(B, t)$ is in NTIME $(\mathbf{F}, B)$.

For each function t in $\mathbf{F}$, construct a set A in stages so that $L(A, t)$ is $\{L(M_i, A) | i \geqq 0\}$-immune. For each m at most one string of length m will be put into A.

For each i let f_i be any function in $\mathbf{F}$ such that for every set B and every x, $\|Q(M_i, B, x)\| \leqq f_i(|x|)$. By condition (i), for each i such an f_i exists.

Now proceed as in the proof of the proposition where condition (a) below replaces condition (i) in that proof and condition (b) replaces condition (ii):

(a) $2^{t(k)} > \sum_{i \leqq n} f_i(k)$;

(b) $k > \max\{|y| \,|$ for some $i < n$, some computation of M_i relative to A_{n-1} on input $0^{\mu(n-1)}$ queries the oracle about $y\}$.

The details are left to the reader. □

Consider the following applications of the First Immunity Theorem.

1. Let **M** be the class of deterministic oracle machines that run in exponential (i.e., 2^{cn}) time. Let **F** be the set of functions $\{2^{in} | i \geqq 1\}$. For each set B, denote $\{L(M_i, B) | i \geqq 0\}$ by DEXT (B) and NTIME $(\mathbf{F}, B)$ by NEXT (B). Then there exists a set A and a set L in NEXT (A) that is DEXT (A)-immune.

2. Let **M** be the class of deterministic oracle machines that use polynomial work space and are restricted so that only a polynomial number of oracle queries are allowed in any computation. For each set B, denote $\{L(M_i, B) | i \geqq 0\}$ by PQUERY (B); see [5], [14] for properties of classes of this form. Let **F** be the set of polynomials, so that for any set B the class NTIME (F, B) is NP (B). Then there exists a set A and a set L in NP (A) such that L is PQUERY (A)-immune.

For other examples of settings where the First Immunity Theorem is applicable, see [6], [14].

4. The Second Immunity Theorem. Now we establish the definitions that enable us to formulate the Second Immunity Theorem.

DEFINITION 3. A machine M *operates in nondeterminism* $g(n)$ if for every input string x to M, any computation of M on x has at most $g(|x|)$ nondeterministic steps.

DEFINITION 4. Let **M** be a set of nondeterministic oracle machines. Let **T** be a set of nondecreasing functions with the property that for each $M \in \mathbf{M}$, there is a function $t \in \mathbf{T}$ such that for every n, in any computation on an input of length n, M can query its oracle at most $t(n)$ times, and, conversely, for every $t \in \mathbf{T}$, there is an $M \in \mathbf{M}$ satisfying this condition. Let **G** be a set of nondecreasing functions with the property that for each $M \in \mathbf{M}$, there is a function $g \in \mathbf{G}$ such that M operates in nondeterminism g, and, conversely, for every $g \in \mathbf{G}$, there is an $M \in \mathbf{M}$ satisfying this condition. Assume that each of **M**, **T**, and **G** are countably infinite, and that there are infinitely many deterministic $M \in \mathbf{M}$ such that for every set A, $L(M, A) = \emptyset$. Any such triple $\langle \mathbf{M}, \mathbf{T}, \mathbf{G} \rangle$ will be called a *proper oracle machine class.*

DEFINITION 5. Let $\langle \mathbf{M}, \mathbf{T}, \mathbf{G} \rangle$ be a proper oracle machine class. For any set A and any $g \in \mathbf{G}$, define $\mathrm{D}(\mathbf{M}, A)_g = \{L(M, A) | M \in \mathbf{M}$ operates in nondeterminism $g\}$. For any set A, define $\mathrm{D}(\mathbf{M}, A)_0 = \{L(M, A) | M \in \mathbf{M}$ operates deterministically$\}$ and define $\mathrm{D}(\mathbf{M}, A)_\infty = \{L(M, A) | M \in \mathbf{M}\}$. Now we can state our result.

SECOND IMMUNITY THEOREM. *Let* $\langle \mathbf{M}, \mathbf{G}, \mathbf{T} \rangle$ *be a proper oracle machine class. Suppose that* $\mathbf{G} = \{g[n] | n > 0\}$ *and* $\mathbf{T} = \{t[n] | n > 0\}$ *have the following properties*:

(i) $\log n \leqq g[1](n)$ *for all but finitely many* n;

(ii) $i < j$ *implies* $g[i] = o(g[j])$;

(iii) *for every* $t \in \mathbf{T}$, $\log t = o(g[1])$;

(iv) *for every integer* $i > 0$ *and every set* X, *the set* $\{0^p |$ *there exists* $w \in X$ *such that* $|w| = g[i](p)\}$ *is in* $\mathrm{D}(\mathbf{M}, X)_{g[i]}$.

Then there exists a set A *with the property that for every* i, j *with* $0 \leqq i < j \leqq \infty$, *there is a set in* $\mathrm{D}(\mathbf{M}, A)_{g[j]}$ *that is* $\mathrm{D}(\mathbf{M}, A)_{g[i]}$*-immune.*

The statement of the theorem is very technical, so before giving the proof, we turn to some applications.

1. Let $\mathbf{M}_1$ be the collection of clocked nondeterministic oracle machines that run in polynomial time. Let $\mathbf{M}_2$ be the collection of oracle machines that are obtained from $\mathbf{M}_1$ be adding clocks that bound the amount of nondeterminism allowed, say $g[i](n) = n^i$ for all $n \geqq 0$, $i \geqq 0$. Thus, $\mathrm{D}(\mathbf{M}_1, \emptyset)_0 = \mathrm{D}(\mathbf{M}_2, \emptyset)_0 = \mathrm{P}$ and $\mathrm{D}(\mathbf{M}_1, \emptyset)_\infty = \mathrm{D}(\mathbf{M}_2, \emptyset)_\infty = \mathrm{NP}$, and for each i, $\mathrm{D}(\mathbf{M}_1, \emptyset)_{g[i]}$ is the class of languages accepted by polynomial time machines that are allowed to make at most $g[i](n) = n^i$ nondeterministic steps in any computation on an input of length n. The theorem states that there exists a set A such that for every i, j with $0 \leqq i < j \leqq \infty$, there is a set L in $\mathrm{D}(\mathbf{M}, A)_{g[j]}$

that is $D(\mathbf{M}, A)_{g[i]}$-immune. In the notation of [10], [11], L is in $P(A)_{n^j}$ and L is $P(A)_{n^i}$-immune. When $i=0$ and $j=\infty$, L is in NP(A) and is P(A)-immune. If the machines in $\mathbf{M}_2$ are enumerated so that M_i runs in time $t[i]$, a polynomial, then the function $t[i]$ also serves as a bound on the number of oracle queries that M_i can make. Thus, the existence of clocks for the $t[i]$ and the $g[j]$ allows one to claim that the set A is recursive.

2. As in 1, consider oracle machines that run in polynomial time but now restrict the nondeterminism by means of the functions $g[i](n)=(\log n)^{i+1}$. It is known that there is a set A such that for all i, $P(A)_{\log^i n} \subsetneq P(A)_{\log^{i+1} n}$ [10], [11]. The theorem states that there is a set A such that for every i, j with $1 \leqq i < j \leqq \infty$, there is a set L in $P(A)_{\log^j n}$ that is $P(A)_{\log^i n}$-immune. As in 1, A can be chosen so as to be recursive.

3. Let $\mathbf{M}_1$ be the collection of clocked nondeterministic oracle machines that operate in polynomial space. Let both $\mathbf{T}$ and $\mathbf{G}$ be the collection of polynomials $\{n^k \mid k \geqq 0 \text{ and integer}\}$. Let $\mathbf{M}_2$ be the collection of clocked oracle machines obtained from $\mathbf{M}_1$ by adding clocks from $\mathbf{G}$ that bound the amount of nondeterminism and clocks from $\mathbf{T}$ that bound the number of oracle queries allowed in any computation. Thus, $D(\mathbf{M}_1, \varnothing)_0 = D(\mathbf{M}_1, \varnothing)_\infty = D(\mathbf{M}_2, \varnothing)_0 = D(\mathbf{M}_2, \varnothing)_\infty =$ PSPACE and for every set A, $D(\mathbf{M}_2, A)_0 =$ PQUERY(A) and $D(\mathbf{M}_2, A)_\infty =$ NPQUERY(A) (see [5]). Extending the notation of [10], [11] to classes PQUERY (?), if $g[i]=n^i$, then for every set A, $D(\mathbf{M}_2, A)_{g[i]} = \text{PQUERY}(A)_{n^i}$ is the class of languages accepted relative to A by polynomial space-bounded oracle machines that make at most a polynomial number of oracle queries in any computation and that operate in nondeterminism $g[i]=n^i$. It is known that there is a set A such that PQUERY$(A) \neq$ NPQUERY(A) [5] and that there is a set B such that for all i, j with $0 \leqq i < j \leqq \infty$, $\text{PQUERY}(B)_{n^i} \subsetneq \text{PQUERY}(B)_{n^j}$ [14]. The theorem states that there exists a set A such that for every i, j with $0 \leqq i < j \leqq \infty$, there is a set L in $D(\mathbf{M}_2, A)_{g[j]}$ that is $D(\mathbf{M}_2, A)_{g[i]}$-immune, that is, L is in $\text{PQUERY}(A)_{n^j}$ and L is $\text{PQUERY}(A)_{n^i}$-immune. As in 1, the set A can be chosen to be recursive.

4. Extend the first example to clocked oracle machines that run in time 2^{kn}, $k>0$ an integer, and bound the amount of nondeterminism allowed by the functions $g[i](n)= 2^{in}$, $i>0$. The theorem states that there exists a set A such that relative to A there is an infinite hierarchy with the property that at each level there is a set that is immune for the previous level.

5. Extend the second example to clocked oracle machines that operate in space 2^{kn}, $k>0$ an integer, and add clocks from $T=G=\{2^{kn} \mid k>0 \text{ an integer}\}$ to bound both the amount of nondeterminism and also the number of oracle queries allowed in computations. The theorem states that there exists a set A such that relative to A there is an infinite hierarchy with the property that at each level there is a set that is immune for the previous level.

For other examples of settings where the theorem is applicable, see [6], [14].

Now we turn to the proof of the theorem. The overall outline follows the proof of the proposition. In that case a time bound for a machine served to bound both the amount of nondeterminism available and also the number of oracle queries allowed. Here these two parameters are independent modulo the hypotheses of the theorem.

Proof of the theorem. Let $M_1, M_2, \cdots,$ be an enumeration of the machines in $\mathbf{M}$. Let $g[0](n)=0$ for all n. Let $\pi(0), \pi(1), \cdots,$ be an enumeration of the set $\{\langle\alpha, \beta, \gamma\rangle \mid \alpha > 0, \gamma > 0, \gamma > \beta \geqq 0$, and machine M_α operates in nondeterminism $g[\beta]\}$. We abuse the notation by writing α, β, and γ for functions such that for all n, $\pi(n) = \langle\alpha(n), \beta(n), \gamma(n)\rangle$.

Notice that hypotheses (i) and (ii) imply that for every $j>0$, $g[j]$ is unbounded.

Construct set A in stages as follows:

Stage 0.
$\mu(0) := 0$.
$A_0 := \varnothing$.
$R_0 := \{0\}$.

Stage $n \geqq 1$. Let $B(n) = \{m \leqq n | \beta(m) \leqq \beta(n)\}$. By hypothesis, if $m \in B(n)$, then $g[\beta(m)] = o(g[\gamma(n)])$ since $\beta(m) \leqq \beta(n) < \gamma(n)$; also, $\log t[\alpha(m)] = o(g[\gamma(n)])$. Thus, there exists k such that

(i) $2^{g[\gamma(n)](k)} > \sum_{m \in B(n)} 2^{g[\beta(m)](k)} \cdot t[\alpha(m)](k)$,
(ii) $g[\gamma(n)](k) > \max(\{|y| \,|\, y \in A_{n-1}\} \cup \{g[\gamma(m)](|y|) | m < n \text{ and } y \in A_{n-1}\})$,
(iii) $k > \mu(n-1)$.

Let $\mu(n)$ be the least such k.

Search for the smallest $j \in R_{n-1}$ such that $0^{\mu(n)} \in L(M_{\alpha(j)}, A_{n-1})$.

Case 1. No such j is found. Search for a string w, $|w| = g[\gamma(n)](\mu(n))$ such that on input $0^{\mu(n)}$ no machine $M_{\alpha(j)}$, $j \in B(n)$, queries its oracle about w in any of its computations on $0^{\mu(n)}$ relative to A_{n-1}. Such a string exists since for any $\langle \alpha, \beta, \gamma \rangle$ on an input of length q relative to a fixed oracle set, M_α can have at most $2^{g[\beta](q)}$ computations and each computation can query the oracle at most $t[\alpha](q)$ times so there are at most $2^{g[\beta](q)} \cdot t[\alpha](q)$ strings that are candidates for oracle queries. There are $2^{g[\gamma(n)](\mu(n))}$ words of length $g[\gamma(n)](\mu(n))$ and, as above, $2^{g[\gamma(n)](\mu(n))} > \sum_{m \in B(n)} 2^{g[\beta(m)](\mu(n))} \cdot t[\alpha(m)](\mu(n))$, and so such a w exists.

Let w be the least such string in some effective enumeration of strings and let $A_n := A_{n-1} \cup \{w\}$. Let $R_n := R_{n-1} \cup \{n\}$.

Case 2. Some index j is found. Let $R_n := (R_{n-1} - \{j\}) \cup \{n\}$ and let $A_n := A_{n-1}$.

End of construction.

Define A as $A := \bigcup_{n \geqq 0} A_n$.

Notice that for all $n \geqq 1$, $\mu(n) > \mu(n-1)$, and if w is put into A_n at stage n, then w is never removed from A. Further, if w is put into A_n at stage n, then $|w| > \max\{|y| \,|\, y \in A_{n-1}\}$. Thus, for each $w \in \{0,1\}^*$ there is at most one stage when w can be put into A and there is at most one string of length $|w|$ in A.

For each integer $j \geqq 0$, let $L_j(A) = \{0^p |$there exist n and w such that $\gamma(n) = j$, w is put into A at stage n, $\mu(n) = p$, and $|w| = g[j](p)\}$. Clearly, $L_j(A) \in \mathrm{D}(\mathbf{M}, A)_{g[j]}$. Let $L_g(A) = \bigcup_{j \geqq 0} L_j(A)$ so that $L_g(A) = \{0^p |$there exist w and n such that w is put into A at stage n, $\mu(n) = p$, and $|w| = g[\mu(n)](p)\}$. Clearly, $L_g(A) \in \mathrm{D}(\mathbf{M}, A)_\infty$.

CLAIM 1. *For all i, j, if $i \neq j$, then $L_i(A) \cap L_j(A) = \varnothing$.*

Proof. As noted above, for every $w \in \{0,1\}^*$ there is at most one stage when w can be put into A and there is at most one string of length $|w|$ in A. Thus, for every $w \in \{0,1\}^*$, there is at most one stage when a string of length $|w|$ can be put into A. If $w \in \{0,1\}^*$ is put into A at stage n, then $|w| = g[\gamma(n)](\mu(n)) > \max(\{|y| \,|\, y \in A_{n-1}\} \cup \{g[\gamma(m)](|y|) | m < n \text{ and } y \in A_{n-1}\})$ so that $0^{\mu(n)} \in L_n(A)$ and $0^{\mu(n)} \notin L_m(A)$ for any $m < n$. Since $i \neq j$ implies $i < j$ or $j < i$, $L_i(A) \cap L_j(A) = \varnothing$ as claimed. □

CLAIM 2. *If $0^p \in L_g(A)$, then there is a unique n such that there is a string w that is put into A at stage n and $|w| = g[\mu(n)](p)$.*

Proof. This follows immediately from the definition of $L_g(A)$ and Claim 1. □

CLAIM 3. *For each $j > 0$, $L_j(A)$ is infinite.*

Proof. Suppose that for some j, $L_j(A)$ is finite. Then there is an n_0 such that $\gamma(n_0) = j$ and for all $n \geqq n_0$, if $\gamma(n) = j$, then Case 2 occurs at stage n so that $\|R_n\| = \|R_{n-1}\|$. But there are infinitely many i such that $L(M_i, A) = \varnothing$ and M_i is deterministic.

Thus, there are infinitely many m such that $L(M_{\alpha(m)}, A)=\emptyset$ and $\gamma(m)=j$ so that Case 1 occurs at stage m and $\|R_m\|=\|R_{m-1}\|+1$, a contradiction. □

CLAIM 4. *For each i, j with $i<j$, $L_j(A)$ is* $\mathrm{D}(M, A)_{g[i]}$*-immune.*

Proof. Suppose that there is an infinite $C \subseteq L_j(A) \subseteq \{0^{\mu(n)} | n \geqq 0\}$ such that $C \in \mathrm{D}(M, A)_{g[i]}$. Let m be the least integer with the property that $\pi(m)=\langle k, i, j\rangle$ and $L(M_k, A)=C$.

Since C is infinite and $C \subseteq \{0^{\mu(n)} | n \geqq 0\}$, there are infinitely many n such that $0^{\mu(n)} \in C$. But there are only finitely many indices smaller than m so there must be a $p>m$ such that Case 2 with index m occurs at stage p. At stage p, $A_p := A_{p-1}$ since Case 2 applies so that $0^{\mu(p)} \notin L_g(A)$ by Claim 2. Since Case 2 with index m occurs at stage p, $0^{\mu(p)} \in L(M_{\alpha(m)}, A_{p-1})$. But $\alpha(m)=k$, and $0^{\mu(p)} \in L(M_k, A_{p-1})$ if and only if $0^{\mu(p)} \in L(M_k, A)$, so $0^{\mu(p)} \in L(M_k, A) \subseteq L_g(A)$, a contradiction.

Thus, there is no infinite subset of $L_j(A)$ in $\mathrm{D}(\mathbf{M}, A)_{g[i]}$. By Claim 3, $L_j(A)$ is infinite. Hence, $L_j(A)$ is $\mathrm{D}(\mathbf{M}, A)_{g[i]}$-immune as claimed. □

As noted above $L_j(A)$ is in $\mathrm{D}(\mathbf{M}, A)_{g[i]}$ and $L_j(A)$ is $\mathrm{D}(M, A)_{g[i]}$-immune. Hence, the theorem is proved. □

5. Remarks. In the statement of the Second Immunity Theorem, nothing is said about the set A being recursive; indeed, generally A is not recursive. However, if $\mathbf{M}$ is a class of machines that halt on every input and if the functions in $\mathbf{G}$ and $\mathbf{T}$ are total recursive so that the sets in $\mathrm{D}(\mathbf{M}, ?)_g$ can be specified by clocked machines for every oracle set, then one can choose A to be a recursive set.

Let us attempt to interpret the Second Immunity Theorem. One might ask whether the various theorems separating relativized complexity classes specified by nondeterministic machines from those specified by deterministic oracle machines (as in [1], [5]) speak to the difference between deterministic and nondeterministic computation in general or, instead, illustrate the power of nondeterminism in steps that write on the query tape. Consider the proof of the Second Immunity Theorem: if sufficient work space is available, membership of $L_j(A)$ in $\mathrm{D}(M, A)_{g[j]}$ may be witnessed by a machine that on input 0^p nondeterministically guesses a string $w \in \{0, 1\}^*$ such that $|w|=g[j](p)$ and queries the oracle about w's membership in $A(j)$. Equivalently, a machine might simply compute $g[j](p)$, write $0^{g[j](p)}$ on the query tape, and then ask "does there exist a string $w \in A(j)$ whose length is the same as that of the string on the query tape?" that is, the additional nondeterminism is absorbed by the existential quantifier in the question "does there exist $w \in A(j)$ such that $|w|=0^{g[j](p)}$?" Juris Hartmanis (personal communication) has pointed out that the use of nondeterministic oracle machines by Baker, Gill, and Solovay in [1] have precisely this form. It is easy to see that the proof of the Separation Theorem of Xu, Doner, and Book [14] uses nondeterminism in precisely this form. The fact that both the proof of both of the Immunity Theorems as well as the proof of the Separation Theorem can be described in this way as results about *limited* nondeterminism, provides strong evidence that indeed these theorems say little about the difference between deterministic and nondeterministic computation. Rather, these theorems illustrate the power of nondeterminism in steps that write on the query tape and so generate a very large set of strings to be queried.

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, *Simplicity, relativizations, and nondeterminism*, submitted for publication.

[3] J. BALCÁZAR AND U. SCHÖNING, *Bi-immune sets for complexity classes*, Math. Systems Theory, to appear.

[4] C. BENNETT AND J. GILL, *Relative to a random oracle A*, $P^A \neq NP^A \neq co\text{-}NP^A$ *with probability* 1, this Journal, 10 (1981), pp. 96–113.

[5] R. BOOK, *Bounded query machines: on* NP *and* PSPACE, Theoret. Comput. Sci., 15 (1981), pp. 27–39.

[6] R. BOOK, C. WILSON AND XU MEI-RUI, *Relativizing time, space, and time-space*, this Journal, 11 (1982), pp. 571–581.

[7] P. FLAJOLET AND J.-M. STEYAERT, *On sets having only hard subsets*, Automata, Languages, and Programming, Lecture Notes in Computer Science 14, Springer-Verlag, New York 1974, pp. 446–457.

[8] S. HOMER AND W. MAASS, *Oracle dependent properties of the lattice of* NP *sets*, Theoret. Comput. Sci., 24 (1983) pp. 279–289.

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

[10] C. M. R. KINTALA, *Computations with a restricted number of nondeterministic steps*, Ph.D. dissertation, Pennsylvania State Univ., University Park, 1977.

[11] C. M. R. KINTALA AND P. FISCHER, *Refining nondeterminism in relativized polynomial time-bounded computations*, this Journal, 9 (1980), pp. 46–53.

[12] K. KO AND D. MOORE, *Completeness, approximation, and density*, this Journal, 10 (1981), pp. 787–796.

[13] U. SCHÖNING, *Relativization and infinite subsets of* NP *sets*, unpublished manuscript, 1982.

[14] XU MEI-RUI, J. DONER AND R. BOOK, *Refining nondeterminism in relativized complexity classes*, J. Assoc. Comput. Mach., 30 (1983), pp. 677–685.

SIAM J. COMPUT.
Vol. 13, No. 2, May 1984

© 1984 Society for Industrial and Applied Mathematics
009

FAST ALGORITHMS FOR FINDING NEAREST COMMON ANCESTORS*

DOV HAREL† AND ROBERT ENDRE TARJAN‡

Abstract. We consider the following problem: Given a collection of rooted trees, answer on-line queries of the form, "What is the nearest common ancester of vertices x and y?" We show that any pointer machine that solves this problem requires $\Omega(\log \log n)$ time per query in the worst case, where n is the total number of vertices in the trees. On the other hand, we present an algorithm for a random access machine with uniform cost measure (and a bound of $O(\log n)$ on the number of bits per word) that requires $O(1)$ time per query and $O(n)$ preprocessing time, assuming that the collection of trees is static. For a version of the problem in which the trees can change between queries, we obtain an almost-linear-time (and linear-space) algorithm.

Key words. graph algorithm, nearest common ancestor, tree, inverse Ackermann's function, random access machine, computational complexity

1. Introduction. Aho, Hopcroft, and Ullman [2] consider the following problem: Given a collection of rooted trees,[1] answer queries of the form, "What is the nearest common ancestor of vertices x and y?" (We shall denote this vertex by *nca* (x, y).) There are actually many versions of this problem, depending upon whether the queries are all specificed in advance and how much the trees change during the course of the queries. We shall consider the five versions of the problem listed below, given in order from least dynamic (easiest) to most dynamic (hardest).

Problem 1 (off-line). The collection of trees is static and the entire sequence of queries is specified in advance.

Problem 2 (static trees). The collection of trees is static but the queries are given on-line. That is, each query must be answered before the next one is known.

Problem 3 (linking roots). The queries are given on-line. Interspersed with the queries are on-line commands of the form *link* (x, y) where x and y are tree roots. The effect of a command *link* (x, y) is to combine the trees containing x and y by making x the parent of y.

Problem 4 (linking). The queries are on-line. Interspersed with the queries are on-line commands *link* (x, y) such that y, but not necessarily x, is a tree root.

Problem 5 (linking and cutting). The queries are on-line. Interspersed with the queries are on-line commands of two types: *link* (x, y), where y but not necessarily x is a tree root, and *cut* (x), where x is not a root. The effect of a command *cut* (x) is to cut the edge connecting x and its parent, splitting the tree containing x into two trees: one containing all descendants of x and another containing all nondescendants of x.

In our discussion, we shall use n to denote the total number of vertices in the trees and m to denote the total number of operations (queries, links, and cuts). We shall distinguish between two machine models: *pointer machines* [8], [12] and *random access machines* [1]. In a pointer machine, memory consists of a collection of *nodes* (sometimes called *records*). Each node consists of a fixed number of *fields*. The fields have associated *types*, such as *pointer, integer, real.* A field of a given type contains a value of that type; for instance, a pointer field contains a pointer to a node. In a

* Received by the editors May 24, 1982, and in final form July 21, 1983.

† University of New Hampshire, Durham, New Hampshire 03824. Present address: Intermetrics, Inc., Cambridge, Massachusetts 02138.

‡ Bell Laboriatories, Murray Hill, New Jersey 07974.

[1] See the appendix for the tree terminology used in this paper.

random access machine the memory is an array of words, each of which holds an integer expressed in binary.

For measuring time on a random access machine, we use the *uniform cost measure*, in which each operation on a word or pair of words (such as an addition, comparison or branch) requires $O(1)$ time. We place an upper bound of $O(\log n)$ on the number of bits a word can hold, thereby precluding fast algorithms that obtain the effect of parallelism by manipulating very large integers. The main difference between our machine models is that address arithmetic is possible on random access machines but not on pointer machines.

Let us review what is known about Problems 1–5. In their seminal paper, Aho, Hopcroft, and Ullman consider Problems 1, 2 and 4. They describe an $O(n+m\alpha(m+n, n))$-time algorithm running on a pointer machine for Problem 1 (off-line). Here α is the functional inverse of Ackermann's function defined by Tarjan [10], [13]. Their algorithm requires $O(n)$ storage. For Problem 2 (static trees) they propose a random access machine algorithm requiring $O(n \log\log n)$ preprocessing time, $O(n \log\log n)$ space, and $O(\log\log n)$ time per query. This algorithm uses their algorithm for Problem 4 (linking), which also runs on a random-access machine and requires $O((m+n)\log n)$ time and $O(n \log n)$ space.

Several more recent papers improve and extend the results of Aho, Hopcroft, and Ullman. Van Leeuwen [14] considers Problem 3 (linking roots). He gives an $O(n+m \log\log n)$-time algorithm that can be modified to run on a pointer machine in $O(n)$ space. Maier [7] addresses Problem 5 (linking and cutting). Although his algorithm is not very time-efficient, his results do improve the space efficiency of Aho, Hopcroft, and Ullman's algorithm for Problem 4 (linking). Sleator and Tarjan [9] use their data structure for dynamic trees to solve Problem 5 in $O(n+m \log n)$ time and $O(n)$ space on a pointer machine.

Table 1 summarizes the known results. For Problem 1, Aho, Hopcroft, and Ullman's $O(n+m\alpha(m+n, n))$-time algorithm is the fastest known. For Problems 2 and 3, van Leeuwen's $O(n+m \log\log n)$-time algorithm is fastest. For Problems 4 and 5, Sleator and Tarjan's $O(n+m \log n)$-time algorithm is best. All these algorithms use $O(n)$ space and run on pointer machines.

TABLE 1
Best pointer machine algorithms for finding nearest common ancestors

Problem	Algorithm	Time	Space
1. Off-line	Aho, Hopcroft, and Ullman [2]; see also Tarjan [11]	$O(n+m\alpha(m+n, n))$	$O(n)$
2. Static	modified van Leeuwen [14]	$O(n+m \log\log n)$	$O(n)$
3. Linking roots	modified van Leeuwen [14]	$O(n+m\log\log n)$	$O(n)$
4. Linking	Sleater and Tarjan [9]	$O(n+m \log n)$	$O(n)$
5. Linking and cutting	Sleator and Tarjan [9]	$O(n+m \log n)$	$O(n)$

In this paper, our goal is to study the effect of the machine model on the nearest common ancester problem. Our results are three. In § 2 we show that any pointer machine requires $\Omega(\log\log n)$ time per query to solve Problem 2 (static trees). This means that van Leeuwen's algorithm is optimum to within constant factors for Problems 2 and 3. In §§ 3–5 we develop an algorithm for Problem 2 that runs on a random access machine and uses $O(n)$ preprocessing time, $O(1)$ time per query, and $O(n)$ space. This algorithm is also optimum to within a constant factor. Harel's paper [5]

TABLE 2
Random access machine algorithms for finding nearest common ancestors

Problem	Time	Space
1. Off-line	$O(n+m)$	$O(n)$
2. Static	$O(n+m)$	$O(n)$
3. Linking roots	$O(n+m\alpha(m+n, n))$	$O(n)$

gives a preliminary version of the results in §§ 2–5. In §§ 6 and 7 we extend our algorithm to Problem 3, for which we obtain an $O(n+m\alpha(m+n, n))$-time, $O(n)$-space algorithm. Our results thus explicate the difference in power between pointer machines and random-access machines. Table 2 summarizes our upper bounds.

2. A lower bound for pointer machines. In this and the next three sections we shall restrict our attention to the nearest common ancester problem on static trees (Problem 2). Without loss of generality we can assume that there is only one tree. If not, we create a new (dummy) vertex r and make it the parent of the roots of all the actual trees. The nearest common ancester *nca* (x, y) of two vertices x and y in the new tree is the same as the nearest common ancester of x and y in the collection of old trees; if *nca* $(x, y)=r$ in the new tree then x and y are in different old trees.

Let us consider pointer machine solutions to the nearest common ancester problem on a static tree. We make the following assumptions about the way such a machine solves the problem. We assume that the tree is represented by a list structure (which may change during the course of the computation), with each tree vertex represented by a single node. The structure may contain additional nodes not representing any tree vertex. Each node contains a fixed number of pointers, independent of n; without loss of generality we may take this number to be two. As input for a query, the algorithm is given pointers to the nodes corresponding to two tree vertices x and y. To answer the query, the algorithm must return a pointer to the node corresponding to the tree vertex *nca* (x, y). We assume that the algorithm remembers nothing between queries. (Any fixed amount of global memory can be encoded into the list structure.)

THEOREM 1. *Let T be a complete binary tree with n leaves. Any pointer machine requires $\Omega(\log \log n)$ time to answer any nca query in the worst case, independent of the representation of the tree.*

Proof. Let us fix our attention on the time just before a query. The key point is that, from any node in the data structure, at most $2^{j+1}-1$ nodes are accessible in j steps or less. Let k be such that any possible *nca* query can be answered in k steps or less. For each leaf x of T let A_x denote the set of nodes representing tree vertices that are accessible from x in k steps or less. Let w be a nonleaf vertex of T and let u and v be its two children. We claim that either w belongs to A_x for every leaf x that is a descendant of u, or w belongs to A_y for every leaf y that is a descendant of v. Otherwise, for some descendant x of u and some descendant y of v, w would be accessible from neither x nor y in k steps, and an *nca* query on x and y would be unanswerable in k steps.

We conclude that w belongs to A_x for at least half the leaves x that are descendants of w. If w has height $i\geqq 1$, then w has 2^i leaf descendants, and thus w occurs in at least 2^{i-1} sets A_x. Since if $1\leqq i\leqq h=\lg n$[2] there are 2^{h-i} vertices of height i, we see

[2] We use $\lg n$ to denote $\log_2 n$.

that vertices of height i contribute $2^{h-i}2^{i-1}=2^{h-1}=n/2$ occurrences to the collection of sets A_x. Summing over all heights i from 1 to h, we find that

$$\sum_{x \in L} |A_x| \geqq \frac{n}{2} \lg n,$$

where L is the set of leaves of T. Since for any leaf x we have $|A_x|<2^{k+1}$, we get

$$n2^{k+1} > \frac{n}{2} \lg n,$$

which means

$$k > \lg \lg n - 2. \qquad \square$$

Theorem 1 implies that van Leeuwen's algorithm for static trees runs in optimum time, to within a constant factor. It also provides a nontrivial lower bound for pointer machines on a natural problem. (For a similar result see [12].) We conjecture that Problem 5 (linking and cutting) requires $\Omega(\log n)$ time per command in the worst case on a pointer machine, and leave the proof (or disproof) of this conjecture as an open problem.

3. Overview of a fast algorithm for static trees. If we allow algorithms on random-access machines, we can beat the lower bound in Theorem 1. In this and the next two sections, we shall develop an algorithm that runs on a random-access machine in $O(n)$ space, using $O(n)$ preprocessing time and $O(1)$ time per query. These bounds are best possible; there are n^{n-1} distinct rooted trees with n labeled vertices [6], implying that $\Omega(n \log n)$ bits, or $\Omega(n)$ words, are necessary to store a tree of n vertices in a random-access machine. Just reading in such a tree requires $\Omega(n)$ time, and answering a query requires $\Omega(1)$ time.

We begin by observing that on complete binary trees the nearest common ancestor and related problems can be solved in $O(1)$ time by direct calculation. Let T be a complete binary tree whose vertices are numbered from 1 to n in symmetric order. (See Fig. 1.) We use *sym* (v) to denote the number of a vertex v and $sym^{-1}(i)$ to

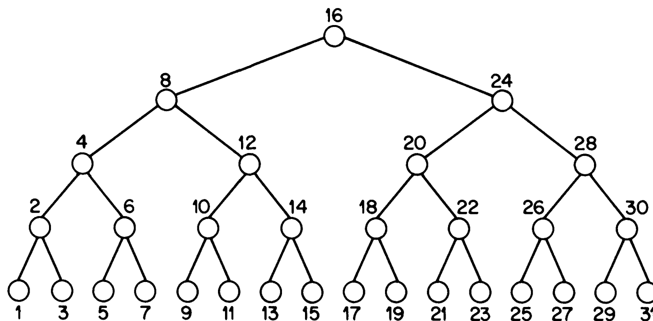


FIG. 1. *Symmetric-order numbering of a complete binary tree.*

denote the vertex whose number is i. For any height h, the vertices of height h are numbered 2^h, $3\cdot 2^h$, $5\cdot 2^h$, $\cdots$ from left to right. It is easy to verify the following facts about the numbering:

LEMMA 1. *The height of a vertex v is the largest integer h such that 2^h divides sym (v). We shall denote this height by $h(v)$.*

LEMMA 2. *The descendants of vertex v are those vertices with numbers in the range $[sym\,(v)-2^{h(v)}+1,\ sym\,(v)+2^{h(v)}-1]$.*

LEMMA 3. *If v is a vertex and h is a height such that $h \geqq h(v)$, then the ancestor of v of height h has number $2^{h+1}\lfloor \mathrm{sym}\,(v)/2^{h+1}\rfloor+2^h$.*

LEMMA 4. *If v and w are two unrelated vertices, the height of the nearest common ancestor of v and w is $\lfloor \lg\,(sym(v) \oplus sym\,(w))\rfloor$, where $i \oplus j$ is the integer whose binary representation is the bitwise exclusive or of the binary representations of i and j.*

As examples of Lemma 3, vertex number 22 has vertex number $20=8\lfloor 22/8\rfloor+4$ as its ancestor of height 2 and vertex number $24=16\lfloor 22/16\rfloor+8$ as its ancestor of height 3. (See Fig. 1.) As an example of Lemma 4, the nearest common ancestor of vertices number 20 and number 27 is number 24, of height 3; $\lfloor \lg\,(20 \oplus 27)\rfloor = \lfloor \lg\,(10100_2) \oplus 11011_2\rfloor = \lfloor \lg\,(1111_2)\rfloor = 3$.

Lemmas 2 and 4 allow us to solve the following problem in $O(1)$ time, given that we know the number and height of each vertex and the depth d of the tree:

The nca depth problem. Given vertices v and w, compute the depth of the nearest common ancestor of v and w.

Algorithm to solve the nca depth problem. If v is an ancestor of w (that is, if $sym\,(w) \in [sym\,(v)-2^{h(v)}+1,\ sym\,(v)+2^{h(v)}-1]$), return $d-h(v)$. If w is an ancestor of v, return $d-h(w)$. If v and w are unrelated, return $d-\lfloor \lg\,(sym\,(v) \oplus sym\,(w))\rfloor$.

Lemmas 1 and 3 provide a way to solve the following problem in $O(1)$ time, if we know the number of each vertex, the vertex corresponding to each number, and the depth d of the tree:

The depth problem. Given a vertex v of depth d_1 and a depth $d_2 \leqq d_1$, compute the ancestor of v whose depth is d_2.

Algorithm to solve the depth problem.
Let $h=d-d_2$. Return $sym^{-1}\,(2^{h+1}\lfloor sym\,(v)/2^{h+1}\rfloor+2^h)$.

These algorithms combine to give an $O(1)$-time algorithm for the nearest common ancestor problem:

Algorithm to compute nca(v, w).
Step 1. Compute d_0, the depth of *nca* (v, w).
Step 2. Compute and return the depth-d_0 ancestor of v.

The $O(1)$ time bounds for these three algorithms depend on the ability to perform multiplication, division, powers of two, base two discrete logarithm, and exclusive or in $O(1)$ time. If these operations are not part of the machine's repertoire, we can in $O(n)$ time construct operation tables that allow these operations to be carried out in $O(1)$ time using (possibly repeated) table look-up.

Let us turn now to the nearest common ancestor problem on an arbitrary tree T. Our plan is to convert the *nca* problem on T into an *nca* problem on a subtree of a moderately-sized complete binary tree; then we can use the method above. The transformation proceeds by a sequence of steps, which involve solving depth and *nca* depth problems on two auxiliary trees: a *compressed tree C* and a *balanced binary tree B*. C has the same vertex set as T; B contains all vertices in T and possibly some auxiliary vertices. To facilitate the solution of depth problems on B C, both of these trees are divided into *plies.* We construct B and C in a preprocessing step requiring $O(n)$ time. We compute nearest common ancestors as follows:

Algorithm to compute $nca_T(v, w)$.

Step 1. Compute $nca_C(v, w)$ as follows:

Step 1a. Compute $nca_B(v, w)$ using algorithms for the *nca* depth problem and the depth problem on *B*.

Step 1b. Given $nca_B(v, w)$, look up $nca_C(v, w)$.

Step 2. Look up the depth in *C* of $nca_C(v, w)$. Using an algorithm for the depth problem on *C*, compute $nca_T(v, w)$.

Each step of the *nca* computation takes O(1) time. Sections 4 and 5, which discuss the compressed tree *C* and the balanced binary tree *B* respectively, give the details of this method.

4. The compressed tree. Let *T* be an arbitrary *n*-vertex tree with root *r*. We define a compressed tree *C*, representing *T*, as follows. For each vertex *v* in *T*, let *size* (v) be the number of descendants of *v* (including *v* itself) in *T*. Define an edge $v \to p_T(v)$ to be *light* if $2 \cdot size_T(v) \leqq size_T(p_T(v))$ and *heavy* otherwise. Since the size of a vertex is one greater than the sum of the sizes of its children, at most one heavy edge enters each vertex. Thus the heavy edges partition the vertices of *T* into a collection of *heavy paths*. (A vertex with no entering or exiting heavy edge is a single-vertex heavy path.) (See Fig. 2.)

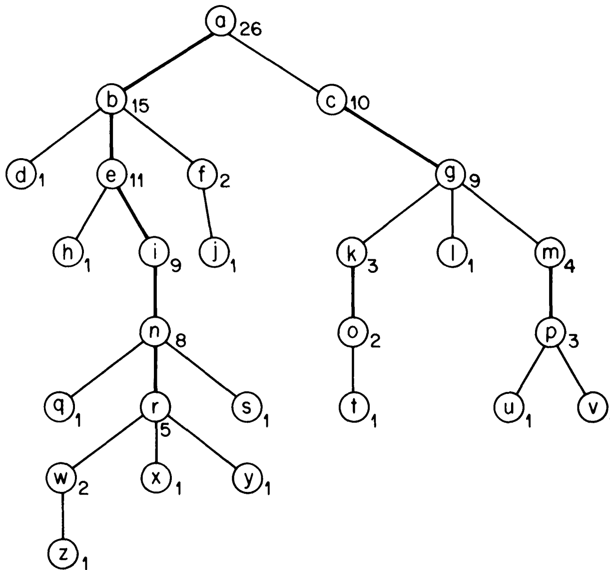


FIG. 2. *Heavy and light edges in a tree. Numbers are vertex sizes.*

The *apex* of a heavy path is the vertex on the path of smallest depth. For any vertex *v* we denote by *apex* (v) the apex of the heavy path containing *v*, and by *hp size* (v) the number of descendants of *v* on the same heavy path as *v*. The *compressed tree C* is defined by the set of edges

$$\{v \to apex\,(p_T(v)) \mid v \text{ is a vertex of } T \text{ other than } r\}.$$

(See Fig. 3.) The compressed tree was used by Tarjan [10] to compute functions defined on paths in trees; Aho, Hopcroft, and Ullman used a closely related idea in their *nca* algorithm for static trees.

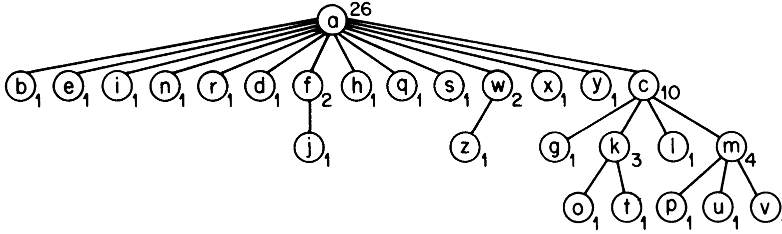


FIG. 3. *The compressed tree corresponding to the tree of Fig. 2. Numbers are vertex sizes.*

In $O(n)$ time, we can compute the following information for each vertex v: $p_T(v)$, $p_C(v)$, *apex* (v), *hp size* (v), $d_C(v)$ (the depth of v in C), and $size_C$ (v) (the size of v in C). Note that we can use *apex* to test in $O(1)$ time whether two vertices are on the same heavy path in T. To compute $nca_T(v, w)$, we first compute $nca_C(v, w)$ using the balanced tree B. (We describe how to do this in the next section.) Then we proceed as follows:

Step 2 *of algorithm to compute* nca_T (v, w):

Step 2a. Let $u = nca_C$ (v, w). (Either $u = v = w$ or u is the apex of the heavy path containing $nca_T(v, w)$). If $u = v$ or $u = w$, return u as nca_T (v, w). Otherwise look up $d_C(u)$.

Step 2b. Compute the ancestor v' of v in C whose depth is $d_C(u)+1$. If *apex* $(v') = u$ (u and v' are on the same heavy path) let $v'' = v'$. Otherwise let $v'' = p_T(v')$.

Step 2c. Compute the ancestor w' of w in C whose depth is $d_C(u)+1$. If *apex* $(w') = u$ (u and w' are on the same heavy path) let $w'' = w'$. Otherwise let $w'' = p_T(w')$.

Step 2d. Return as $nca_T(v, w)$ whichever of v'' and w'' has the larger value of *hp size.*

As an example of Step 2, consider nca_T (n, o), where T is the tree of Fig. 2. We have $nca_C(n, o) = a$ (see Fig. 3), $n' = n$, $n'' = n$, $o' = c$, $o'' = a$, and nca_T $(n, o) = a$. We omit the easy proof that Step 2 is correct; crucial to the proof is the observation that $v''(w'')$ is the nearest ancestor of v (respectively w) on the heavy path containing u. Step 2 requires $O(1)$ time plus the solution of at most two depth problems on C. To solve depth problems on C, we must do some additional preprocessing. We need some simple facts about the structure of C, which we state without proof. (See [11].)

LEMMA 5. *If v is an apex, $size_C$ $(v) = size_T$ (v); if v is not an apex, $size_C(v) = 1$.*

LEMMA 6. *Every edge $v \to p_C(v)$ of C satisfies $2 \cdot size_C$ $(v) \geqq size_C$ $(p_C(v))$.*

LEMMA 7. *C has depth at most $\lfloor \lg n \rfloor$.*

We divide C into three plies as follows. Define the *rank* of a vertex v to be *rank* $(v) = \lfloor \lg (size_C$ $(v)) \rfloor$. Ply three consists of all vertices with rank $\lfloor \lg^{(2)} n \rfloor$[3] or greater; ply two consists of all vertices with rank between $\lfloor \lg^{(3)} n \rfloor$ and $\lfloor \lg^{(2)} \rfloor - 1$ (inclusive); ply one consists of all vertices with rank less than $\lfloor \lg^{(3)} n \rfloor$. Fig. 4 schematically illustrates this definition.

LEMMA 8. *For any rank i, the number of vertices of rank i is at most $n/2^i$.*

Proof. By Lemma 6, any two vertices having the same rank are unrelated in C and hence have disjoint sets of descendants in C. Any vertex v of rank i has $size_C$ $(v) \geqq 2^i$. Thus there can be at most $n/2^i$ vertices of rank i. □

[3] For any nonnegative integer i and function f, we define $f^{(i)}(x)$ by $f^{(0)}(x) = x$, $f^{(i+1)}(x) = f(f^{(i)}(x))$.

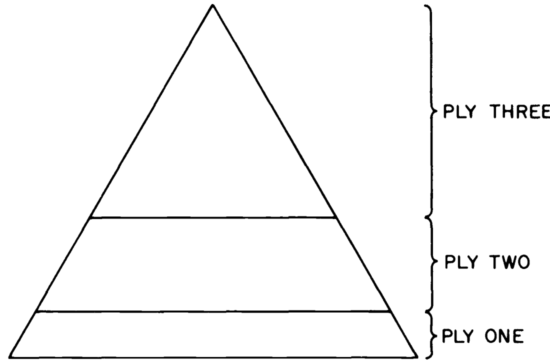


FIG. 4. *Plies of a compressed tree.*

LEMMA 9. *Ply three contains at most $O(n/\log n)$ vertices. Ply two contains at most $O(n/\log^{(2)} n)$ vertices. Each connected component of ply one is a subtree of C containing at most $\log^{(2)} n$ vertices.*

Proof. By Lemma 8, the total number of vertices with rank k or greater is at most $\sum_{i=k}^{\infty} n/2^i = n/2^{k-1}$. This implies that ply three contains at most $4n/\lg n$ and ply two at most $4n/\lg^{(2)} n$ vertices. Each connected component of ply one is a subtree of C whose root has rank at most $\lfloor \lg^{(3)} n \rfloor - 1$ and thus contains at most $2^{\lfloor \lg^{(3)} n \rfloor} \leqq \lg^{(2)} n$ vertices. □

With each vertex v in ply i ($i \in \{1, 2, 3\}$), we store the vertex $a(v)$ such that $a(v)$ is the shallowest ancestor of v in C whose ply is i. We also represent each ply so that it is possible to carry out the following computation in $O(1)$ time: given a vertex v in ply i and a depth $d \in [d_C(a(v)), d_C(v))]$, compute the ancestor of v whose depth is d. (We shall describe this representation below.) Then we can solve the depth problem on C as follows.

Algorithm to compute the ancestor of vertex v in C whose depth is d. Repeat the following step until a vertex is returned:

General step. If $d \in [d_C(a(v)), d_C(v)]$ compute and return the ancestor of v whose depth is d. Otherwise, replace v by $p_C(a(v))$.

This method requires $O(1)$ time. (The ply of v increases by at least one with each iteration of the general step.)

The representation of plies two and three is very simple: with each vertex v of ply i ($\in \{2, 3\}$), we store an array of all ply-i ancestors of v, indexed by depth. Then we can solve a depth problem within ply two or three in $O(1)$ time by table look-up. Note that the size of all arrays for ply three is $O((n/\log n) \log n) = O(n)$ by Lemma 9, since any vertex has $O(\log n)$ ancestors in C. Similarly the size of all arrays for ply two is $O((n/\log^{(2)} n)\log^{(2)} n = O(n)$ by Lemma 9, since any vertex has $O(\log^{(2)} n)$ ply-two ancestors in C. It is straightforward to divide the vertices into plies, compute $a(v)$ for every vertex, and construct all the arrays for plies two and three in $O(n)$ time.

Ply one has very small connected components but contains most of the vertices in C, and we represent it differently than plies two and three. Let v be any vertex in ply one. We denote by $D_{a(v)}$ the subtree of C rooted at $a(v)$; this is the connected component of ply one containing v. $D_{a(v)}$ contains at most $\lg^{(2)} n$ vertices and has depth at most $\lg^{(3)} n$. Let d be the depth of $D_{a(v)}$. We embed the vertices of $D_{a(v)}$ in a complete binary tree $E_{a(v)}$ of depth $d\lceil \lg^{(3)} n \rceil$ so that a vertex of depth i in $D_{a(v)}$

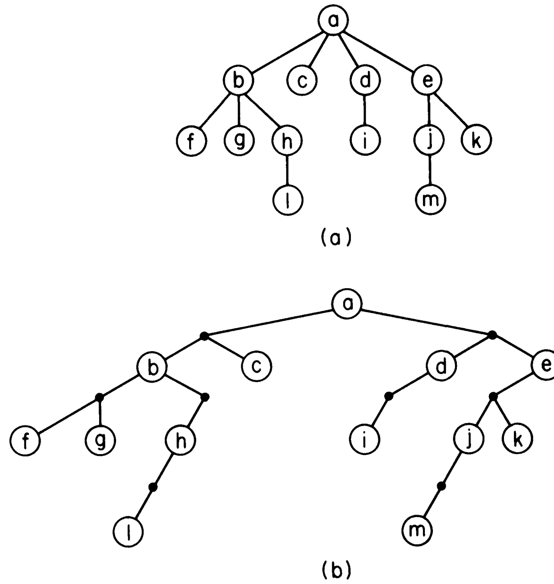


FIG. 5. *Embedding a multiway tree into a complete binary tree.* (a) *Multiway tree.* (b) *Part of complete binary tree containing the multiway tree.*

has depth $i\lfloor \lg^{(3)} n \rceil$ in $E_{a(v)}$, and so that if v is a vertex of depth i in $D_{a(v)}$ and $0 \leqq i' \leqq i$, the ancestor of v in $D_{a(v)}$ whose depth is i' is also the ancestor of v in $E_{a(v)}$ whose depth is $i'\lceil \lg^{(3)} n \rceil$. (See Fig. 5.) Then we can solve depth problems in ply one using the method of § 3. The algorithm is as follows:

Algorithm to compute the ancestor of v in C whose depth is d (v is in ply one and $d \geqq d_C(a(v))$).

Let $h = (d_c(a(v)) - d)\lceil \lg^{(3)} n \rceil$. Return

$$sym^{-1}_{a(v)}\,(2^{h+1}\lfloor sym\,(v)/2^{h+1}\rfloor + 2^h).$$

The notation in this algorithm requires a little explanation. If v is any vertex in ply one, $sym\,(v)$ is the symmetric-order number of v corresponding to its position in $E_{a(v)}$, and $sym^{-1}_{a(v)}\,(i)$ is the vertex in $E_{a(v)}$ with number i, if this vertex is in $D_{a(v)}$; otherwise $sym^{-1}_{a(v)}\,(i)$ is undefined. This method solves depth problems within ply one in $O(1)$ time, if sym and $sym^{-1}_{a(v)}$ are precomputed.

The difficulty with this method is that the trees $E_{a(v)}$ are too big to construct explicitly, so we must represent them implicitly. To represent the trees $E_{a(v)}$, we store two numbers, $sym\,(v)$ and $pre\,(v)$, with each vertex v in ply one, and two arrays, $pre^{-1}_{a(v)}$ and $inverse_{a(v)}$, with each vertex $a(v)$ in ply one. The values of pre, pre^{-1}, $inverse$ are defined as follows: We number the vertices of each tree $D_{a(v)}$ in preorder from 1 to $|D_{a(v)}|$. If v is a vertex in $D_{a(v)}$, $pre\,(v)$ is the number of v and $pre^{-1}_{a(v)}\,(pre\,(v)) = v$. Note that the total size of the arrays $pre^{-1}_{a(v)}$ for all vertices $a(v)$ in ply one is $O(n)$. For $i \in [1, |E_{a(v)}|]$, $inverse_{a(v)}(i)$ is the pre-order number of the vertex in $D_{a(v)}$ whose symmetric-order number in $E_{a(v)}$ is i; $inverse_{a(v)}\,(i) = 0$ if the vertex in $E_{a(v)}$ whose symmetric-order number is i is not in $D_{a(v)}$. Note that $inverse_{a(v)}$ contains $|D_{a(v)}|$ nonzero entries and needs $O(|E_{a(v)}| \lg |D_{a(v)}|) = 2^{O((\lg^{(3)} n)^2)} \lg^{(3)} n = O(\log n)$ bits for its storage. Thus the total number of nonzero entries in all the $inverse_{a(v)}$ arrays is $O(n)$, and each $inverse_{a(v)}$ array fits into one word of storage.

Given *sym* (v) for each vertex v in ply one and $pre^{-1}_{a(v)}$ and $inverse_{a(v)}$ for each vertex $a(v)$ in ply one, we can solve depth problems within ply one in $O(1)$ time, since $sym^{-1}_{a(v)}(i) = pre^{-1}_{a(v)}(inverse_{a(v)}(i))$ if i is the symmetric-order number in $E_{a(v)}$ of a vertex in $D_{a(v)}$. It is easy to construct the $pre^{-1}_{a(v)}$ arrays in $O(n)$ time. We can construct the *sym* numbers and the $inverse_{a(v)}$ arrays by traversing each ply-one tree in preorder as follows: Let $D_{a(v)}$ be a ply one tree of height h, to be embedded in the complete binary tree $E_{a(v)}$. We initialize $inverse_{a(v)}$ to be zero. Then we execute $traverse(a(v), h\lceil \lg^{(3)} n\rceil, 1)$, where *traverse* is defined as follows:

Procedure *traverse*(v, h, i):

[v is a vertex in $D_{a(v)}$, h is the height of v in $E_{a(v)}$, and i is the symmetric-order number of the smallest-numbered descendant of v in $E_{a(v)}$]

*Step*1. Let $j = 2^h\lceil i/2^h\rceil$ (j is the symmetric-order number of v in $E_{a(v)}$). Assign $sym(v) = j$ and $inverse_{a(v)}(j) = pre(v)$.

Step 2. Initialize $k = i$. For each child w of v, carry out the following steps:

Step 2a. Execute $traverse(w, h - \lceil \lg^{(3)} n\rceil, k)$.

Step 2b. Replace k by $k + 2^{h-\lceil \lg^{(3)} n\rceil+1}$.

It is routine to verify that this method is correct and requires $O(n)$ time to preprocess all the ply-one trees. This completes our description of the algorithm for solving depth problems on C and with it our description of the role of C in the nearest common ancestor algorithm. As noted by Hal Gabow (private communication), it is possible to simplify the solution of depth problems in ply one by using the table look-up technique of Gabow and Tarjan [4]. This requires additional preprocessing, but the additional time is only $O(n)$.

5. The balanced binary tree. To complete our solution of the *nca* problem, we need a way to solve the *nca* problem on the compressed tree C. For this purpose we represent C by a second auxiliary tree B called the *balanced binary tree.* B contains all the vertices of C, called *green vertices*, and some additional vertices, called *red vertices.* For each family in C consisting of a parent v and a set of children W, B contains a subtree whose root is v, whose leaves are the vertices in W, and whose remaining vertices are red. Each vertex v contains a pointer to its nearest green ancestor *green* (v). Nearest common ancestors in B and C are related by the equation

$$nca_C(v, w) = green(nca_B(v, w))$$

if v and w are vertices in C. Thus if we can solve the *nca* problem on B, we can solve the *nca* problem on C in $O(1)$ additional time.

Thus it remains for us to describe how to construct B and how to solve the *nca* problem on B. Consider a family in C consisting of a parent v and a set of children W such that $|W| \geqq 3$. To binarize the family, we execute the recursive procedure *binarize* (v, W), where *binarize* is defined as follows:

Procedure *binarize* (v, W):

Step 1. Let $W = \{w_1, \cdots, w_k\}$ and $s = \sum_{i=1}^{k} size_C(w_i)$. Let j be the minimum index such that $\sum_{i=1}^{j} size_C(w_i) \geqq s/2$. If $j = k$, replace j by $k-1$.

Step 2. *If* $j = 1$, attach w_1 as the left child of v. Otherwise, let x_1 be a new red vertex. Attach x_1 as the left child of v and execute *binarize* (x_1, W_1), where $W_1 = \{w_1, \cdots, w_j\}$.

Step 3. If $j = k-1$, attach w_k as the right child of v. Otherwise, let x_2 be a new red vertex. Attach x_2 as the right child of v and execute *binarize* (x_2, W_2), where $W_2 = \{w_{j+1}, \cdots, w_k\}$.

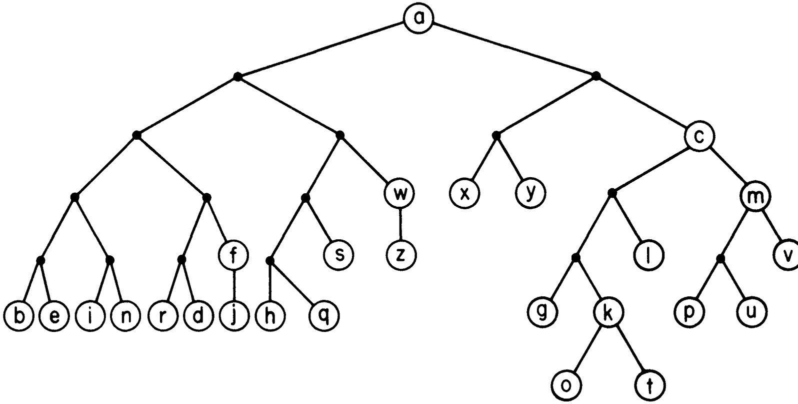


FIG. 6. *Balanced binary tree corresponding to the compressed tree of Fig. 3. Dots are red vertices.*

(See Fig. 6.) This method can be implemented to run in $O(|W|)$ time [3]. The idea is to construct an array of size k, whose jth position contains $\sum_{i=1}^{j} size_C\,(w_i)$. Then one can find the appropriate index j in $O(\log(\min\{j, k-j\}))$ time by using a form of binary search simultaneously from both ends of the array. The same array can be used for all the recursive subproblems. The running time $t(k)$ of the method is given by the recurrence

$$t(k) = \max_{0<j<k} \{t(j) + t(k-j) + O(\log(\min\{j, k-j\}) + 1)\},$$

which has the solution $t(k) = O(k)$.

To construct B, we binarize each family of C using the method above. Since B contains at most $2n-1$ vertices, the total time to construct B is $O(n)$.

Let us define $size_B\,(v)$ for a vertex v in B to be the number of green descendants of v in B. The following lemmas show that the tree B has properties similar to those of C:

LEMMA 10. *If v is a green vertex $size_B\,(v) = size_C\,(v)$.*

Proof. Immediate. □

LEMMA 11. *If v is a vertex such that $p_B^4(v)$ is defined, then $2\ size_B\,(v) \leqq size_B\,(p_B^4(v))$.*

Proof. If the path in B from v to $p_B^4(v)$ contains two green vertices, the lemma is immediate from Lemmas 6 and 10. Otherwise, the path from v to $p_B^4(v)$ contains three consecutive vertices $w \to p_B(w) \to p_B^2(w)$ such that w and $p_B(w)$ are red. If $2 \cdot size_B\,(w) > size_B\,(p_B^2(w))$, the operation of *binarize* guarantees that w is the left child of $p_B(w)$ and $p_B(w)$ is the left child of $p_B^2(w)$. But the right children of $p_B(w)$ and $p_B^2(w)$ together have less than half the size in C of $p_B^2(w)$, which contradicts the operation of *binarize* on $p_B^2(w)$. □

LEMMA 12. *B has depth $O(\log n)$.*

Proof. Immediate from Lemma 11. □

We solve the *nca* problem on B by solving the *nca* depth problem and the depth problem on B. To solve the *nca* depth problem on B we embed B in a complete binary tree B' and use direct calculation as described in § 3; all we need to know for each vertex in B is its symmetric-order number and height (as a vertex in B'). To number the vertices of B we execute the recursive procedure *number*$(r, h, 2^h)$, where

r is the root of B, h is the height of B, and *number* is defined as follows:

Procedure *number* (v, h, i): [v is a vertex in B, h is the height of v in B', and i is the symmetric-order number of v in B']
Step 1. Assign number i and height h to v.
Step 2. If v has a left child w_1, execute *number* $(w_1, h-1, i-2^{h-1})$.
Step 3. If v has a right child w_2, execute *number* $(w_2, h-1, i+2^{h-1})$.

This computation takes $O(n)$ time. Once the vertices of B are numbered, we can solve the *nca* depth problem on B in $O(1)$ time.

We solve the depth problem on B in the same way we solved it on C. For each vertex v in B, we define $rank_B(v) = \lfloor \lg(size_B(v)) \rfloor$. We divide B into three plies as follows: Ply three consists of all vertices with rank $\lfloor \lg^{(2)} n \rfloor$ or greater, ply two consists of all vertices with rank between $\lfloor \lg^{(3)} n \rfloor$ and $\lfloor \lg^{(2)} n \rfloor - 1$ (inclusive), and ply one consists of the remaining vertices. Lemma 9 holds for the plies in B just as for the plies in C, except that the constants are larger by a factor of four because the weaker Lemma 11 holds for B in place of Lemma 6. We represent the plies of B exactly as we did the plies of C, and we solve the depth problem in the same way. We can simplify the representation of ply one a little because B is binary; in particular, we do not need to binarize the ply one trees. Filling in the details is routine and we leave it as an exercise.

Constructing the plies of B requires $O(n)$ preprocessing time and allows us to solve the depth problem on B in $O(1)$ time. Hence we can solve the *nca* problem on B, on C, and on T in $O(1)$ time. This completes our description of the *nca* algorithm for static trees.

6. A fast algorithm for the linking roots problem. We now turn our attention to the linking roots problem. In this and the next section we shall develop an algorithm for this problem that runs in $O(n + m\alpha(m+n, n))$ time and $O(n)$ space on a random-access machine. We begin in this section by developing a simpler $O(n + m \log^* n)$-time algorithm.

We shall use $\mathcal{T}$ to denote the set of trees defined by the link operations. We maintain the following information for each vertex v: $p_{\mathcal{T}}(v)$, $size_{\mathcal{T}}(v)$, a list of the children of v in $\mathcal{T}$, the child w of v (if any) such that $w \to v$ is a heavy edge, *hp size* (v) (the number of descendants of v on the same heavy path as v), and *light* (v), defined to be true if $p_{\mathcal{T}}(v)$ is defined and $v \to p_{\mathcal{T}}(v)$ is light, and false otherwise. It is easy to update this information in $O(1)$ time per link. Note that a link can cause a formerly heavy edge to become light (but not vice versa) and can create either a heavy or a light edge. We also maintain a data structure [1], [11] that allows us to rapidly compute, for any vertex v, the root $r_{\mathcal{T}}(v)$ of the tree in $\mathcal{T}$ containing v. Updating this data structure during links and carrying out $O(m)$ root calculations requires $O(n + m\alpha(m+n, n))$ time.

Corresponding to $\mathcal{T}$ we maintain a forest $\mathcal{C}$ of compressed trees; $\mathcal{C}$ contains one or more trees representing each tree in $\mathcal{T}$. We build the trees in $\mathcal{C}$ family-by-family in delayed fashion. When a link creates a light edge $v \to p_{\mathcal{T}}(v)$ or causes a formerly heavy edge $v \to p_{\mathcal{T}}(v)$ to become light, we explore the heavy path σ whose apex is v, constructing the set $S = \{w \neq v \mid w \text{ is on } \sigma \text{ or } p_{\mathcal{T}}(w) \text{ is on } \sigma\}$. We then combine the compressed trees whose roots are in S into a single compressed tree; the root of this tree is v and the children of the root are the vertices in S. In general a tree T in $\mathcal{T}$ is represented by several compressed trees; if r is the root of T and σ is the heavy path whose apex is r then $\mathcal{C}$ contains a tree for every vertex v on σ whose single vertex is v, and a tree for every vertex w such that $p_{\mathcal{L}}(w)$ but not w is on σ whose vertices are all the descendants of w in T. (See Fig. 7.)

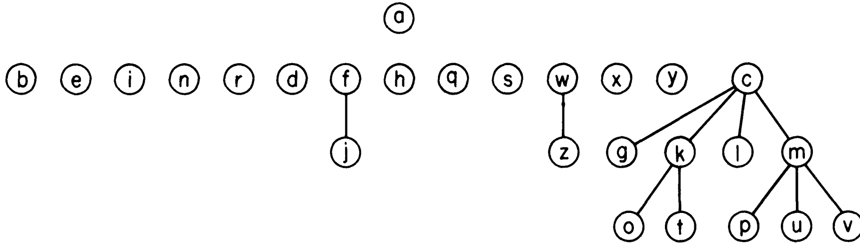


FIG. 7. *Set of compressed trees corresponding to the tree in Fig. 2, for the linking roots problem (see Fig. 3).*

We maintain certain information about the compressed trees. With each vertex v we store $size_{\mathscr{C}}(v)$. With each vertex v such that $p_{\mathscr{C}}(v)$ is defined, we store $p_{\mathscr{C}}(v)$ and *apex* (v). Updating this information during links requires $O(n)$ total time. We also maintain a data structure [1], [11] that allows us to rapidly compute two pieces of information about any vertex v: $d_{\mathscr{C}}(v)$ (the depth of v in the forest $\mathscr{C}$) and $r_{\mathscr{C}}(v)$ (the root of the tree in $\mathscr{C}$ containing v). Maintaining this data structure during links and carrying out $O(m)$ root and depth computations requires $O(n+m\alpha(m+n, n))$ time. We compute nearest common ancestors in $\mathscr{T}$ using the following variant of the method in §§ 3 and 4:

Algorithm to compute $nca_{\mathscr{T}}(v, w)$.

Step 1. Compute $r_{\mathscr{T}}(v)$ and $r_{\mathscr{T}}(w)$. If $r_{\mathscr{T}}(v) \neq r_{\mathscr{T}}(w)$, return a message that v and w are in different trees.

Step 2. Compute $r_{\mathscr{C}}(v)$ and $r_{\mathscr{C}}(w)$. If $r_{\mathscr{C}}(v) = r_{\mathscr{C}}(w)$, go to Step 3. Otherwise (v and w are in different compressed trees), let $v' = r_{\mathscr{C}}(v)$ if *light* $(r_{\mathscr{C}}(v))$ is false, $v' = p_{\mathscr{T}}(r_{\mathscr{C}}(v))$ if *light* $(r_{\mathscr{C}}(v))$ is true. Let $w' = r_{\mathscr{C}}(w)$ if *light* $(r_{\mathscr{C}}(w))$ is false, $w' = p_{\mathscr{T}}(r_{\mathscr{C}}(v))$ if *light* $(r_{\mathscr{C}}(w))$ is true. Return as $nca_{\mathscr{T}}(v, w)$ whichever of v' and w' has the larger value of *hp size.*

Step 3 (v and w are in the same compressed tree). Compute $u = nca_{\mathscr{C}}(v, w)$. If $u = v$ or $u = w$, return u as $nca_{\mathscr{T}}(v, w)$. Otherwise, compute $d_{\mathscr{C}}(u)$.

Step 4. Compute the ancestor v' of v in $\mathscr{C}$ whose depth is $d_{\mathscr{C}}(u)+1$. If *apex* $(v') = u$, let $v'' = v'$. Otherwise let $v'' = p_{\mathscr{T}}(v')$.

Step 5. Compute the ancestor w' of w in $\mathscr{C}$ whose depth is $d_{\mathscr{C}}(u)+1$. If *apex* $(w') = u$, let $w'' = w'$. Otherwise let $w'' = p_{\mathscr{T}}(w')$.

Step 6. Return as $nca_{\mathscr{T}}(v, w)$ whichever of v'' and w'' has the larger value of *hp size.*

This method requires $O(n+m\alpha(m+n, n))$ time plus time to solve m *nca* problems on $\mathscr{C}$ and $2m$ depth problems on $\mathscr{C}$. We shall postpone a discussion of how to solve depth problems on $\mathscr{C}$, as this is the hardest part of the algorithm. To solve *nca* problems on $\mathscr{C}$, we use the method of § 5. That is, we represent the forest $\mathscr{C}$ of compressed trees by a forest $\mathscr{B}$ of balanced binary trees. If C is a tree in $\mathscr{C}$, $\mathscr{B}$ contains a tree B whose green vertices are the same as those of C. B is constructed exactly as in § 5; each time we create a new family in $\mathscr{C}$ we combine the corresponding trees in $\mathscr{B}$, using procedure *binarize* if the set of trees has size three or greater. For each vertex v, we maintain $size_{\mathscr{B}}(v)$. Constructing $\mathscr{B}$ and updating the size information during links takes $O(n)$ total time.

As in § 5, we solve the *nca* problem on $\mathscr{B}$ by solving the *nca* depth problem and the depth problem on $\mathscr{B}$. To solve the *nca* depth problem on $\mathscr{B}$ we embed $\mathscr{B}$ in a complete binary tree $\mathscr{B}'$ and use direct calculation as described in § 3; all we need to

know for each vertex in $\mathcal{B}$ is its symmetric-order number and height (as a vertex in $\mathcal{B}'$). These numbers require updating as $\mathcal{B}$ is built. Let u be a vertex in $\mathcal{B}$ with left child v and right child w. When edges $v \to u$ and $w \to u$ are added to $\mathcal{B}$, we must update the numbers of the descendants of u as follows:

Let $h_1 = height_{\mathcal{B}'}(v)$, $h_2 = height_{\mathcal{B}'}(w)$, and $h = \max\{h_1, h_2\}$. Redefine

$$height_{\mathcal{B}'}(x) = \begin{cases} height_{\mathcal{B}'}(x) - h_1 + h & \text{if } x \text{ is a descendant of } v, \\ h+1 & \text{if } x = u, \\ height_{\mathcal{B}'}(x) - h_2 + h & \text{if } x \text{ is a descendant of } w; \end{cases}$$

$$sym_{\mathcal{B}'}(x) = \begin{cases} 2^{h-h_1}\, sym_{\mathcal{B}'}(x) & \text{if } x \text{ is a descendant of } v, \\ 2^{h+1} & \text{if } x = u, \\ 2^{h-h_2}\, sym_{\mathcal{B}'}(x) + 2^{h+1} & \text{if } x \text{ is a descendant of } w. \end{cases}$$

We can use the data structure of [11, § 3] to maintain the numbers implicitly. The total time to maintain the data structure during links and compute $O(m)$ numbers is $O(n + m\alpha(m+n, n))$. Given the numbers, solving an *nca* depth problem on $\mathcal{B}$ requires $O(1)$ time.

We come now to the last and hardest part of the method: solving depth problems on $\mathcal{B}$ and $\mathcal{C}$. Since we use the same method for both forests, we shall discuss only $\mathcal{C}$. To solve the depth problem on $\mathcal{C}$, we divide $\mathcal{C}$ into $O(\log^* n)$ plies. As in § 4, define $rank(v) = \lfloor \lg(size_{\mathcal{C}}(v)) \rfloor$. Define the function $f(i)$ by the recursion

$$f(1) = 8;$$

$$f(i) = 2^{f(i-1)/2} \text{ for any integer } i \geqq 2.$$

Define the ply i of vertex v to be the minimum value of i such that $f(i) > rank(v)$. It is easy to compute the ply of a vertex at the moment its size is determined; this takes $O(1)$ time per vertex for a total of $O(n)$ time. (We precompute a table that for any integer $x \in [0, \lfloor \lg n \rfloor]$ gives the minimum i such that $f(i) > x$.) The total number of plies is $O(\log^* n)$.

With each vertex v, we store not only its ply but several other values: the vertex $a(v)$ that is the shallowest ancestor of v in the same ply as v, a list of the children of v that are in the same ply as v, and an array, indexed by depth, of all the ancestors of v in the same ply as v. Each time we add a new family to $\mathcal{C}$, we update this information by performing a search from the root r of the new tree, using the lists of children to reach all descendants of r in the same ply as r. The total amount of time spent updating this information, and the total size of the ancestor arrays, is bounded by a constant times the following sum:

$$8n + \sum_{i=1}^{\infty} \frac{f(i+1)n}{2^{f(i)}} = 8n + \sum_{i=1}^{\infty} \frac{n}{2^{f(i)/2}} = O(n).$$

To solve depth problems on $\mathcal{C}$, we use the same method as in § 4.

Algorithm to compute the ancestor of vertex v in $\mathcal{C}$ whose depth is d. Repeat the following step until a vertex is returned:

General step. If $d \in [d_{\mathcal{C}}(a(v)), d_{\mathcal{C}}(v)]$, look up in v's ancestor array the ancestor whose depth is d. Otherwise, replace v by $p_{\mathcal{C}}(a(v))$.

This method requires $O(1)$ time per execution of the general step, or $O(\log^* n)$ time per depth problem. (Note that if we store with each vertex v the current size of v's ancestor array, we can compute $d_{\mathcal{C}}(a(v))$ and $d_{\mathcal{C}}(p_{\mathcal{C}}(a(v)))$ from $d_{\mathcal{C}}(v)$ in $O(1)$ time.)

Using this method on both $\mathscr{B}$ and $\mathscr{C}$, we obtain a total time bound of $O(n+m\log^* n)$ for the linking roots problem; solving the depth problem is the most time-consuming part of the algorithm.

7. A faster algorithm for the linking roots problem. In order to obtain a faster algorithm for the linking roots problem, we must improve our method for solving depth problems on $\mathscr{B}$ and $\mathscr{C}$. Our method in § 6 was based on the representation of plies used in § 4 for plies two and three. By including the method used in § 4 for ply one and defining the plies more dynamically, we obtain an $O(n+m\alpha(m+n, n))$-time method. As in § 6, we shall discuss the depth problem only for $\mathscr{C}$; we use the same method for $\mathscr{B}$. We shall assume that n, the number of vertices, and m, the number of *nca* queries, are known in advance. Working knowledge of the analysis of the disjoint set union algorithm [10], [13] will help the reader in what follows.

We divide $\mathscr{C}$ into three super-plies exactly as in § 4: super-ply one consists of vertices with rank less than $\lg^{(3)} n$, super-ply two consists of vertices with rank between $\lg^{(3)} n$ and $\lg^{(2)} n-1$ (inclusive), and super-ply three consists of vertices with rank $\lg^{(2)} n$ or greater. We maintain super-plies two and three exactly as we maintained the plies in § 6. The results of § 4 imply that updating super-plies two and three requires $O(n)$ total time.

We divide the trees of super-ply one into a number of subtrees whose definition is based on Ackermann's function. For integers $i, j \geqq 0$ define $A(i, j)$ by

$A(0, j)=j$ for $j \geqq 0$;
$A(1, j)=2^j$ for $j \geqq 0$;
$A(i, 0)=A(i-1, 1)$ for $i \geqq 2$ (also valid for $i=1$);
$A(i, j)=A(i-1, A(i, j-1))$ for $i \geqq 2, j \geqq 1$.

Let $\alpha(m+n, n)=\min\{i \geqq 1 \mid A(i, \lfloor (m+n)/n \rfloor)>\lg n\}$.

Let v be a vertex that is in super-ply one and such that $p_{\mathscr{C}}(v)$ exists. We define the *level* of v to be the maximum value of $i \in [0, \alpha(m+n, n)]$ such that, for some j, $rank\,(v)<A(i, j) \leqq rank\,(p_{\mathscr{C}}(v))$; we define the *position* of v to be the minimum such j. We can compute the level and position of a vertex when its parent is defined in $O(1)$ time by using a small precomputed table.

We maintain super-ply one as a collection of subtrees. Each vertex in super-ply one for which $p_{\mathscr{C}}(v)$ is undefined is in a single-vertex tree by itself. When a vertex v in ply one of positive rank has its parent defined, we build a new subtree S_v with root v. S_v includes all descendants of v with ranks in the range $[x, A(i, j)-1]$, where i is the level of v, j is the position of v, and x is defined as follows: if there is an integer $A(i', j')$ such that $i<i' \leqq \alpha(m+n, n)$, $j' \geqq 1$, and $A(i', j')<A(i, j)$ then x is the maximum such integer; if there is no such integer let x be the maximum integer less than the rank of v of the form $A(i', 0)$ for some $0 \leqq i' \leqq i$. We can compute x in $O(1)$ time by table look-up. The definitions of S_v and Ackermann's function guarantee that the vertex set of the new subtree S_v is the union of the vertex sets of one or more older subtrees; thus each vertex is only in one subtree at a time.

This partitioning into subtrees has the following effect. Let r be a shallowest vertex in super-ply one and let v be a leaf descendant of r. The subtrees partition the path from v to r as follows. Let l be the maximum level of any vertex along the path from v to r and let x be the last vertex along this path of level l. The subtrees partition the path from v to x into segments $(v=v_1=x_1)$, $(v_2, \cdots, x_2), \cdots, (v_l, \cdots, x_l)$, $(v_{l+1}, \cdots, x), (y_{l-1}, \cdots, w_{l-1}), (y_{l-2}, \cdots, w_{l-2}), \cdots, (y_0, w_0), (r)$, where x_i for $i \in [1, l]$ is the first vertex on the path of level i preceding all vertices of higher level and w_i for $i \in [0, l]$ is the last vertex on the path of level i succeeding all vertices of higher

level. Some of the v_i's and w_i's may be undefined; the corresponding segments of the path are empty. It is also possible that $(v_l, \cdots, x_l) = (v_{l+1}, \cdots, x)$. When $p_{\mathscr{C}}(r)$ becomes defined, subtrees are combined to form a segment (y_i, r), where i is the level of r; if $i = l$, the new segment is $(v_{l+1}, \cdots, r)$, and if $i > l$, the new segment is $(v_l, \cdots, r)$.

To facilitate the construction of the subtrees, we store with each vertex in super-ply one a list of its children, in order by rank. Then we can determine the vertex set of S_v in $O(|S_v|)$ time by searching from v. If we represent each list of children as a list of buckets, each bucket containing the children of a given rank, then inserting a new child requires time proportional to its rank, and Lemma 8 implies that the total time for constructing these lists of children is $O(\sum_{i=0}^{\infty} (i+1)n/2^i) = O(n)$.

We represent the subtrees of super-ply one exactly as we represented the trees of ply one in § 4. We also store with each vertex in such a subtree a pointer to the root of the subtree. If S_v is one of these subtrees, constructing the representation of S_v requires $O(|S_v|)$ time.

We solve depth problems in $\mathscr{C}$ exactly as we solved them in § 6. The following argument shows that this method solves a single depth problem in $O(\alpha(m+n, n))$ time. Let v be a vertex, d a depth such that $d \leqq d_{\mathscr{C}}(v)$, and w the ancestor of v whose depth is d. With each iteration of the general step, v is replaced by an ancestor that is either in a new subtree of super-ply one or in a new super-ply, until eventually w is reached. Each execution of the general step requires $O(1)$ time. Thus it suffices to bound the number of subtrees in super-ply one encountered during this process. Let r be the shallowest ancestor of (the original) v in super-ply one, let l be the maximum level of any vertex along the path from v to r, and let x be the last vertex along this path of level l. Before x is reached, every iteration of the general step (except possibly the one that reaches x) causes the level of the current vertex to increase by at least one; after x is reached but before r is reached, every iteration of the general step causes the level of the current vertex to decrease by at least one. Thus there are $O(\alpha(m+n, n))$ iterations of the general step.

It remains for us to bound the total time spent constructing subtrees in super-ply one. For each vertex v, this time is $O(1)$ for each subtree in which v is placed. Our analysis is almost identical to the analysis of the disjoint set union algorithm [10], [13]. For $i \in [0, \alpha(m+n, n)+1]$, we define a multiple partition on ranks. The blocks of the level i partition (see Fig. 8) are given by

$$\textit{block}\ (i, j) = [A(i, j), A(i, j+1)-1] \text{ for } i \in [0, \alpha(m+n, n)], j \geqq 0;$$
$$\textit{block}\ (i, 0) = [0, \lfloor \lg^{(3)} n \rfloor] \text{ for } i = \alpha(m+n, n)+1.$$

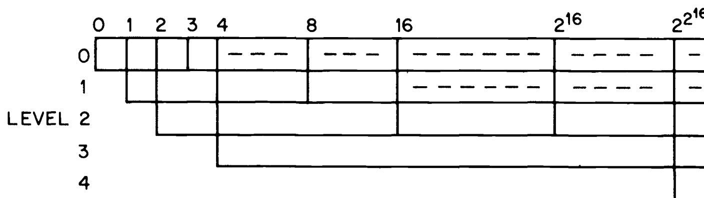


FIG. 8. *Multiple partition for analysis of subtree construction. Some block boundaries in levels zero and one are omitted.*

Note. For $i \in [0, \alpha(m+n, n)]$, the level i partition does not include ranks smaller than $A(i, 0)$. Thus not all ranks are in every partition. □

We define n_{ij} to be number of vertices with rank in *block* (i, j) and b_{ij} to be the number of level $i-1$ blocks whose intersection with *block* (i, j) is nonempty ($i \geqq 1$). As the algorithm proceeds, we define the *effective level* of a vertex v in a nontrivial subtree of super-ply one to be the minimum value of i such that *rank* (v) and *rank* $(p_{\mathscr{C}}(r(v)))$ are in the same block of the level i partition, where $r(v)$ is the root of the subtree containing v. As v is placed in larger-and-larger subtrees, its effective level increases from a minimum of one up to a maximum of $\alpha(m+n, n)+1$.

Consider a vertex v in *block* (i, j). We would like to bound the number of different subtrees that can contain v while v is at effective level i. First consider the case $i \leqq \alpha(m+n, n)$. When v is the first at level i, *rank* (v) and *rank* $(p_{\mathscr{C}}(r(v)))$ are in different level $i-1$ blocks. Subsequently, each time v is placed in a new subtree, *rank* $(p_{\mathscr{C}}(r(v)))$ moves to a new level $i-1$ block. Thus the number of subtrees that can contain v while v is at effective level i is at most $b_{ij}-1$. Second consider the case $i=\alpha(m+n, n)+1$. If $v \in$ *block* $(i', 0)$ for some $i' < \alpha(m+n, n)$, then v is only in one subtree while at effective level i. If $v \in$ *block* (i, j') for some $j' \geqq 0$, v while at effective level i is in at most as many subtrees as there are level-$\alpha(m+n, n)$ blocks, namely $\lfloor (m+n)/n \rfloor - 1$.

By Lemma 8, the number of vertices in *block*(i, j) is at most $2n/2^{A(i,j)}$. The following sum gives an upper bound, to within a constant factor, on the time spend building subtrees in super-ply one:

$$\sum_{i=1}^{\alpha(m+n,n)} \sum_{j=0}^{\infty} b_{ij} 2n/2^{A(i,j)} + n \lfloor (m+n)/n \rfloor.$$

Now we must estimate b_{ij}. For $i=1$, $j \geqq 0$, $A(i, j+1) - A(i, j) = 2^j = A(i, j)$, which means $b_{ij} < A(i, j)$. For $i \geqq 2$, $j \geqq 0$, $A(i, j+1) = A(i-1, A(i, j))$, which means $b_{ij} < A(i, j)$ in this case also. Thus the time spent building subtrees in super-ply one is at most a constant times the following estimate:

$$\begin{aligned} \sum_{i=1}^{\alpha(m+n,n)} \sum_{j=0}^{\infty} 2nA(i, j)/2^{A(i,j)} + O(m+n) &\leqq \sum_{i=1}^{\alpha(m+n,n)} O(nA(i, 0)/2^{A(i,0)}) + O(m+n) \\ &= O(m+n). \end{aligned}$$

Thus the entire algorithm solves the linking roots problem in $O(n + m\alpha(m+n, n))$ time.

This algorithm has the disadvantage that both m and n must be known ahead of time. We leave to the reader the easy exercise of modifying the algorithm to avoid this. (The idea is to re-estimate m and n each time the actual value of either grows by a factor of two.)

Appendix. Tree terminology. Throughout this paper we consider only rooted trees. A rooted tree T consists of a *vertex set* V, a *root* $r \in V$, and a mapping $p(v)$: $V - \{r\} \to V$ such that, for every vertex v, there is an integer $i \geqq 0$ such that $p^i(v) = r$, where $p^i(v)$ is defined inductively by $p^0(v) = 0$, $p^{i+1}(v) = p(p^i(v))$. For $v \in V - \{r\}$, $p(v)$ is the *parent* of v; v is a *child* of $p(v)$. The *edges* of T are the ordered vertex pairs $v \to p_T(v)$ for $v \in V - \{r\}$. A *leaf* is a vertex with no children. If $p^i(v) = w$ for some $i \geqq 0$, v is a *descendant* of w and w is an *ancestor* of v. Two vertices are *unrelated* if neither is an ancestor of the other. A *path* in T is a sequence of vertices $v, p(v), p^2(v), \cdots, p^k(v)$; the *length* of this path is k. The *depth* of a vertex v is the length of the path from v to r; the *height* of v is the length of the longest path from a leaf to v. The depth (and height) of T is the length of the longest path in T. We use $|T|$ to

denote the number of vertices in T. The *nearest common ancestor* $nca(x,y)$ of two vertices x and y is the vertex of greatest depth that is an ancestor of both x and y.

A *forest* $\mathscr{T}$ is a collection of vertex-disjoint trees. (We use capital italic letters to denote trees and capital script letters to denote forests.) When discussing parameters associated with several trees or forests, we use the names of the trees or forests as subscripts to distinguish the parameters. For example, $p_T(v)$ is the parent of v in T.

Acknowledgment. The first author's research was done partly while he was a graduate student at the University of California, Irvine. He would like to thank his thesis advisor, George Lueker, for his patience, encouragement, and numerous conversations regarding preliminary versions of the results presented here. Both authors thank the referees for their exceptionally thorough job in catching and correcting a number of minor errors in the original manuscript.

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] ——, *On finding lowest common ancestors in trees*, this Journal, 5 (1976), pp. 115–132.

[3] M. L. FREDMAN, *Two applications of a probabilistic search technique: sorting $X + Y$ and building balanced search trees*, Proc. Seventh ACM Symposiun on Theory of Computing, 1975, pp. 240–244.

[4] H. N. GABOW AND R. E. TARJAN, *A linear-time algorithm for a special case of disjoint set union*, J. Comp. Sys. Sci., to appear; also Proc. Fifteenth ACM Symposium on Theory of Computing, 1983, pp. 246–251.

[5] D. HAREL, *A linear time time algorithm for the lowest common ancestors problem*, Proc. 21st IEEE Symposium on Foundations of Computer Science, 1980, pp. 308–319.

[6] D. E. KNUTH, *The Art of Computer Programming, Volume* I: *Fundamental Algorithms*, 2nd ed., Addison-Wesley, Reading, MA, 1973.

[7] D. MAIER, *An efficient method for storing ancestor information in trees*, this Journal, 8 (1979), pp. 599–618.

[8] A. SCHÖNHAGE, *Storage modification machines*, this Journal, 9 (1980), pp. 490–508.

[9] D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 26 (1983), pp. 362–391.

[10] R. E. TARJAN, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215–225.

[11] ———, *Applications of path compression on balanced trees*, J. Assoc. Comput. Mach., 26 (1979), pp. 690–715.

[12] ———, *A class of algorithms which require non-linear time to maintain disjoint sets*, J. Comput. System Sci., 18 (1979), pp. 110–127.

[13] R. E. TARJAN AND J. VAN LEEUWEN, *Worst-case analysis of set union algorithms*, J. Assoc. Comput. Mach., to appear.

[14] J. VAN LEEUWEN, *Finding lowest common ancestors in less than logarithmic time*, unpublished report, 1976.

SIAM J. COMPUT.
Vol. 13, No. 2, May 1984

© 1984 Society for Industrial and Applied Mathematics
010

THE SPECTRA OF FIRST-ORDER SENTENCES AND COMPUTATIONAL COMPLEXITY*

ETIENNE GRANDJEAN†

Abstract. The spectrum of a first-order sentence is the set of cardinalities of its finite models. We refine the well-known equality between the class of spectra and the class of sets (of positive integers) accepted by nondeterministic Turing machines in polynomial time. Let Sp $(d\forall)$ denote the class of spectra of sentences with d universal quantifiers. For any integer $d \geqq 2$ and each set of positive integers, A, we obtain:

$$A \in \text{NTIME}(n^d) \rightarrow A \in \text{Sp}(d\forall) \rightarrow A \in \text{NTIME}(n^d(\log n)^2).$$

Further the first implication holds even if we use multidimensional nondeterministic Turing machines. These results hold similarly for generalized spectra. As a consequence, we obtain a simplified proof of a hierarchy result of P. Pudlák about (generalized) spectra. We also prove that the set of primes is the spectrum of a certain sentence with only one variable.

Key words. first-order sentences, spectrum, generalized spectrum, computational complexity, nondeterministic Turing machine

Introduction. The spectrum of a first-order sentence is the set of cardinalities of its finite models. If instead of cardinalities of models, we conserve some relations and functions of the models, then we obtain a generalized spectrum. More precisely, let φ be a first-order sentence with relation and function symbols $U_1, \cdots, U_k, V_1, \cdots, V_p$; the generalized spectrum of φ is the set of finite structures $\mathcal{M}$ of type $\{U_1, \cdots, U_k\}$ which have an expansion $\langle \mathcal{M}, V_1, \cdots, V_p \rangle$ satisfying φ.

There are equivalence between certain model-theoretic concepts such as (generalized) spectra and complexity classes. Jones and Selman [11] proved that if A is a set of positive integers,

(a) A is a spectrum iff $A \in \bigcup_d \text{NTIME}(n^d)$.

(n represents the input integer.) Similarly, Fagin [5] proved that if G is an isomorphism invariant set of structures of a given type,

(b) G is a generalized spectrum iff $G \in \bigcup_d \text{NTIME}(m^d)$.

(m represents the cardinality of the input structure.) More recently, Immerman [10] gave purely logic characterizations of the classes P and PSPACE.

In the present paper we adopt the philosophy expressed by N. Immerman in [9], [10] and J. Lynch in [13]; that is, logical sentences act like automata. Let $\mathcal{P}$ be a property (of integers, of graphs, $\cdots$). According to our choice of a logical or a computational viewpoint, there are two kinds of complexity for $\mathcal{P}$:

(*) the "complexity" of the sentences which characterize property $\mathcal{P}$;

(**) the computational complexity of the automata which recognize property $\mathcal{P}$.

Connections between complexities (*) and (**) allow one to translate some automata-theoretic results into model-theoretic results: Pudlák [15] uses Cook's hierarchy theorem [4] in an essential manner to prove that there is a strict hierarchy of generalized spectra depending on the number of quantifiers. Conversely, there are potential translations of model-theoretic results into computational ones. For example, by equivalence (a), if there is a spectrum whose complement is not a spectrum, then $P \neq NP$. However, we do not know any proved computational result whose proof uses model-theoretic arguments in an essential manner: we think the reason is that automata and their computations are more "supple" concepts than sentences and their models.

* Received by the editors February 25, 1981, and in revised form December 3, 1982.

† 3 rue de Sévigné, 69003 Lyon, France. Now at Université Claude Bernard–Lyon 1, 69622 Villeurbanne Cedex, France.

The logical concepts that we investigate in this paper are exclusively spectra and generalized spectra which are sets of directed graphs. We justify this last restriction by the fact that results for directed graphs can be easily generalized to other types of structures, and that most natural problems of structures concern (directed) graphs.

We think that there are two natural complexity measures of the (generalized) spectrum of a sentence:

$(*_1)$ the maximum arity of the relation and function symbols of the sentence;

$(*_2)$ the number of quantifiers, or, equivalently , of universal quantifiers of the sentence.

Concerning the relationship between complexities $(*_1)$ and $(**)$, there is:

THEOREM 0.1 (J. Lynch [13]). *If a set, A, of positive integers belongs to* NTIME (n^d) *for an integer $d \geqq 2$, then A is the spectrum of a sentence with relation symbols of arity at most d and without function symbols. (The same result holds for generalized spectra.)*

This is a nice result because we easily see that any improvement of it would imply an improvement of the inclusion NTIME $(n^d) \subseteq$ DSPACE (n^d). However, we do not know any kind of converse: for example, if a sentence has only binary relation symbols, we do not know any fixed polynomial time upper bound for its spectrum. (See [6] for more details about the hypothetical hierarchy of spectra depending on arity of relation symbols.)

We improve a theorem of Pudlák [15] which connects complexities $(*_2)$ and $(**)$ in both directions. Let Sp $(d\forall)$ (resp. GenSp $(d\forall)$) denote the class of spectra (resp. generalized spectra) of sentences with at most d universal quantifiers.

THEOREM 0.2 (Pudlák). *Let G be an isomorphism invariant set of directed graphs. Then, for all integers $d \geqq 2$:*

$$G \in \text{GenSp}\,(d\forall) \textit{ implies } G \in \text{NTIME}\,(m^{3d}),$$

$$G \in \text{NTIME}\,(m^{d}) \textit{ implies } G \in \text{GenSp}\,(2d\forall).$$

Pudlák also states the following translational lemma of model theory:

LEMMA 0.3 (Pudlák). *For all integers $d \geqq 2$, $e \geqq 1$,*

$$\text{GenSp}\,(d\forall) = \text{GenSp}\,(d+1\forall) \textit{ implies } \text{GenSp}\,(ed\forall) = \text{GenSp}\,(e(d+1)\forall).$$

From these results and from Cook's hierarchy theorem, Pudlák deduces the nice hierarchy result mentioned above which can be reformulated as follows:

COROLLARY 0.4 (Pudlák). *For all integers $d \geqq 0$,*

$$\text{GenSp}\,(d\forall) \subsetneqq \text{GenSp}\,(d+1\forall).$$

Unfortunately, Pudlák's paper [15] does not provide the proofs of the results that he states. Therefore the present paper includes an explicit proof of Pudlák's hierarchy result. However, its main merit is that it considerably improves the connections Pudlák states between GenSp $(d\forall)$ and NTIME (–) since we prove:

THEOREM 0.5. *Let A be a set of positive integers and G be an isomorphism invariant set of directed graphs. Then for all integers $d \geqq 2$:*

(i) $A \in \text{NTIME}\,(n^d) \to A \in \text{Sp}\,(d\forall) \to A \in \text{NTIME}\,(n^d(\log n)^2)$;

(ii) $G \in \text{NTIME}\,(m^d) \to G \in \text{GenSp}\,(d\forall) \to G \in \text{NTIME}\,(m^d(\log m)^2)$.

These results are interesting for two reasons:

(1) They state that the polynomial degree of the (nondeterministic) time complexity of a property $\mathcal{P}$ is almost equal to the number of universal quantifiers required to express $\mathcal{P}$.

(2) They allow one to prove Pudlák's hierarchy result (and the similar result for spectra) without any model-theoretic lemma, by an immediate translation of the nondeterministic time hierarchy theorem (Cook [4], Seiferas et al. [17]).

Note that the second implication of (i) is essentially proved by H. Lewis [12] in a different context: he investigates the complexity of the satisfiability problem for classes of quantificational sentences.

These results are optimal in the following sense: each first implication of (i) and (ii) holds even if the nondeterministic Turing machine (NTM) is multidimensional; so any improvement of one of the above implications would improve the known simulation of a multidimensional $T(n)$ time-bounded NTM by a (one-dimensional) $T(n)\cdot(\log T(n))^2$ time-bounded NTM.

Notice that Immerman [9], [10] also investigates the number of quantifiers as a complexity measure: using connections between first-order expressibility and computational complexity, he hopes to translate into computational complexity, some lower bounds he obtains for first-order expressibility of "natural" properties of graphs. However, his results and methods are quite different from ours because he characterizes a property, not by only *one* sentence, but by a *uniform sequence* of sentences. (Of course, by the uniformity condition, such a sequence can be regarded in a certain sense as a unique sentence.) As a consequence, he no longer needs additional relation and function symbols, but only a successor relation. Immerman's opinion in [9] is that "it is difficult to show lower bounds for the expressibility of (existential) second-order sentences" and that "first-order sentences mimic computations much more closely." In fact, from our results and from the time hierarchy theorem, it is immediate that there is a spectrum in Sp $(d\forall)$ which cannot be accepted by an NTM in time less than n^d. However, Immerman is right in a certain sense: we are not able to prove a nontrivial lower bound for any naturally defined (generalized) spectrum.

Our paper includes the following sections. Notation and definitions are given in § 1. In § 2, we give two arguments for the naturalness of the measure (Gen)Sp $(d\forall)$: first that it is preserved under intersection and union, secondly that it is equivalent to other complexity measures such as quantifier depth. We also prove the previously mentioned upper bound of spectra.

The announced lower bound for spectra is proved in § 3. Besides the usual "folding" technique [5], [11], [13] for encoding the time units and the tape cells of a computation of a NTM, the proof essentially uses a "numbering" of the ordered pairs $(H(t), t)$, where $H(t)$ is the position of a tape head at instant t; informally, this numbering, N, is such that if $N(x)=(H(t), t)$, and if t' is the first time after t such that $H(t)=H(t')$, then $N(x+1)=(H(t'), t')$.

Lastly, § 4 presents two corollaries of the previous results. First is the hierarchy result of the classes (Gen)Sp $(d\forall)$. The second is a rather surprising result: The set of primes and most "natural" sets of positive integers are spectra of certain sentences with only one variable.

1. Preliminaries.

1.1. Preliminaries in logic. We will use the usual notation and definitions in first-order logic and model theory (see [3, Chap. 1], for example). In particular, our formulas include the equality symbol $=$, and relation and function symbols.

The *arity* of a relation or function symbol is a nonnegative integer; in fact, a 0-ary relation (resp. function) symbol is a proposition (resp. an (individual) constant) symbol.

Our logical connectives are exlusively $\vee$, $\wedge$, $\neg$, interpreted as "or," "and," "not," respectively. The existential and universal quantifiers $\exists$ and $\forall$ are interpreted as "there exists" and "for all," respectively.

(Individual) variables are called x, y, z, t with or without subscripts or primes. The metavariable v (with or without subscripts or primes) will denote any variable.

A *term* is an expression constructed from variables and function symbols in the usual way. An *atomic formula* is of the form $\tau_1 = \tau_2$ or $R(\tau_1, \cdots, \tau_r)$, where τ_i is a term and R is an r-ary relation symbol. A (*first-order*) *formula* is built out of atomic formulas in the usual way, using $\vee$, $\wedge$, $\neg$, $\exists$, $\forall$. A *signed atomic* formula is an atomic formula or its negation.

We suppose familiarity with the notions of subformula, of (existentially or universally) quantified variable, of free occurrence of a variable, and of free variable. A (*first-order*) *sentence* is a formula all of whose variables are quantified. We use $\varphi(v_1, \cdots, v_k)$ to denote a formula φ whose free variables form a subset of $\{v_1, \cdots, v_k\}$.

A *prenex sentence* is a sentence φ of the form

$$Q_1 v_1 \cdots Q_k v_k \psi(v_1, \cdots, v_k)$$

where ψ is a quantifier-free formula and $Q_1, \cdots, Q_k$ are quantifiers; $Q_1 v_1 \cdots Q_k v_k$ and ψ are respectively the *prefix* and the *matrix* of φ. A quantifier-free formula is in *disjunctive normal form* if it is a disjunction of conjunctions of signed atomic formulas.

Sometimes we will use the following abbreviations: $v \neq v'$ for $\neg v = v'$; $\varphi \rightarrow \psi$ for $\neg\varphi \vee \psi$; $\varphi \leftrightarrow \psi$ for $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$. The conjunction of the indexed formulas φ_i, for $i \in J$ and J a finite set, will be denoted $\bigwedge_{i \in J} \varphi_i$, and similarly for the disjunction. Let $\bar{v}_k$ and $Q\bar{v}_k$ (where Q is a quantifier) abbreviate the k-tuple $v_1, \cdots, v_k$ and the string $Qv_1 \cdots Qv_k$, respectively. (More generally, let $\bar{a}_k$ denote a k-tuple of elements $a_1, \cdots, a_k$ in a given set.)

A *type* $\mathcal{T}$ is a finite set of relation and function symbols $\{V_1, \cdots, V_k\}$. The *arity* of $\mathcal{T}$ is the maximum arity of $V_1, \cdots, V_k$. A formula is *of type* $\mathcal{T}$ if all its relation and function symbols are in $\mathcal{T}$.

A *structure* $\mathcal{M} = \langle D, V_1, \cdots, V_k \rangle$ *of type* $\mathcal{T}$ consists of a nonempty set D called the *domain* of $\mathcal{M}$ (denoted $D(\mathcal{M})$) and for each r-ary relation (resp. function) symbol of $\mathcal{T}$, an interpretation, i.e., an r-ary relation (resp. function) on D. If $V_{k+1}, \cdots, V_p$ are other relations and functions on D, the structure $\langle D, V_1, \cdots, V_k, V_{k+1}, \cdots, V_p \rangle$ is called an *expansion* of $\mathcal{M}$ and is denoted $\langle \mathcal{M}, V_{k+1}, \cdots, V_p \rangle$. For convenience, our notation does not distinguish between a relation or function symbol and its interpretation. The cardinality of a structure is the cardinality of its domain. A *finite structure* is a structure of finite cardinality.

Let φ and $\mathcal{M}$ be respectively a sentence and a structure of type $\mathcal{T}$. We put $\mathcal{M} \models \varphi$, and we say that $\mathcal{M}$ is a *model* of φ, to mean that φ becomes a true assertion when each logical symbol $\vee$, $\wedge$, $\neg$, $\exists$, $\forall$, $=$ is given its usual meaning and each relation (resp. function) symbol is given its interpretation in the structure $\mathcal{M}$.

Let D be a nonempty finite set and $\mathcal{T}$ be a type; we will regard the elements of D as constant symbols; let $\mathcal{T}_D = \mathcal{T} \cup D$ be the type $\mathcal{T}$ enlarged by these constant symbols. Any structure of type $\mathcal{T}$ on the domain D is identified with its expansion of type $\mathcal{T}_D$ where constant symbols $a \in D$ are interpreted as themselves. Let $\varphi(\bar{v}_k)$ be a formula of type $\mathcal{T}$ and $\bar{a}_k$ be a k-tuple of elements in D; $\varphi(\bar{a}_k)$ will denote the

sentence of type $\mathscr{T}_D$ constructed from $\varphi(\bar{v}_k)$ be replacing each free occurrence of v_i by a_i. If there is a structure, $\mathscr{M}$, of type $\mathscr{T}$ on the domain D, such that $\mathscr{M} \models \varphi(\bar{a}_k)$, we will say that $\varphi(\bar{a}_k)$ is *satisfiable in* D, or, in case the domain D is implicit, *satisfiable*.

Existential second-order sentences are expressions of the form $\exists V_1 \cdots \exists V_p \varphi$ where φ is a first-order sentence and $V_1, \cdots, V_p$ are among the relation and function symbols of φ. (Unless stated otherwise, formula and sentence will mean first-order formula and first-order sentence.) Let $\mathscr{M}$ be a structure of type $\mathscr{T}$ and ψ be a sentence of type $\mathscr{T} \cup \{V_1, \cdots, V_p\}$, where $V_1, \cdots, V_p$ are symbols not in $\mathscr{T}$. Then we put

$$\mathscr{M} \models \exists V_1 \cdots \exists V_p \psi$$

to mean that $\mathscr{M}$ has an expansion $\mathscr{M}'$ of type $\mathscr{T} \cup \{V_1, \cdots, V_p\}$ such that $\mathscr{M}' \models \psi$. (Of course, the type of $\exists V_1 \cdots \exists V_p \psi$ is $\mathscr{T}$.)

Two (first-order or second-order) sentences of type $\mathscr{T}$, φ and ψ, are (*semantically*) *equivalent* if for each structure $\mathscr{M}$ of type $\mathscr{T}$,

$$\mathscr{M} \models \varphi \quad \text{iff} \quad \mathscr{M} \models \psi.$$

We similarly define the (*semantical*) *equivalence* of formulas $\varphi(\bar{v}_k)$ and $\psi(\bar{v}_k)$.

In the following, for each integer $n > 0$, D_n will denote the set $\{0, 1, \cdots, n-1\}$.

The *spectrum* of a sentence φ, denoted Sp (φ), is the set of cardinalities of its finite models, or, equivalently, the set of integers $n > 0$ such that

$$\langle D_n \rangle \models \exists V_1 \cdots \exists V_p \varphi,$$

if φ is of type $\{V_1, \cdots, V_p\}$.

A *directed graph* or, in short, a *graph*, will be a finite structure $\mathscr{G} = \langle D_m, \mathbf{R} \rangle$, where $\mathbf{R}$ is a binary relation. (In graph-theoretic terminology, $\mathscr{G}$ is a labeled directed graph on m vertices labeled $0, 1, \cdots, m-1$.)

The *generalized spectrum* of a sentence φ of type $\{\mathbf{R}, V_1, \cdots, V_p\}$ where $\mathbf{R}$ is a specified binary relation symbol, is the set of directed graphs $\mathscr{G} = \langle D_m, \mathbf{R} \rangle$ such that

$$\mathscr{G} \models \exists V_1 \cdots \exists V_p \varphi.$$

It is denoted GenSp (φ).

We will investigate the number of universal quantifiers as a complexity measure. There is a little difficulty: a universal (resp. existential) quantifier in the scopes of an odd number of negation signs must be treated as an existential (resp. universal) quantifier. So, to make sense, we always assume in this paper that no quantifier occurs in the scope of a negation sign. (Clearly, each formula is equivalent to a formula of the requisite form.)

Let Sp $(d\forall)$ (resp. GenSp $(d\forall)$) denote the class of spectra (resp. generalized spectra) of sentences with at most d universal quantifiers.

The *depth* (resp. *universal depth*) of a formula φ, denoted by depth (φ) (resp. $\forall$-depth (φ)), is the maximum number of quantifiers (resp. universal quantifiers) in the scopes of which a subformula of φ can be found. More formally, for a quantifier-free formula ψ, $\text{depth}(\psi) = 0$; for any formulas ψ, ψ_0, ψ_1, $\text{depth}(\exists v \psi) = \text{depth}(\forall v \psi) = 1 + \text{depth}(\psi)$, and $\text{depth}(\psi_0 \vee \psi_1) = \text{depth}(\psi_0 \wedge \psi_1) = \max[\text{depth}(\psi_0), \text{depth}(\psi_1)]$. $\forall$-depth$(-)$ is defined as depth$(-)$, except that $\forall\text{-depth}(\exists v \psi) = \forall\text{-depth}(\psi)$.

1.2. Preliminaries in computational complexity. For all real numbers r, $\lceil r \rceil$ (resp. $\lfloor r \rfloor$) is the least (resp. greatest) integer $n \geqq r$ (resp. $n \leqq r$) and $\log r$ is the logarithm of r in base 2.

Let $f(n)$ and $g(n)$ be two nonnegative real-valued functions on positive integers. We use the notation $f(n) = O(g(n))$ to mean that there is a constant number c such that, for all sufficiently large integers n, $f(n) \leqq cg(n)$.

We suppose that the reader knows the main definitions about Turing machines and nondeterministic computations (see [8, Chap. 7]). Our model of computation will be a one-dimensional nondeterministic Turing machine (NTM); more precisely, an NTM has several one-dimensional tapes, infinite to the right only, which consist of one (read-only) input tape and several (read-write) worktapes. A *multidimensional* NTM is an NTM whose worktapes are multidimensional. (Unless otherwise specified, NTM will mean a one-dimensional NTM.)

Let Σ be a finite set. Σ^+ will denote the set of finite nonempty words over the alphabet Σ. A subset of Σ^+ is called a *language over* Σ.

Let M be an NTM and Σ be the set of input symbols of M. (An input of M is a word of Σ^+.) M *accepts* an input w if, when it is started in start state with all tape cells blank except that the leftmost input tape cells contain w, and with all tape heads at the leftmost cells of the tapes, a computation (i.e., some sequence of moves) takes M to the accepting state. Further, if T is the number of moves of such a computation, then we say that *M accepts w in time T.*

Let $T(n)$ be a function from positive integers to positive integers. An NTM M *accepts* a language $L \subseteq \Sigma^+$ *in time* $T(n)$, if

(i) M accepts no input except the words of L, and

(ii) each word of L of length n is accepted by M in time at most $T(n)$.

To make sense, we require that $T(n) \geqq n+1$ since it needs $n+1$ moves to read an input of length n and the first blank symbol. Let NTIME $(f(n))$ denote the class of languages accepted by a NTM in time $\max(n+1, \lceil f(n) \rceil)$, for a real-valued function f.

Let A be a set of positive integers. Identifying each positive integer n with 1^n, (1^n is the word of n 1's), we can regard A as a language over the one-letter alphabet $\Sigma = \{1\}$. Thus A belongs to NTIME $(T(n))$ if the corresponding language belongs to this class. Let Bin (A) denote the set of binary representations of the integers of A. (Of course, Bin $(A) \subseteq \{0, 1\}^+$.)

Our complexity results for spectra will be presented with integers in unary notation. However, if one prefers binary notation, then the following lemma will give an immediate translation of our results.

LEMMA 1.1. *Let A be a set of positive integers. Then for all integers $d \geqq 1$ and $k \geqq 0$, the following statements are equivalent*:

(i) $A \in$ NTIME $(n^d (\log n)^k)$.

(ii) *There is an* NTM *which accepts* Bin (A) *in time* $O(n^d (\log n)^k)$, *where n is the input integer.*

(iii) Bin $(A) \in$ NTIME $(2^{dn} n^k)$, *where n is the length of the input integer.*

Proof. It is sufficient to remark that there is a deterministic Turing machine which transforms any integer n from unary to binary notation (resp. from binary to unary notation) in time $O(n)$: the lemma follows by linear speed-up (see respectively [7], [8] and [2], [17] for linear speed-up of nonlinear and linear functions). □

It is natural to *encode* a directed graph, $\mathcal{G}$, of domain D_m, with the word $w_1 w_2 \cdots w_{m^2}$ over the alphabet $\{0, 1\}$, defined by: for $1 \leqq i \leqq m^2$,

$$w_i = 1 \quad \text{iff} \quad \mathcal{G} \models R(a_1, a_2),$$

where (a_1, a_2) is the ith element of D_m^2 for lexicographical ordering. An NTM, M, *accepts* a set of graphs, G, *in time* $T(m)$ if M accepts in time $T(m)$ the set of words $w_1 \cdots w_{m^2}$ which encode the graphs of G. (Notice that the input has length m^2, not m; m is the number of vertices of the encoded graph.) We will say that a set of graphs, G, belongs to NTIME $(f(m))$ if an NTM accepts G in time $\max(m^2+1, \lceil f(m) \rceil)$.

2. Upper bounds for spectra. Lemmas, Propositions and Theorems 2.1 to 2.5 will be expressed for spectra. However, they hold as well for generalized spectra with the same proofs.

LEMMA 2.1. *For each sentence φ with at most d universal quantifiers, there is a quantifier-free formula $\varphi'(\bar{x}_d)$ such that*

$$\mathrm{Sp}(\varphi) = \mathrm{Sp}(\forall \bar{x}_d \varphi'(\bar{x}_d)).$$

Proof. By standard manipulation of quantifiers, it is easy to put all the quantifiers of φ in front of the sentence. Therefore we can assume that φ is prenex. We use the following fact: a sentence of the form $\forall \bar{v}_k \exists v' \psi(\bar{v}_k, v')$ is equivalent to the second-order sentence $\exists F \forall \bar{v}_k \psi(\bar{v}_k, F(\bar{v}_k))$ where F is a new k-ary function symbol. For example, the sentence $\forall x \exists y \forall z \exists t\, \psi(x, y, z, t)$ is equivalent to $\exists F_1 \forall x \forall z \exists t\, \psi(x, F_1(x), z, t)$ and then to

$$\exists F_1 \exists F_2 \forall x \forall z \psi(x, F_1(x), z, F_2(x, z)).$$

From this example, we clearly see that the prenex sentence φ can be transformed according to the following rule: To each existentially quantified variable v, associate a function symbol F (a "Skolem function") and replace each occurrence of v in the matrix of φ by the term $F(v_1, \cdots, v_k)$, where $v_1, \cdots, v_k$ are the universally quantified variables lying before v in the prefix of φ; lastly, remove all the existential quantifiers. □

It is natural to require that complexity classes be closed under intersection and union. We have:

PROPOSITION 2.2. *Let A, B be two sets in* $\mathrm{Sp}(d\forall)$, *for $d \geq 1$, and A' be a finite modification of A (i.e., A' is constructed from A by adding or removing finitely many positive integers). Then*

(i) $A \cap B \in \mathrm{Sp}(d\forall)$;

(ii) $A \cup B \in \mathrm{Sp}(d\forall)$;

(iii) $A' \in \mathrm{Sp}(d\forall)$.

Proof. (i) By Lemma 2.1, we can assume that $A = \mathrm{Sp}(\forall \bar{x}_d \psi_0(\bar{x}_d))$ and $B = \mathrm{Sp}(\forall \bar{x}_d \psi_1(\bar{x}_d))$, where ψ_0 and ψ_1 are quantifier-free formulas. We can also assume that ψ_0 and ψ_1 have no common relation or function symbol. Then $A \cap B = \mathrm{Sp}(\forall \bar{x}_d (\psi_0(\bar{x}_d) \wedge \psi_1(\bar{x}_d)))$. This proves (i).

(ii) Clearly, $A \cup B = \mathrm{Sp}(\varphi)$ with $\varphi = \forall \bar{x}_d \psi_0(\bar{x}_d) \vee \forall \bar{x}_d \psi_1(\bar{x}_d)$. Let R_0 be a new 0-ary relation symbol. (Intuitively, the proposition symbol R_0 stands for the disjunct $\forall \bar{x}_d \psi_0(\bar{x}_d)$.) φ is equivalent to the second-order sentence $\exists R_0[(\neg R_0 \vee \forall \bar{x}_d \psi_0(\bar{x}_d)) \wedge (R_0 \vee \forall \bar{x}_d \psi_1(\bar{x}_d))]$. The first-order sentence in brackets is equivalent to the sentence

$$\varphi' = \forall \bar{x}_d[(\neg R_0 \vee \psi_0(\bar{x}_d)) \wedge (R_0 \vee \psi_1(\bar{x}_d))],$$

and then $A \cup B = \mathrm{Sp}(\varphi')$.

(iii) By (i) and (ii), it is sufficient to prove that for each positive integer k, the sets $\{n : n > k\}$ and $\{k\}$ belong to $\mathrm{Sp}(1\forall)$. We have

$$\{n: n > k\} = \mathrm{Sp}\left(\exists x_0 \cdots \exists x_k \bigwedge_{0 \leq i < j \leq k} x_i \neq x_j\right),$$

$$\{k\} = \mathrm{Sp}\left(\exists x_1 \cdots \exists x_k \left(\bigwedge_{1 \leq i < j \leq k} x_i \neq x_j \wedge \forall y \bigvee_{i=1}^{k} y = x_i\right)\right). \qquad \square$$

Another argument for the naturalness of a complexity measure is the fact that it has several reformulations.

THEOREM 2.3. *Let A be a set of positive integers. $A \in \mathrm{Sp}\,(d\forall)$ if and only if there is a sentence φ such that $A = \mathrm{Sp}\,(\varphi)$ with the following property* (i) (*resp.* (ii), (iii)):

(i) $\forall$-depth $(\varphi) = d$;

(ii) *φ has d quantifiers*;

(iii) depth $(\varphi) = d$.

Theorem 2.3 is an immediate consequence of

PROPOSITION 2.4. *For each sentence φ of universal depth d, there is a sentence φ' with d universal quantifiers only and no existential quantifier, so that* $\mathrm{Sp}\,(\varphi) = \mathrm{Sp}\,(\varphi')$.

Proposition 2.4 is a particular case of Lemma 2.4′ proved in the Appendix. The proof uses additional "Skolem functions" and relations, as do the proofs of Lemma 2.1 and Proposition 2.2(ii), respectively. Notice that Proposition 2.2 (except part (iii)) is an immediate consequence of Theorem 2.3.

To prove our upper bound theorem for spectra, we shall use the following definitions and lemma.

DEFINITIONS. An *elementary formula of type $\mathcal{T}$* is a signed atomic formula of one of the five following forms:

$$(\neg)v_1 = v_2, \quad (\neg)R(\bar{v}_r), \quad F(\bar{v}_r) = v_{r+1},$$

where R, F are respectively r-ary relation and function symbols of $\mathcal{T}$.

Let n be a strictly positive integer. An *n-formula of type $\mathcal{T}$* is a formula of type $\mathcal{T}_n = \mathcal{T} \cup D_n$ which is constructed from an elementary formula of type $\mathcal{T}$ by replacing each variable by an element of D_n. (So an n-formula of type $\mathcal{T}$ is of the form $(\neg)e_1 = e_2$, $(\neg)R(\bar{e}_r)$ or $F(\bar{e}_r) = e_{r+1}$, where $R, F \in \mathcal{T}$ and $e_i \in D_n$.)

Remark. Since n has length $O(\log n)$ *in base* 2, *an* n-formula can be encoded with $O(\log n)$ symbols.

LEMMA 2.5. *If $A \in \mathrm{Sp}\,(d\forall)$, then there are a type $\mathcal{T}$ and a sentence*

$$\varphi = \forall \bar{x}_d \exists \bar{y}_k \bigvee_{i=1}^{c} \psi_i(\bar{x}_d, \bar{y}_k)$$

such that:

(i) $A = \mathrm{Sp}\,(\varphi)$;

(ii) *each ψ_i, $1 \leqq i \leqq c$, is a conjunction of elementary formulas of type $\mathcal{T}$.*

Proof. By Lemma 2.1, $A = \mathrm{Sp}\,(\forall \bar{x}_d \varphi'(\bar{x}_d))$ for a quantifier-free formula φ'. We construct φ as follows. First put φ' in disjunctive normal form. Secondly, transform the signed atomic subformulas of φ' as in the following example: the subformula $\neg R(F_1(x, y), F_2(y))$ is replaced by the equivalent formula

$$\exists z \exists t (F_1(x, y) = z \wedge F_2(y) = t \wedge \neg R(z, t)).$$

Lastly, put the added existential quantifiers in the prefix. □

THEOREM 2.6. *Let A be a set of positive integers and G be an isomorphism invariant set of directed graphs. Then*:

(i) $A \in \mathrm{Sp}\,(d\forall)$ *implies* $A \in \mathrm{NTIME}\,(n^d(\log n)^2)$, *for each integer* $d \geqq 1$;

(ii) $G \in \mathrm{GenSp}\,(d\forall)$ *implies* $G \in \mathrm{NTIME}\,(m^d(\log m)^2)$, *for each integer* $d \geqq 2$.

Proof. of (i) Let $A = \mathrm{Sp}\,(\varphi)$ where the sentence φ of type $\mathcal{T}$ is of the form

$$\forall \bar{x}_d \exists \bar{y}_k \bigvee_{i=1}^{c} \psi_i(\bar{x}_d, \bar{y}_k)$$

of Lemma 2.5. The principle of the algorithm which checks if an integer belongs to A, is given by the following equivalences.

Let n be a positive integer and $\mathcal{M}$ be a structure of type $\mathcal{T}$ on the domain D_n. Then:
$\mathcal{M} \models \varphi \leftrightarrow$ for all $\bar{a}_d \in D_n^d$, there are $\bar{b}_k \in D_n^k$ and $i \in \{1, \cdots, c\}$, such that

$$\mathcal{M} \models \psi_i(\bar{a}_d, \bar{b}_k)$$

$\leftrightarrow$ there are a function $g: D_n^d \to D_n^k$ and a function $h: D_n^d \to \{1, \cdots, c\}$, such that

$$\mathcal{M} \models \bigwedge_{\bar{a}_d \in D_n^d} \psi_{h(\bar{a}_d)}(\bar{a}_d, g(\bar{a}_d)).$$

Let $\varphi_{g,h}$ denote the above conjunction. Clearly, $\varphi_{g,h}$ is of the form $\bigwedge_{i=1}^{n'} \pi_i$, where each π_i is an n-formula of type $\mathcal{T}$ and $n' = O(n^d)$. (Here and in the following, the constant numbers implicit in O-notation only depend on the sentence φ.)

We have:

$$n \in \mathrm{Sp}(\varphi) \leftrightarrow \varphi \text{ has a model of domain } D_n,$$

and then from the previous equivalences:

(*) $n \in \mathrm{Sp}(\varphi) \leftrightarrow$ there are functions $g: D_n^d \to D_n^k$ and $h: D_n^d \to \{1, \cdots, c\}$ such that the conjunction $\varphi_{g,h}$ is satisfiable in D_n.

For each (relation or function) symbol $s \in \mathcal{T}$, let Π_s denote the set of n-formulas $\pi_i (1 \leqq i \leqq n')$ mentioned above which contain the symbol s. The following equivalences are obvious:

$\bigwedge_{s \in \mathcal{T}} \bigwedge_{\pi \in \Pi_s} \pi$ is satisfiable $\leftrightarrow$ for each $s \in \mathcal{T}$, $\bigwedge_{\pi \in \Pi_s} \pi$ is satisfiable

$\leftrightarrow$ for each $s \in \mathcal{T}$, Π_s includes no pair of incompatible n-formulas.

(Incompatible n-formulas are of the forms $R(\bar{e}_r)$ and $\neg R(\bar{e}_r)$ or $F(\bar{e}_r) = e$ and $F(\bar{e}_r) = e'$ with $e \neq e'$.)

The following nondeterministic algorithm (divided in two procedures, (a) and (b)), thus emerges.

(a) Construct the binary representation of n. Guess (nondeterministically) the functions g and h and write the conjunction $\varphi_{g,h}$. There are n^d values to guess for each function. Since integers are written in base 2, procedure (a) requires a time $O(n^d \log n)$.

(b) Check (deterministically) if $\varphi_{g,h}$ is satisfiable in D_n. (Recall: $\varphi_{g,h} = \bigwedge_{i=1}^{n'} \pi_i$, where $n' = O(n^d)$ and each π_i is an n-formula of type $\mathcal{T}$.) Procedure (b) divides into three steps:

(b_1) Evaluate the (in)equalities, i.e., n-formulas of the form $(\neg)e = e'$, among the n-formulas π_i; if any is false, then $\varphi_{g,h}$ is not satisfiable; otherwise, delete the (true) (in)equalities and sort the conjunction $\varphi_{g,h}$ in the form $\bigwedge_{s \in \mathcal{T}} \bigwedge_{\pi \in \Pi_s} \pi$.

(b_2) Sort each conjunction $\bigwedge_{\pi \in \Pi_s} \pi$ according to the lexicographical order of the arguments $\bar{e}_r$ of the n-formulas π. (Recall that π has form $(\neg)R(\bar{e}_r)$ or $F(\bar{e}_r) = e_{r+1}$.)

(b_3) For each $s \in \mathcal{T}$, test whether Π_s includes a pair of incompatible n-formulas, using the fact that incompatible n-formulas (if any) now appear side by side.

We clearly see that steps (b_1) and (b_3) each require time $O(n^d \log n)$. Therefore procedure (b) requires time $O(n^d (\log n)^2)$ which is the time to execute step (b_2) with the sorting algorithm of [1, p. 78].

It is easy to implement procedures (a) and (b) on an NTM, and then $A \in \mathrm{NTIME}(n^d (\log n)^2)$, by linear speed-up.

Proof of (ii) Similar to the proof of (i). Therefore we only emphasize the differences. Let $G = \mathrm{GenSp}(\varphi)$ where the sentence φ of type $\mathcal{T} = \{\mathbf{R}, V_1, \cdots, V_p\}$ is as in Lemma 2.5.

Let $\mathcal{G}$ be a directed graph of domain D_m. By the argument used in the proof of (i), we obtain:

$\mathcal{G} \in \mathrm{GenSp}(\varphi) \leftrightarrow$ there are functions $g: D_m^d \to D_m^k$ and $h: D_m^d \to \{1, \cdots, c\}$ such that $\mathcal{G}$ has an expansion $\langle \mathcal{G}, V_1, \cdots, V_p \rangle$ which is a model of $\varphi_{g,h}$.

Let $\Delta(\mathcal{G})$ denote the conjunction of the m^2 m-formulas $(\neg)\mathbf{R}(e_1, e_2)$, where $(e_1, e_2) \in D_m^2$, such that $\mathcal{G} \models (\neg)\mathbf{R}(e_1, e_2)$. Then we obtain the following equivalence:

(**) $\mathcal{G} \in \mathrm{GenSp}(\varphi) \leftrightarrow$ there are functions $g: D_m^d \to D_m^k$ and $h: D_m^d \to \{1, \cdots, c\}$, such that the conjunction $\varphi_{g,h} \wedge \Delta(\mathcal{G})$ is satisfiable in D_m.

The second member of equivalence (**) is similar to the second member of equivalence (*) in the proof of (i), except that $\varphi_{g,h}$ is now replaced by $\varphi_{g,h} \wedge \Delta(\mathcal{G})$ which is a conjunction of $O(m^d)$ m-formulas of type $\mathcal{T}$, since $d \geqq 2$. Therefore the remainder of the proof is exactly like that of (i). □

Remarks. Part (i) of Theorem 2.6 is essentially stated by H. Lewis [12, Prop. 3.2] with a proof more informal than ours. More precisely, he states the following: "Whether a prenex sentence with d universal quantifiers has a model of cardinality n can be ascertained nondeterministically in time $f(|\varphi| \cdot n^d)$, for some polynomial f." ($|\varphi|$ denotes the length of sentence φ.) However, H. Lewis states his proposition in a different context: he uses it as a tool to prove a complexity upper bound for the satisfiability problem of a class of sentences with a fixed number of universal quantifiers; he does not need to know a precise value of polynomial f. (Moreover he assumes that φ contain no function symbol and no equality symbol.)

By Theorems 2.3 and 2.6, the spectrum of any sentence φ of universal depth d belongs to $\mathrm{NTIME}(n^d(\log n)^2)$. In fact, there is a natural generalization of the algorithm of Theorem 2.6 which accepts $\mathrm{Sp}(\varphi)$ in time $O(n^d(\log n)^2)$, and similarly for generalized spectra.

3. Lower bounds for spectra. We want to "simulate" a computation of an NTM in a finite structure. Therefore we need a numbering of the structure to express the numbering of the tape cells and of the time units the computation requires. We can construct a linear order by:

LEMMA 3.1. *There is a first-order sentence ψ such that:*

(i) *ψ has only two universal quantifiers;*

(ii) *ψ is of type $\mathcal{T} = \{F_{\mathrm{Suc}}, R_<, R_{\mathrm{Suc}}, c_0, c_l\}$, where F_{Suc} is a unary function symbol, $R_<$ and R_{Suc} are binary relation symbols and c_0, c_l are constant symbols;*

(iii) *if $\mathcal{M} = \langle D, R_<, R_{\mathrm{Suc}}, c_0, c_l \rangle$ is a finite structure of cardinality at least two, then:*

$\mathcal{M} \models \exists F_{\mathrm{Suc}} \psi \leftrightarrow R_<$ *is a strict linear order of D, and R_{Suc}, c_0, c_l are respectively the corresponding successor relation and the first and last elements of this order.*

Proof. Let us give some sentences with their intuitive meaning.

ψ_1: $\forall x \exists y F_{\mathrm{Suc}}(y) = x \wedge \forall x F_{\mathrm{Suc}}(x) \neq x$.

ψ_1 expresses that F_{Suc} is a permutation of the (finite) domain and has no loop.

ψ_2: $\forall x \neg R_<(x, x) \wedge \forall x \forall y [x \neq y \to (R_<(x, y) \leftrightarrow \neg R_<(x, y))]$.

ψ_2 expresses that $R_<$ is a tournament.

$$\psi_3: \forall x \forall y[(R_<(x, y) \wedge y \neq c_l) \rightarrow R_<(x, F_{\text{Suc}}(y))] \wedge F_{\text{Suc}}(c_l) = c_0.$$

ψ_3 expresses that $R_<$ is a transitive relation for function F_{Suc}, except for the value $F_{\text{Suc}}(c_l)$ which is c_0.

A consequence of the conjunction $\psi_1 \wedge \psi_2 \wedge \psi_3$ is that the permutation F_{Suc} has only one cycle. In investigating this cycle, we clearly see that this conjunction implies that $R_<$ is a linear order of the domain, with first and last elements c_0, c_l, respectively, and that F_{Suc} maps each element to its immediate successor for this order, with moreover $F_{\text{Suc}}(c_l) = c_0$. Therefore the following sentence defines the successor relation:

$$\psi_4: \forall x \forall y[R_{\text{Suc}}(x, y) \leftrightarrow (F_{\text{Suc}}(x) = y \wedge x \neq c_l)].$$

So the conjunction $\psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4$ has properties (ii) and (iii). Clearly, it has an equivalent form with property (i) also since it is a conjunction of sentences with at most two universal quantifiers. □

In the following, we will use Lemma 3.1 with the expressive symbols $<$, Suc, 0, l instead of $R_<$, R_{Suc}, c_0, c_l, respectively.

The following is our second main result.

THEOREM 3.2. *Let A be a set of positive integers and G be an isomorphism invariant set of directed graphs. Then, for any integer $d \geqq 2$:*

(i) *$A \in \text{NTIME}(n^d)$ implies $A \in \text{Sp}(d\forall)$;*

(ii) *$G \in \text{NTIME}(m^d)$ implies $G \in \text{GenSp}(d\forall)$.*

Moreover these two implications hold even if we use multidimensional NTMs.

Proof of (i). Let A be a set in NTIME (n^d). By Proposition 2.2(iii), we can assume that A is a set of integers $n \geqq 2$. For each integer $n \geqq 2$, let us consider ¢1^{n-2}\$, a word of length n, that we regard as a "self-bounded" unary representation of n. Clearly, the set $A' = \{¢1^{n-2}\$: n \in A\}$ also belongs to NTIME (n^d), and then, by speed-up, there is an NTM, M, which accepts A' in time $n^d - 1$. The input head of M does not visit any cell outside the input, because of the "bounds" ¢ and \$. (An NTM which accepts A must visit the n cells of the input plus the next one to the right. It is less convenient to encode $n+1$ cells than n cells in a domain of n elements: this is why we consider A' in place of A. Similarly, we choose the time bound $n^d - 1$ for technical reasons which will be explained later.)

In the following, we will adopt almost the same notation as J. Lynch [13] used in the proof of his theorem (mentioned in our introduction), so that it is easy to compare his proof and ours.

Suppose that the tapes of M are the input tape and only one (one-dimensional) worktape. The set of input symbols of M is of course $\Sigma = \{1, ¢, \$\}$. Let Γ be the set of worktape symbols of M, including the blank symbol $\mathbf{b}$. Let Q be the set of M's states, including q_0, the start state, and q_a, the accepting state. Let δ be the transition function of the NTM M. More precisely, δ maps each $(\sigma, \gamma, q) \in \Sigma \times \Gamma \times Q$ to a subset of $\Gamma \times \{-1, 0, 1\}^2 \times Q$; the set $\delta(\sigma, \gamma, q)$ consists of those $(\gamma', \alpha, \beta, q')$ such that if at some time, the symbols under the input head and the worktape head are respectively σ, γ and M is in state q, then the following is a possible move: M prints γ' in place of γ, and at the following time, the input head and the worktape head move accordingly to numbers α, β, respectively (0 means no movement, 1 and -1 a movement to the right and to the left, respectively) and M enters state q'.

Since M accepts A' in time $n^d - 1$, we can regard each computation of M (on an input of length n) as a sequence of exactly $n^d - 1$ moves by adopting the following conventions: each computation of M is truncated by an $n^d - 1$ time-bounded clock;

in case M enters an (accepting or rejecting) final state before the clock rings, then M continues "running" in the same configuration until the clock rings. Therefore each computation of M uses exactly n^d configurations (including the start and the final configurations) and at most n^d worktape cells.

By Lemma 3.1, there is an existential second-order sentence which defines a linear order $<$ of the domain with the corresponding successor relation, Suc, and first and last elements, 0 and l, respectively. To encode the moves of M, we shall use the lexicographical order of k-tuples, constructed from $<$, and also denoted $<$, and the corresponding successor relation, also denoted Suc. More precisely, $\bar{x}_k < \bar{y}_k$ is an abbreviation of

$$x_1 < y_1 \vee \bigvee_{j=2}^{k} \left(\bigwedge_{i=1}^{j-1} x_i = y_i \wedge x_j < y_j \right).$$

Similarly, Suc $(\bar{x}_k, \bar{y}_k)$ abbreviates the following formula:

$$\left[\mathrm{Suc}\,(x_1, y_1) \wedge \bigwedge_{i=2}^{k} (x_i = l \wedge y_i = 0) \right]$$

$$\vee \bigvee_{j=2}^{k-1} \left[\bigwedge_{i=1}^{j-1} x_i = y_i \wedge \mathrm{Suc}\,(x_j, y_j) \wedge \bigwedge_{i=j+1}^{k} (x_i = l \wedge y_i = 0) \right]$$

$$\vee \left[\bigwedge_{i=1}^{k-1} x_i = y_i \wedge \mathrm{Suc}\,(x_k, y_k) \right].$$

Clearly $\bar{x}_k < \bar{y}_k$ and $\mathrm{Suc}\,(\bar{x}_k, \bar{y}_k)$ express the required relations. Let $\bar{y}_k = \bar{x}_k + 1$ and $\bar{x}_k = \bar{y}_k - 1$ be synonymous with $\mathrm{Suc}\,(\bar{x}_k, \bar{y}_k)$. Lastly, let $\bar{x}_k = \bar{y}_k$ (or $\bar{x}_k = \bar{y}_k + 0$) and $\bar{x}_k \neq \bar{y}_k$ abbreviate the conjunction $\bigwedge_{i=1}^{k} x_i = y_i$ and its negation, respectively.

Let $\bar{v}$ in short denote the d-tuple of variables $\bar{v}_d = v_1, \cdots, v_d$. Similarly, let $\bar{0}$ (resp. $\bar{l}$) denote the constant symbol 0 (resp. l) repeated d times. For convenience, our notation does not distinguish between a variable and its assignment.

Now let us consider a domain of n elements, denoted E_n, and let us intuitively describe how to encode in E_n a computation of M on an input of length n. There are as many elements in E_n^d as time units (resp. worktape cells) used in the computation. So each element $\bar{t}$ (resp. $y, \bar{y}$) of E_n^d (resp. E_n, E_n^d) corresponds to a time unit (resp. an input cell, a worktape cell), also denoted $\bar{t}$ (resp. $y, \bar{y}$). (This is the "folding" technique of [5], [11], [13].) For convenience, we will use (d, k)-ary functions: a (d, k)-ary function symbol F abbreviates a k-tuple $F_1, \cdots, F_k$ of d-ary function symbols. Let us introduce the following relation and function symbols for which the argument $\bar{t}$ intuitively means "at time $\bar{t}$":

The d-ary relation symbols $C_\sigma^*, C_\gamma, C'_\gamma, \sigma \in \Sigma, \gamma \in \Gamma$: $C_\sigma^*(\bar{t})$ (intuitively) means "the symbol under the input head is σ"; $C_\gamma(\bar{t})$ (resp. $C'_\gamma(\bar{t})$) means "the symbol under (resp. printed by) the worktape head is γ."

The d-ary function symbol H^*: $H^*(\bar{t})$ is "the cell, y, under the input head."

The (d, d)*-ary function symbol* H: $H(\bar{t}) = (H_1(\bar{t}), \cdots, H_d(\bar{t}))$ is "the cell, $\bar{y}$, under the worktape head."

The d-ary relation symbols $S_q, q \in Q$: $S_q(\bar{t})$ means "M is in state q."

For technical reasons, we also introduce a $(d, 2d)$-ary function symbol, N, less intuitive than the previous symbols. It will be used to lexicographically number the n^d couples $(H(\bar{t}), \bar{t})$ of the computation. The (intuitive) value of $N(\bar{x}) = (N_1(\bar{x}), \cdots, N_{2d}(\bar{x}))$ is "the couple $(H(\bar{t}), \bar{t})$ of number $\bar{x}$."

Function N is defined by the two following sentences with only d universal quantifiers:

φ_1: $\forall \bar{t} \exists \bar{x} (H(\bar{t}), \bar{t}) = N(\bar{x})$,

where the subformula $(H(\bar{t}), \bar{t}) = N(\bar{x})$ abbreviates

$$\bigwedge_{i=1}^{d} H_i(\bar{t}) = N_i(\bar{x}) \wedge \bigwedge_{i=1}^{d} t_i = N_{d+i}(\bar{x});$$

φ_2: $(\forall \bar{x} \neq \bar{l})(\exists \bar{x}' = \bar{x} + 1) N(\bar{x}) < N(\bar{x}')$,

where $(\forall \bar{x} \neq \bar{l})$ and $(\exists \bar{x}' = \bar{x} + 1)$ abbreviate $\forall \bar{x} (\bar{x} \neq \bar{l} \rightarrow \cdots)$ and $\exists \bar{x}' (\bar{x}' = \bar{x} + 1 \wedge \cdots)$, respectively. Clearly, φ_1 expresses that the equality $(H(\bar{t}), \bar{t}) = N(\bar{x})$ defines a bijection between the n^d d-tuples $\bar{t}$ and the n^d d-tuples $\bar{x}$, and φ_2 expresses that N is an increasing function for lexicographical order.

Figure 1 illustrates (for a particular computation) how function N numbers the ordered pairs $(H(\bar{t}), \bar{t})$ which are intuitively the worktape head positions during the computation. (We assume that $d = 1$ and $n = 6$; each element of E_6 is denoted in Fig. 1 by its rank for order $<$.)

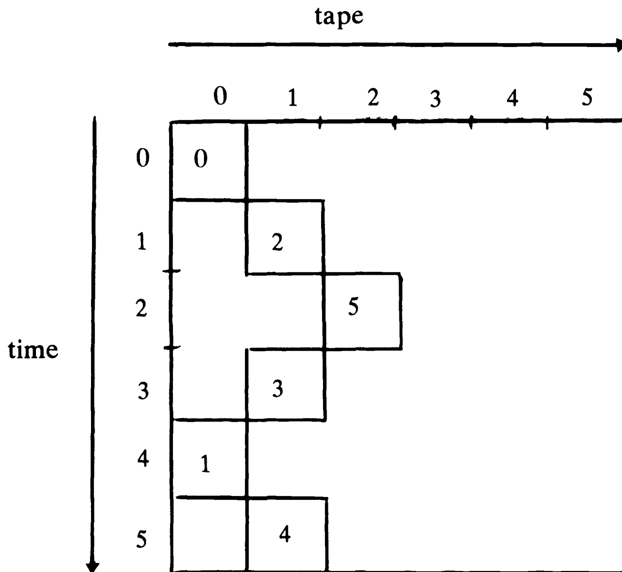


FIG. 1

It should be clear that $\varphi_1 \wedge \varphi_2$ implies that function N satisfies the two following (informal) statements:

(a) $\bar{t}'$ is the first time after $\bar{t}$ such that $H(\bar{t}) = H(\bar{t}')$ iff $H(\bar{t}) = H(\bar{t}')$ and there is $\bar{x}$ such that $N(\bar{x}) = (H(\bar{t}), \bar{t})$ and $N(\bar{x}+1) = (H(\bar{t}'), \bar{t}')$.

(b) $\bar{t}' \neq \bar{0}$ and the worktape cell $H(\bar{t}')$ has never been visited before time $\bar{t}'$ iff there are $\bar{x}, \bar{t}$ such that $N(\bar{x}) = (H(\bar{t}), \bar{t})$ and $N(\bar{x}+1) = (H(\bar{t}'), \bar{t})$ and $H(\bar{t}) \neq H(\bar{t}')$.

The following sentences express how to encode an accepting computation of M in a structure with the relation and function symbols mentioned above.

$$\varphi_3\colon \forall\bar{t}[(H^*(\bar{t})=0 \leftrightarrow C^*_{¢}(\bar{t})) \wedge (H^*(\bar{t})=l \leftrightarrow C^*_{\$}(\bar{t})) \\ \wedge (0<H^*(\bar{t})<l \leftrightarrow C^*_1(\bar{t}))].$$

φ_3 expresses what is the symbol under the input head. (Recall that the input is $¢1^{n-2}\$$.)

$$\varphi_4\colon\ \underline{C}_b(\bar{0}) \wedge H^*(\bar{0})=0 \wedge H(\bar{0})=\bar{0} \wedge \underline{S}_{q_0}(\bar{0}).$$

(In φ_4 and in the following sentences, $\underline{C}_\gamma(\bar{t})$, $\gamma\in\Gamma$, abbreviates the conjunction $C_\gamma(\bar{t}) \wedge \bigwedge_{\gamma'\in\Gamma-\{\gamma\}}\neg C_{\gamma'}(\bar{t})$; $\underline{C}'_\gamma(\bar{t})$, $\gamma\in\Gamma$, and $\underline{S}_q(\bar{t}), q\in Q$, are similar abbreviations.) φ_4 describes the start configuration.

Let φ_5 be the conjunction, for all $(\sigma,\gamma,q)\in\Sigma\times\Gamma\times Q$, of:

$$(\forall\bar{t}\neq\bar{l})(\exists\bar{t}'=\bar{t}+1) \\ [(C^*_\sigma(\bar{t}) \wedge C_\gamma(\bar{t}) \wedge S_q(\bar{t})) \\ \rightarrow \bigvee (\underline{C}'_{\gamma'}(\bar{t}) \wedge H^*(\bar{t}')=H^*(\bar{t})+\alpha \wedge H(\bar{t}')=H(\bar{t})+\beta \wedge \underline{S}_{q'}(\bar{t}'))],$$

where the disjunction extends over all $(\gamma',\alpha,\beta,q')$ of $\delta(\sigma,\gamma,q)$. φ_5 describes the set of possible moves determined by the transition function δ. (Notice that we assume that the worktape head prints the symbol γ' at time $\bar{t}$, not at time $\bar{t}+1$.)

$$\varphi_6\colon\ (\forall\bar{t}'\neq\bar{0})\exists\bar{t}\exists\bar{x}(\exists\bar{x}'=\bar{x}+1) \\ [N(\bar{x})=(H(\bar{t}),\bar{t}) \wedge N(\bar{x}')=(H(\bar{t}'),\bar{t}') \\ \wedge \bigwedge_{\gamma\in\Gamma} [(H(\bar{t})=H(\bar{t}') \wedge C'_\gamma(\bar{t})) \rightarrow \underline{C}_\gamma(\bar{t}')] \\ \wedge [H(\bar{t})\neq H(\bar{t}') \rightarrow \underline{C}_b(\bar{t}')]].$$

Using equivalences (a) and (b), φ_6 expresses what is the symbol under the worktape head at time $\bar{t}'\neq\bar{0}$, according to whether the scanned worktape cell has been visited or not before.

$$\varphi_7\colon S_{q_a}(\bar{l}).$$

Let φ_0 be the sentence ψ of Lemma 3.1 and φ be the conjunction $\bigwedge_{i=0}^{7}\varphi_i$. Then it should be clear that M accepts $¢1^{n-2}\$$ iff φ has a model of cardinality n. Hence $n\in A$ iff $n\in \mathrm{Sp}(\varphi)$. Moreover, φ is a conjunction of sentences with at most d universal quantifiers. This proves part (i) of the theorem in case the machine M has only one (one-dimensional) worktape.

In the general case, for every k-dimensional worktape of M, we need a (d, kd)-ary function symbol, H, and a $(d,(k+1)d)$-ary function symbol, N, and the corresponding sentences φ_1, φ_2 and φ_6. There are also obvious modifications of sentences φ_4 and φ_5.

Proof of (ii). Similar to the proof of (i). Therefore we only dwell on the differences. Let G be an isomorphism invariant set of directed graphs such that $G\in \mathrm{NTIME}(m^d)$. Let Σ be the alphabet of six symbols $\{0, 1, (0, ¢), (1, ¢), (0, \$), (1, \$)\}$. A "self-bounded" encoding of a directed graph $\mathscr{G}$ is a word of Σ^{m^2},

$$(w_1, ¢)w_2\cdots w_{m^2-1}(w_{m^2}, \$),$$

where the word $w_1\cdots w_{m^2}$ of $\{0,1\}^{m^2}$ encodes the graph $\mathscr{G}$. Clearly, there is an NTM which accepts the set of self-bounded encodings of the graphs of G in time $\max(m^2, m^d-1)$.

We construct a sentence φ such that $G = \mathrm{GenSp}(\varphi)$ exactly as in the proof of (i), except that the conjunct φ_3, expressing what is the symbol under the input head, is now replaced by the conjunction of:

$$\forall \bar{t}[H^*(\bar{t}) = (0, 0) \leftrightarrow C^*_{¢}(\bar{t})],$$

$$\forall \bar{t}[H^*(\bar{t}) = (l, l) \leftrightarrow C^*_{\$}(\bar{t})],$$

$$\forall \bar{t} \exists x \exists y [H^*(\bar{t}) = (x, y) \wedge (\mathbf{R}(x, y) \leftrightarrow C^*_1(\bar{t})) \wedge (\neg \mathbf{R}(x, y) \leftrightarrow C^*_0(\bar{t}))].$$

Note that H^* is now a $(d, 2)$-ary function symbol since we have to "fold up" an input of length m^2 to encode it in a domain of m elements.

There is a slight technical difficulty in case $G \in \mathrm{NTIME}(m^2)$, because an NTM which accepts the set of self-bounded encodings of graphs of G in time m^2 cannot be sped up, and then each computation requires m^2+1 configurations. In this case, the ordered pair $\bar{l} = (l, l)$ does not encode the last time unit, but rather the time unit before last. However, we are not interested by the last configuration, except for its state. From these remarks, the reader should be able to modify sentences φ_5 and φ_7 for this case. □

Remarks. In fact, we have proved a little more than the stated theorem: if A, a set of positive integers, belongs to $\mathrm{NTIME}(n^d)$, for an integer $d \geqq 2$, then A is the spectrum of a sentence with at most d universal quantifiers and a type of arity d. Similarly, for generalized spectra.

As P. Pudlák [16] points out, the implication $G \in \mathrm{NTIME}(m^d) \rightarrow G \in \mathrm{GenSp}(d\forall)$ might be useful for finding some nontrivial lower bound for a concrete problem in NP. In particular, if $G \notin \mathrm{GenSp}(2\forall)$ then $G \notin \mathrm{NTIME}(m^2)$.

4. Corollaries.

COROLLARY 4.1 (Pudlák [15] for (ii)). *Let d be a nonnegative integer. Then:*

(i) *there is a set of positive integers, A, such that:*

$$A \in \mathrm{Sp}(d+1\forall) - \mathrm{Sp}(d\forall);$$

(ii) *there is a set of directed graphs, G, such that*

$$G \in \mathrm{GenSp}(d+1\forall) - \mathrm{GenSp}(d\forall).$$

Proof. (i) Clearly, $\{1\} \in \mathrm{Sp}(1\forall) - \mathrm{Sp}(0\forall)$. So assume that $d \geqq 1$. A particular case of the nondeterministic time hierarchy [4], [17] is that there is a set of positive integers

$$A \in \mathrm{NTIME}(n^{d+1}) - \mathrm{NTIME}(n^d(\log n)^2).$$

From Theorems 2.6 and 3.2, it immediately follows that

$$A \in \mathrm{Sp}(d+1\forall) - \mathrm{Sp}(d\forall).$$

(ii) Let us consider a sentence φ with $d+1$ universal quantifiers only, such that $A = \mathrm{Sp}(\varphi)$. Let $\mathcal{T}$ be the type of φ. Now let us regard φ as a sentence of type $\mathcal{T} \cup \{\mathbf{R}\}$, $\mathbf{R}$ a new binary relation symbol. Let us define $G(A) = \mathrm{GenSp}(\varphi)$. $G(A)$ is the set of directed graphs on n vertices, such that $n \in A$. Let us assume that $G(A) = \mathrm{GenSp}(\varphi')$ where φ' has at most d universal quantifiers; then $A = \mathrm{Sp}(\varphi')$, a contradiction. □

Remarks. By the same proof, Corollary 4.1 holds not only for directed graphs, but also for structures of any type.

Corollary 4.1 can be strengthened as follows: The sentence φ (with $d+1$ universal quantifiers) such that $A = \mathrm{Sp}(\varphi)$ (resp. $G(A) = \mathrm{GenSp}(\varphi)$), has a type of arity $d+1$.

In analyzing the proof of Theorem 3.2(i), we clearly see that the only reason why we assume $d \geqq 2$ is that we need two universal quantifiers to define a linear order. If we suppose that a structure has a "built-in" linear ordering, then the restriction $d \geqq 2$ can be removed.

DEFINITION. Let φ be a sentence whose type includes $<$, a binary relation symbol. Let $\mathrm{Sp}_<(\varphi)$ denote the set of integers n, such that there is a model of φ of domain $D_n = \{0, \cdots, n-1\}$, where $<$ is interpreted as the natural order of D_n.

LEMMA 4.2. *There is a first-order sentence ψ such that:*

(i) *ψ has only one universal quantifier;*

(ii) *ψ is of type $\mathcal{T} = \{<, F_{\mathrm{Suc}}, c_0, c_l\}$ where $<$ is a binary relation symbol, F_{Suc} is a unary function symbol and c_0, c_l are constant symbols;*

(iii) *Let $\mathcal{M}$ be a structure of type $\mathcal{T}$ on the domain D_n where $<$ is the natural order of D_n. Then*

$$\mathcal{M} \models \psi \text{ iff } c_0 = 0, c_l = n-1 \text{ and } F_{\mathrm{Suc}}(e) = e+1 \text{ for each } e \in D_n,$$
$$\text{except that } F_{\mathrm{Suc}}(n-1) = 0.$$

Proof. Properties (ii) and (iii) clearly hold for the conjunction of

$$\forall x \, \exists y \, x = F_{\mathrm{Suc}}(y) \quad \text{and} \quad (\forall x \neq c_l) x < F_{\mathrm{Suc}}(x) \wedge F_{\mathrm{Suc}}(c_l) = c_0. \qquad \square$$

COROLLARY 4.3. *Let A be a set of positive integers in* NTIME (n). *Then there are a type $\mathcal{T}$ and a quantifier-free formula $\varphi(x)$ of type $\mathcal{T} \cup \{<\}$, such that:*

(i) *φ has only one variable x;*

(ii) $A = \mathrm{Sp}_<(\forall x \varphi(x))$;

(iii) *the arity of $\mathcal{T}$ is* 1.

Proof. Let φ' be the conjunction of the sentence ψ of Lemma 4.2 and of the sentences $\varphi_1, \cdots, \varphi_7$ (of Theorem 3.2) in which each occurrence of $\mathrm{Suc}(v, v')$ is replaced by $F_{\mathrm{Suc}}(v) = v' \wedge v \neq c_l$. (Moreover φ_5 and φ_7 are modified as in case $G \in$ NTIME (m^2) of Theorem 3.2(ii).) Let φ'' be the equivalent form of φ' with only one universal quantifier. $\forall x \varphi(x)$ is the sentence constructed from φ'' as in Lemma 2.1. $\square$

Let Prime denote the set of prime numbers. The usual algorithm for testing whether an integer n (written in base 2, for example) is prime, is to divide n by the $\lfloor \sqrt{n} \rfloor$ first positive integers. This algorithm works in time $\sqrt{n} f(\log n)$, for a polynomial f. (Pratt's nondeterministic algorithm [8, p. 342], [14] works in time $f(\log n)$, for a polynomial f.) From Lemma 1.1, it follows that Prime $\in$ NTIME (n). Indeed it seems that most "natural" sets of integers belong to NTIME (n).

COROLLARY 4.4. *There is a type $\mathcal{T}$ and a prenex sentence φ of type $\mathcal{T} \cup \{<\}$ such that:*

(i) Prime $= \mathrm{Sp}_<(\varphi)$;

(ii) *φ has only one variable (universally quantified);*

(iii) *the arity of $\mathcal{T}$ is* 1.

Appendix. We want to prove:

PROPOSITION 2.4. *For each sentence φ of universal depth d, there is a sentence φ' with d universal quantifiers only and no existential quantifier, so that* $\mathrm{Sp}(\varphi) = \mathrm{Sp}(\varphi')$.

LEMMA 2.4'. *For each formula $\varphi(\bar{x}_k)$ of type $\mathcal{T}$, such that* $\forall$-depth $(\varphi) = d$, *there is a quantifier-free formula $\varphi^*(\bar{x}_k, \bar{y}_d)$ of type $\mathcal{T}^* \supseteq \mathcal{T}$, so that* (i) *and* (ii) *are true:*

(i) *Any finite structure, $\mathcal{M}$, of type $\mathcal{T}$, has an expansion $\langle \mathcal{M}, \mathcal{N} \rangle$ of type $\mathcal{T}^*$, such that for each k-tuple $\bar{a}_k$ in $D(\mathcal{M})$,*

$$\mathcal{M} \models \varphi(\bar{a}_k) \quad \text{implies} \quad \langle \mathcal{M}, \mathcal{N} \rangle \models \forall \bar{y}_d \varphi^*(\bar{a}_k, \bar{y}_d).$$

(ii) *Given any finite structures, $\mathcal{M}$ and $\langle\mathcal{M},\mathcal{N}\rangle$, of respective types $\mathcal{T}$ and $\mathcal{T}^*$, and any k-tuple $\bar{a}_k$ in $D(\mathcal{M})$,*

$$\langle\mathcal{M},\mathcal{N}\rangle \models \forall \bar{y}_d \varphi^*(\bar{a}_k, \bar{y}_d) \quad \textit{implies} \quad \mathcal{M} \models \varphi(\bar{a}_k).$$

From Lemma 2.4′, we obtain Proposition 2.4 for each sentence φ, by taking $\varphi' = \forall \bar{y}_d \varphi^*(\bar{y}_d)$.

Proof of Lemma 2.4′. By induction on the "complexity" of the formula φ. We build φ^* as follows:

Case 1. For a quantifier-free formula ψ, $\psi^* = \psi$.

For all formulas $\psi_0(\bar{x}_k)$, $\psi_1(\bar{x}_k)$, $\psi(\bar{x}_k, z)$:

Case 2. $(\psi_0 \wedge \psi_1)^* = \psi_0^* \wedge \psi_1^*$;

Case 3. $(\forall z \psi)^* = \psi^*$;

Case 4. $(\psi_0 \vee \psi_1)^* = [R(\bar{x}_k) \vee \psi_0^*(\bar{x}_k, \bar{y}_d)] \wedge [\neg R(\bar{x}_k) \vee \psi_1^*(\bar{x}_k, \bar{y}_d)]$, where R is a new k-ary relation symbol and $d = \forall$-depth $(\psi_0 \vee \psi_1)$;

Case 5. $(\exists z \psi(\bar{x}_k, z))^* = \psi^*(\bar{x}_k, F(\bar{x}_k), \bar{y}_d)$, where F is a new k-ary function symbol and $d = \forall$-depth (ψ).

We shall prove (i) and (ii) only for the hardest case, Case 4, and shall give a sketch of proof of (i) in Case 5. The reader can easily complete the proof for the other cases.

Case 4. $\varphi = \psi_0 \vee \psi_1$. Let $\mathcal{M}$ be a finite structure of type $\mathcal{T}$ (the type of φ, and also of ψ_0 and ψ_1). Let $\langle\mathcal{M},\mathcal{N}_0\rangle$ and $\langle\mathcal{M},\mathcal{N}_1\rangle$ be the expansions of $\mathcal{M}$, of respective types $\mathcal{T}_0$ (the type of ψ_0^*) and $\mathcal{T}_1$ (the type of ψ_1^*), given by the induction hypothesis. We can suppose $\mathcal{T}_0 \cap \mathcal{T}_1 = \mathcal{T}$. Let R be the new k-ary relation on $D(\mathcal{M})$, defined as follows: for any k-tuple $\bar{a}_k$ in $D(\mathcal{M})$,

$$R(\bar{a}_k) \text{ is true iff } \langle\mathcal{M},\mathcal{N}_1\rangle \models \forall \bar{y}_d \psi_1^*(\bar{a}_k, \bar{y}_d).$$

Now suppose that $\mathcal{M} \models \varphi(\bar{a}_k)$ for a k-tuple $\bar{a}_k$ in $D(\mathcal{M})$. By the induction hypothesis, we have either $\langle\mathcal{M},\mathcal{N}_0\rangle \models \forall \bar{y}_d \psi_0^*(\bar{a}_k, \bar{y}_d)$ or $\langle\mathcal{M},\mathcal{N}_1\rangle \models \forall \bar{y}_d \psi_1^*(\bar{a}_k, \bar{y}_d)$, and then as a consequence,

$$\text{(A.1)} \quad \langle\mathcal{M},\mathcal{N}_0,\mathcal{N}_1,R\rangle \models \forall \bar{y}_d [[R(\bar{a}_k) \vee \psi_0^*(\bar{a}_k, \bar{y}_d)] \wedge [\neg R(\bar{a}_k) \vee \psi_1^*(\bar{a}_k, \bar{y}_d)]].$$

So (i) is proved.

Now we prove (ii). Let $\langle\mathcal{M},\mathcal{N}_0,\mathcal{N}_1,R\rangle$ be a finite structure of type $\mathcal{T}_0 \cup \mathcal{T}_1 \cup \{R\}$ and let $\bar{a}_k$ be a k-tuple in $D(\mathcal{M})$ for which (A.1) is true. Then according to whether $R(\bar{a}_k)$ is true or false, we have

$$\text{either} \quad \langle\mathcal{M},\mathcal{N}_1\rangle \models \forall \bar{y}_d \psi_1^*(\bar{a}_k, \bar{y}_d) \quad \text{or} \quad \langle\mathcal{M},\mathcal{N}_0\rangle \models \forall \bar{y}_d \psi_0^*(\bar{a}_k, \bar{y}_d),$$

and so $\mathcal{M} \models \varphi(\bar{a}_k)$.

Case 5. $\varphi(\bar{x}_k) = \exists z \psi(\bar{x}_k, z)$. Let $\mathcal{M}$ be a finite structure of type $\mathcal{T}$ (the type of φ and also of ψ) and let $\langle\mathcal{M},\mathcal{N}\rangle$ be the expansion of $\mathcal{M}$ of type $\mathcal{T}^*$ (the type of $\psi^*(\bar{x}_k, z, \bar{y}_d)$), given by the induction hypothesis. We construct a k-ary function F on $D(\mathcal{M})$ as follows: for each $\bar{a}_k$ in $D(\mathcal{M})$, $F(\bar{a}_k)$ is a chosen element b such that $\langle\mathcal{M},\mathcal{N}\rangle \models \forall \bar{y}_d \psi^*(\bar{a}_k, b, \bar{y}_d)$ if there exists one, and if not, then $F(\bar{a}_k)$ is any element in $D(\mathcal{M})$. It is clear that (i) is true with the expansion $\langle\mathcal{M},\mathcal{N},F\rangle$. □

Acknowledgments. Many thanks to Pascal Michel for helpful technical discussions. Thanks to Peter Clote for his help.

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] R. V. BOOK AND S. A. GREIBACH, *Quasi-realtime languages*, Math. Systems Theory, 4 (1970), pp. 97–111.

[3] C. C. CHANG AND H. J. KEISLER, *Model Theory*, North-Holland, Amsterdam, 1973.

[4] S. A. COOK, *A hierarchy for nondeterministic time complexity*, J. Comput. Systems Sci., 7 (1973), pp. 343–353.

[5] R. FAGIN, *Generalized first-order spectra and polynomial-time recognizable sets*, in Complexity of Computations, R. M. Karp, ed., American Mathematical Society, Providence, RI, 1974, pp. 43–73.

[6] ———, *A spectrum hierarchy*, Z. Math. Logik Grundlag. Math., 21 (1975), pp. 123–134.

[7] J. HARTMANIS AND R. E. STEARNS, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc., 117 (1965), pp. 285–306.

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

[9] N. IMMERMAN, *Number of quantifiers is better than number of tape cells*, J. Comput. Systems Sci., 22 (1981), pp. 384–405.

[10] ———, *Upper and lower bounds for first-order expressibility*, J. Comput. System Sci., 25, 1 (1982).

[11] N. D. JONES AND A. L. SELMAN, *Turing machines and the spectra of first-order formulas with equality*, J. Symbolic Logic, 39 (1974), pp. 139–150.

[12] H. R. LEWIS, *Complexity results for classes of quantificational formulas*, J. Comput. Systems Sci., 21 (1980), pp. 317–353.

[13] J. F. LYNCH, *Complexity classes and theories of finite models*, Math. Systems Theory, 15 (1982), pp. 127–144.

[14] V. R. PRATT, *Every prime has a succinct certificate*, this Journal, 4 (1975), pp. 214–220.

[15] P. PUDLÁK, *The observational predicate calculus and complexity of computations*, Comment. Math. Univ. Carolin., 16 (1975), pp. 395–398.

[16] ———, Personal communication (1982).

[17] J. I. SEIFERAS, M. J. FISHER AND A. R. MEYER, *Separating nondeterministic time complexity classes*, J. Assoc. Comput. Mach., 25 (1978), pp. 146–167.

SIAM J. COMPUT.
Vol. 13, No. 2, May 1984
© 1984 Society for Industrial and Applied Mathematics
011

RECURSIVE PROGRAMS AS DEFINITIONS IN FIRST ORDER LOGIC*

ROBERT CARTWRIGHT†

Abstract. Despite the reputed limitations of first order logic, it is easy to state and prove almost all interesting properties of recursive programs within a simple first order theory, by using an approach we call "first order programming logic". Unlike higher order logics based on fixed-point induction, first order programming logic is founded on deductive principles that are familiar to most programmers. Informal structural induction arguments (such as termination proofs for LISP append, McCarthy's 91-function, and Ackermann's function) have direct formalizations within the system.

The essential elements of first order programming logic are:

(1) The data domain D must be a finitely generated set that explicitly includes the "undefined" object $\perp$ (representing nontermination) as well as ordinary data objects.

(2) Recursive programs over D are treated as logical definitions augmenting a first order theory of the data domain.

(3) The interpretation of a recursive program is the least fixed-point of the functional corresponding to the program.

Since the data domain D is a finitely generated set, the first order axiomatization of D includes a structural induction axiom scheme. This axiom scheme serves as the fundamental "proof rule" of first order programming logic.

The major limitation of first order programming logic is that *every* fixed-point of the functional corresponding to a recursive program is an acceptable interpretation for the program. The logic fails to capture the notion of *least* fixed-point. To overcome this limitation, we present a simple, effective procedure for transforming an arbitrary recursive program into an equivalent recursive program that has a unique fixed-point, yet retains the logical structure of the original. Given this transformation technique, it is our experience that first order programming logic is sufficiently powerful to prove almost any property of practical interest about the functions computed by recursive programs.

Key words. programming logic, recursive programs, recursive definitions, rewrite rules, semantics, verification, program transformations

1. Introduction. It is a widely accepted part of computer science folklore that first order logic is too limited a formalism for stating and proving the interesting properties of recursive programs. Hitchcock and Park [16], for example, claim that the termination (totality) of a recursively defined function on a data domain **D** cannot be expressed by a sentence in a first order theory[1] of **D** augmented by the recursive definition. As a result of this criticism, most researchers developing programming logics for recursive programs have rejected first order logic in favor of more complex higher order systems, e.g., Milner [19], [20], [21], Park [23], deBakker [11], Gordon et al. [15], Scott and deBakker [25], Scott [26], deBakker and deRoever [12]. Nevertheless, we will show that a properly chosen, axiomatizable first order theory is a natural programming logic for recursive programs. In fact, we will present evidence which suggests that first order logic may be a more appropriate formalism for reasoning about specific recursive programs than higher order logics.

---

* Received by the editors September 1, 1978, and in final revised form July 15, 1983.

† Computer Science Program, Department of Mathematical Sciences, Rice University, Houston, Texas 77251. This research was partially supported by the National Science Foundation under grants MCS76-14293, MCS78-05850, and MCS81-04209.

[1] A brief synopsis of the important definitions from mathematical logic (such as theory) appears in the next section.

2. Logical preliminaries. As a foundation for the remainder of the paper, we briefly summarize the important definitions and notational conventions of first order logic. Readers who are unfamiliar with the fundamental concepts of first order predicate calculus are encouraged to consult Enderton's excellent introductory text [14].

In first order programming logic, recursive definitions are expressed within a conventional *first order logical language L with equality* determined by a countable set of function symbols G, a countable set of predicate symbols R, and an associated "arity" function $\#: G \cup R \to \mathrm{Nat}$ (where Nat denotes the set of natural numbers) specifying the *arity* $\# p$ (required number of arguments) for each function and predicate symbol p. Nullary function symbols serve as constants. The function and predicate symbols are the names of the primitive operations of the data domain. The first order language L determined by G, R, and $\#$ contains two classes of strings: a set of *terms* constructed from variables and function symbols G, and a set of *formulas* constructed from predicate symbols $\{=\} \cup R$ applied to terms (forming *atomic formulas*) and from logical connectives $\{\forall, \wedge, \vee, \neg\}$ applied to simpler formulas. Each function and predicate symbol p is constrained to take exactly $\# p$ arguments.

A context free grammar defining the (context free) syntax of terms and formulas appears below.

$$
\begin{array}{ll}
\langle\text{term}\rangle & \to \langle\text{constant}\rangle \mid \langle\text{variable}\rangle \mid \langle\text{function-symbol}\rangle\,(\langle\text{termlist}\rangle) \\
\langle\text{termlist}\rangle & \to \langle\text{term}\rangle \mid \langle\text{term}\rangle, \langle\text{termlist}\rangle \\
 & \\
\langle\text{atomic-formula}\rangle & \to \langle\text{predicate-symbol}\rangle\,(\langle\text{termlist}\rangle) \mid \langle\text{term}\rangle = \langle\text{term}\rangle \\
\langle\text{formula}\rangle & \to \langle\text{atomic-formula}\rangle \mid \forall\langle\text{variable}\rangle\langle\text{formula}\rangle \mid \neg\langle\text{formula}\rangle \mid \\
 & \quad (\langle\text{formula}\rangle \wedge \langle\text{formula}\rangle) \mid (\langle\text{formula}\rangle \vee \langle\text{formula}\rangle)
\end{array}
$$

An occurrence of a variable v in a formula α is *bound* if the occurrence is contained within a subformula of the form $\forall v\beta$ or $\exists v\beta$. An occurrence of a variable is *free* iff it is not bound. Terms and formulas containing no occurrences of free variables are called *variable-free terms* and *sentences*, respectively. Let $\alpha(x)$ denote a formula possibly containing the variable x and let t denote an arbitrary term. Then $\alpha(t)$ denotes the formula obtained from $\alpha(x)$ by replacing every free occurrence of x by t.

The additional logical connectives $\{\oplus, \supset, \equiv, \exists, \exists!\}$ are defined as abbreviations for combinations of primitive connectives as follows

$$
\begin{array}{lll}
(\alpha \oplus \beta) & \text{abbreviates} & ((\alpha \wedge \neg\beta) \vee (\neg\alpha \wedge \beta)) \\
(\alpha \supset \beta) & \text{abbreviates} & (\neg\alpha \vee \beta) \\
(\alpha \equiv \beta) & \text{abbreviates} & ((\alpha \supset \beta) \wedge (\beta \supset \alpha)) \\
\exists v\alpha & \text{abbreviates} & \neg\forall v \neg\alpha \\
\exists! v\alpha(v) & \text{abbreviates} & \exists v(\alpha(v) \wedge \forall u(\alpha(u) \supset u = v))
\end{array}
$$

where α and β denote arbitrary formulas and v denotes an arbitrary variable.

A formula with elided parentheses abbreviates the fully parenthesized formula generated by giving unary connectives precedence over binary ones, ranking binary connectives in order of decreasing precedence: $\{\wedge\} > \{\vee, \oplus\} > \{\supset\} > \{\equiv\}$, and associating adjacent applications of connectives of equal precedence to the right. For the sake of clarity, we will occasionally substitute square brackets {[,]} for parentheses within formulas. In place of a sentence, a formula α abbreviates the sentence $\forall \bar{v}\alpha$ where $\bar{v}$ is a list of the free variables of α. Similarly, the forms $\forall x : p\,\alpha$ and $t : p$, where p is a unary predicate symbol, abbreviate the formulas $\forall x[p(x) \supset \alpha]$ and $p(t)$, respectively.

Let S denote a (possibly empty) set of function and predicate symbols (with associated arities) not in the language L. Then $L \cup S$ denotes the first order language

determined by the function and predicate symbols of L augmented by S; $L \cup S$ is called an *expansion* of L.

Although logicians occasionally treat first order logic as a purely syntactic system (the subject of *proof theory*), we are interested in what terms and formulas *mean.* The meaning of a first order language L is formalized as follows. A *structure* **M** *compatible with* L is a triple $\langle |M|, \mathbf{M_G}, \mathbf{M_R} \rangle$ where $|M|$ (called the *universe*) is a set of (data) values; $\mathbf{M_G}$ is a function mapping each function symbol $g \in G$ into a $\# g$-ary function on $|M|$; and $\mathbf{M_R}$ is a function mapping each predicate symbol $r \in R$ into a $\# r$-ary *predicate* on $|M|$—a function mapping $|M|^{\# r}$ into the set Tr of truth values {**TRUE, FALSE**}. The universe $|M|$ must be disjoint from Tr. Given a structure **M** compatible with L and a *state* s (often called an *interpretation function*) mapping the variables of L into $|M|$, every term in L denotes an object in $|M|$ and every formula denotes a truth value. The meaning of terms and formulas of L is defined by structural induction in the obvious way; for a rigorous definition, consult Enderton's text.

Let H be a subset of the function symbols of the first order language L. A structure **M** compatible with the language L is called an *H-term structure* iff the universe $|M|$ consists of equivalence classes of variable-free terms in L constructed solely from the function symbols in H. A structure compatible with L is *finitely generated* iff there exists a finite subset H of the function symbols of L such that **M** is isomorphic to an H-term structure.

Let **M** be a structure compatible with the language L and let **S** denote a set of functions and predicates over $|M|$ interpreting a set S of function and predicate symbols not in L. Then $\mathbf{M} \cup \mathbf{S}$ denotes the structure consisting of **M** augmented by the functions and predicates **S**; $\mathbf{M} \cup \mathbf{S}$ is called an *expansion* of **M**.

In mathematical logic, it is often important to make a clear distinction between a function symbol and its interpretation. To cope with this issue, we will use the following notation. Function symbols appear in ordinary type and stand for themselves. In contexts involving a single structure **M**, a function or predicate symbol p written in **boldface (p)** denotes $\mathbf{M_G}(p)$ or $\mathbf{M_R}(p)$, the interpretation of p in **M**. In more general contexts, $\mathbf{M}[p]$ denotes the interpretation of the symbol p in the structure **M**. Similarly, $\mathbf{M}[\alpha][s]$ denotes the meaning of the formula or term α in **M** under the state s. If α is a variable-free term or a sentence, then its meaning in a structure is independent of the particular choice of state s. In this case, the abbreviated notation $\mathbf{M}[\alpha]$ denotes the meaning of α in **M**.

Let T be a set of sentences in the first order language L. A *model* of T is a structure **M** compatible with L such that every sentence of T is **TRUE** in **M**. We say that a structure **M** *satisfies* T or alternatively, that T is an *axiomatization* of **M**, iff **M** is a model of T. The set of sentences T forms a *theory* iff it satisfies the following two properties:

(i) *Semantic consistency*: there exists a model of T.

(ii) *Closure under logical implication*: every sentence that is **TRUE** in all models of T is a member of T.

Given a structure **M** compatible with L, the set of sentences in L that are **TRUE** in **M** (denoted Th **M**) obviously forms a theory; it is called the *theory of* **M**. Given an arbitrary set A of sentences of L, the *theory generated by* A is the set of sentences that are logically implied by A. A theory T is *axiomatizable* iff there exists a recursive set of sentences $A \subseteq T$ such that A generates T. In this case, the set of sentences A is called an *effective axiomatization* of T.

A theory T typically has an intended model called the *standard model.* Any model that is not isomorphic to the standard model is called a *nonstandard model.* Two

structures compatible with the same language L are *elementarily distinct* iff there exists a sentence S in L such that S is true in one structure but not in the other. A theory is *incomplete* iff it has elementarily distinct models; otherwise, it is *complete*. For any structure $\mathbf{M}$, $Th\ \mathbf{M}$ is obviously complete.

Given a recursively enumerable set of axioms A, there is a mechanical procedure for enumerating all of the sentences that are logically implied by A. A *first order deductive system* Γ is a finite set of syntactic rules (often formulated as productions in a phrase structure grammar) that generates a set of sentences from A. A *proof* of a sentence α from A in the deductive system Γ is simply its derivation in Γ from A. A deductive system Γ is *sound* iff every sentence derivable from an axiom set A is logically implied by A. A deductive system Γ is *complete* iff every sentence in the theory generated by A is derivable (provable) in Γ from A. A remarkable property of first order logic is the existence of sound, complete deductive systems for arbitrary axiom sets A. Higher order logics generally do not share this property.

There are many different ways to formulate a sound, complete deductive system for first order logic. Two approaches that are well known to computer scientists are resolution and Gentzen natural deduction [17]. Of course, every first order deductive system that is sound and complete derives exactly the same set of sentences. In this paper, we will leave the choice of deductive system unspecified, since we are not interested in the syntactic details of formal proofs. In our examples, we will present proofs in informal (yet rigorous) terms that readily translate into formal proofs in a Gentzen natural deduction system.

Let $\mathbf{A}$ and $\mathbf{B}$ be two structures compatible with the languages L_A and L_B, respectively, where $L_A \subseteq L_B$ (i.e., L_B is an expansion of L_A). $\mathbf{B}$ is an *extension* of A iff $|B| \supseteq |A|$ and every operation (function or predicate) of $\mathbf{A}$ is the restriction of the corresponding operation of $\mathbf{B}$ to $|A|$. If $|B|$ is identical to $|A|$, then $\mathbf{B}$ is obviously an *expansion* of $\mathbf{A}$. Otherwise, $|B|$ properly contains $|A|$, and $\mathbf{B}$ is called a *proper extension of* $\mathbf{A}$.

Let $S = \{s_1, \cdots, s_n\}$ be a finite set of function and predicate symbols not in the language L_A. A *definition for S over the structure* $\mathbf{A}$ is a collection of sentences Δ in the language $L_A \cup S$ such that $\mathbf{A}$ can be expanded—by adding interpretations for the new function and predicate symbols in S—to a model for Δ. An *unambiguous definition for S over* $\mathbf{A}$ is a definition that determines a *unique* expansion of $\mathbf{A}$.

A formula $\alpha(x_1, \cdots, x_k)$ in L_A *defines the k-ary predicate* $\mathbf{r}$ *in* $\mathbf{A}$ iff α contains no free variables other than $x_1, \cdots, x_k$ and for all states s over $|A|$, $\mathbf{A}[\alpha(x_1, \cdots, x_k)][s] = \mathbf{r}(s(x_1), \cdots, s(x_k))$. Similarly, a formula $\alpha(x_1, \cdots, x_k, y)$ in L_A *defines the k-ary function* $\mathbf{g}$ in $\mathbf{A}$ iff α contains no free variables other than $x_1, \cdots, x_k$ and for all states s over $|A|$, $\mathbf{A}[\alpha(x_1, \cdots, x_k, y)][s] = \mathbf{TRUE}$ iff $\mathbf{g}(s(x_1), \cdots, s(x_k)) = s(y)$. A set $\mathbf{S} = \{\mathbf{s_1}, \cdots, \mathbf{s_n}\}$ of predicates and functions interpreting the symbols S is *definable in* $\mathbf{A}$ iff there exist formulas $\alpha_1, \cdots, \alpha_n$ defining $\mathbf{s_1}, \cdots, \mathbf{s_n}$, respectively.

Let T be a semantically consistent set of sentences in the language L_A and let $\mathbf{A}$ be a model of T. A *definition for S augmenting T* is a collection of sentences Δ in the language $L_A \cup S$ such that every model of T can be expanded—by adding interpretations for the new function and predicate symbols in S—to a model for $T \cup \Delta$. An *unambiguous definition for S augmenting T* is a definition that determines a *unique* expansion in *every* model of T. Note that a definition for S over a model of T is not necessarily a definition augmenting T. Similarly, an unambiguous definition for S over a model of T is not necessarily an unambiguous definition augmenting T.

A set $\mathbf{S} = \{\mathbf{s_1}, \cdots, \mathbf{s_n}\}$ of predicates and functions over $|A|$ interpreting the symbols S is *implicitly definable in the theory generated by T* iff there an unambiguous definition

Δ for S augmenting T such that $\mathbf{S}$ is interpretation of S determined by Δ in the structure $\mathbf{A}$. The set $\mathbf{S}$ is *explicitly definable in T* iff there exists a set of formulas $\alpha_1, \cdots, \alpha_n$ in L_A, defining $\mathbf{s_1}, \cdots, \mathbf{s_n}$, respectively, in $\mathbf{A}$. One of the most important results in the theory of definitions, Beth's Definability Theorem [2], asserts that a set $\mathbf{S}$ of functions and predicates over $|A|$ is implicitly definable in T iff it is explicitly definable in T. Hence, we are justified in dropping the modifiers "implicitly" and "explicitly" when discussing the issue of definability in a theory.

In first order programming logic, we formalize data domains as structures in first order logic. In this context, a recursive program is simply a particular form of logical definition over the data domain. Before proceeding with the development of the formal theory, we will first examine and refute a widely accepted argument asserting that first order logic is incapable of expressing and proving that functions defined in recursive programs are total.

3. Hitchcock and Park's critique of first order logic. As motivation for developing a higher order logic for reasoning about recursive programs, Hitchcock and Park [16] claim that first order logic is too weak to express and prove that the functions defined in a recursive program are total. As justification, they consider the following recursive program over the natural numbers:

$$\text{(1)} \qquad \text{zero}\,(n) = \text{IF } n = 0 \text{ THEN } 0 \text{ ELSE zero}\,(n-1)$$

where IF-THEN-ELSE is interpreted as a logical connective (as in reference [17]). This program (1) can be expressed within the usual language of first order number theory (eliminating the special IF-THEN-ELSE connective) by the sentence:

$$\text{(2)} \qquad \forall n[(n = 0 \supset \text{zero}\,(n) = 0) \wedge (n \neq 0 \supset \text{zero}\,(n) = \text{zero}\,(n-1))].$$

While they concede that it is very easy to prove informally by induction that the zero function is total on the natural numbers they claim that no sentence provable in a first order theory of the natural numbers augmented by (2) can state that zero is total. To justify this claim, they propose the following argument.

Let $\mathbf{N}$ denote the structure consisting of the natural numbers, the constants (0-ary functions) $\{\mathbf{0}, \mathbf{1}\}$, the binary functions $\{+, \times, -\}$ and the binary predicates $\{=, <\}$. By the upward Lowenheim–Skolem theorem, the theory (set of true sentences) of $\mathbf{N}$ has a nonstandard model $\bar{\mathbf{N}}$ that is a proper extension of $\mathbf{N}$. The additional objects in the universe $|\bar{N}|$ are "nonstandard" natural numbers that are greater than all standard integers (the elements of the universe $|N|$). Hitchcock and Park assert that the recursive definition for zero obviously does not terminate for all elements of $|\bar{N}|$, since the nonstandard numbers in this model have infinitely many predecessors. Given this assertion, no sentence θ provable in a first order theory for $\mathbf{N}$ can state zero is total since θ must be true in $\bar{\mathbf{N}}$.

The flaw in Hitchcock and Park's analysis is their assumption that the interpretation of the function symbol zero in a nonstandard model must be obtained by applying standard computation (reduction) rules to (1). In the theory of program schemes [15], where the concept of program execution is embedded in the formalism (just as the meaning of logical connectives such as $\wedge$ and $\vee$ is embedded in first order logic), this point of view makes sense. But in first order logic, there is no notion of execution constraining the interpretation of recursively defined functions. Recursive definitions are simply equations that introduce new function symbols; they do not necessarily have a computational interpretation. In fact, they may have no interpretation at all (see example (4) below). In first order programming logic, we prevent potential

inconsistencies by restricting logical theories to a form that guarantees that arbitrary recursive definitions have computational interpretations in the standard model.

We can gain additional insight into the difference between first order logic and the theory of program schemes by examining Hitchcock and Park's example in more detail. Let A be the standard first order Peano axiomatization for the natural numbers including an axiom scheme expressing the induction principle (such an axiomatization appears in Appendix I). Given A and the recursive definition of zero, we can easily prove the sentence

(3) $\forall n[\text{zero}(n)=0]$

by induction on n[2]. Both the base case and induction step are trivial consequences of (2). Consequently, the function zero defined by (2) is identically zero in every model of A—including $\bar{\mathbf{N}}$. Furthermore, since the models of A do not contain an object (usually denoted $\perp$) representing a divergent computation,[3] all of them must, by definition, interpret every function symbol by a (total) function on the universe of the model (the set of standard or nonstandard natural numbers). Hence, no recursion equation augmenting A can define a nontotal function in any model of A. For the same reason, some recursion equations such as

(4) $f(x)=f(x)+1$

define no function at all because they are inconsistent with the original theory.

The situation is more interesting if we start with an axiomatization of the structure $\mathbf{N}^+$ consisting of $\mathbf{N}$ augmented by the undefined object $\perp$, instead of an axiomatization for $\mathbf{N}$. In this case, the interpretation for a function symbol f may be partial in the sense that it maps some elements of the data domain into $\perp$. Note that $\perp$ is an ordinary constant which is forced by the axiomatization to behave like a "divergent" or "undefined" data object. It is *not* a new logical primitive.

Within the first-order language for $\mathbf{N}^+$, we can assert that f is total on $|N|$ by simply stating

$$\forall x_1, \cdots, x_n[(x_1 \neq \perp) \wedge \cdots \wedge (x_n \neq \perp) \supset f(x_1, \cdots, x_n) \neq \perp].$$

Let A^+ be an axiomatization (including an induction axiom scheme) for $\mathbf{N}^+$ analogous to Peano's axioms (in first order form) for $\mathbf{N}$. A suitable formulation of A^+ appears in Appendix I. Given A^+ and the recursive definition (2), we can easily establish that the zero function is total on $|N|$ by proving the sentence

$$\forall n[n \neq \perp \supset \text{zero}(n) \neq \perp].$$

The proof (which appears in Appendix II) is a direct translation of the informal structural induction proof that Hitchcock and Park cite in their paper. Consequently, we are forced to conclude that a careful analysis of Hitchcock and Park's example actually supports the thesis that the totality of recursively defined functions can be naturally expressed and proven within first order logic. We will rigorously establish this result in the next section.

4. Basic concepts of first order programming logic. As we suggested in the previous section, the undefined object $\perp$ plays a crucial role in first order programming

[2] Note that adding new function symbols to L implicitly augments A by additional instances of the induction axiom scheme (containing the new symbols).

[3] No object in *any* model of A—including those within nonstandard integers—has the same properties as the divergent object $\perp$. For instance, successor $(\perp)=\perp$, yet in any model of A, $\forall x$ successor $(x) \neq x$.

logic, just as it does in higher order logics such as LCF [15], [19], [20], [21]. If we fail to include the undefined object $\perp$ in the data domain, recursive definitions like

$$f(x)=f(x)+1$$

on the natural numbers $\mathbf{N}$ are inconsistent with the axiomatization of the domain; the interpretation of f must be a (total) function on the natural numbers, yet no such function exists.

Consequently, first order programming logic imposes certain constraints on the data domain (and hence on any corresponding theory). In particular, the program data domain must be *continuous.* The following collection of definitions defines this property and several related concepts.

DEFINITION. A *complete partial ordering* $\subseteq$ on a set S is a binary relation over S such that:

(i) $\subseteq$ is a partial ordering on S (a reflexive, antisymmetric, and transitive relation on S).
(ii) The set S contains a least element $\perp$ (under the partial ordering $\subseteq$).
(iii) Every chain (denumerable sequence ordered by $\subseteq$) $x_0 \subseteq x_1 \subseteq x_2 \subseteq \cdots$ has a least upper bound.

A set S with a corresponding complete partial ordering $\subseteq$ is called a *complete partial order* (abbreviated *cpo*); the partial ordering $\subseteq$ is called the *approximation ordering* for S.

DEFINITION. Given cpo's A and B, a function $f: A \to B$ is *continuous* iff the image of an arbitrary chain $X = x_0 \subseteq x_1 \subseteq x_2 \subseteq \cdots$ in A is a chain in B and the image of the least upper bound of X is the least upper bound of the chain image.

There are two standard methods for building composite cpo's from simpler ones. First, given the cpo's $A_1, \cdots, A_m$ under the approximation orderings $\subseteq_1, \cdots, \subseteq_m$, respectively, the Cartesian product $A_1 \times \cdots \times A_m$ forms a cpo under the ordering $\subseteq$ defined by

$$\bar{x} \subseteq \bar{y} \equiv \bigwedge_{1 \leq i \leq n} [x_i \subseteq_i y_i].$$

Second, given the cpo A under $\subseteq_A$ and the cpo B under $\subseteq_B$, the set of continuous functions mapping A into B forms a cpo under the ordering $\subseteq$ defined by

$$g \subseteq h \equiv \forall \bar{x} \in A[g(\bar{x}) \subseteq_B h(\bar{x})].$$

DEFINITION. A structure $\mathbf{D}$ including the constant $\perp$ is *continuous* under the binary relation $\subseteq$ on $|D|$ iff $|D|$ forms a complete partial order under $\subseteq$ and every function $\mathbf{f}: |D|^{\#f} \to |D|$ in $\mathbf{D}$ is continuous.

DEFINITION. Given a continuous data domain $\mathbf{D}$ compatible with the language L_D determined by the function symbols G and the predicate symbols R, a *recursive program* P over $|\mathbf{D}|$ has the form:

$$\{f_1(\bar{x}_1) = t_1, f_2(\bar{x}_2) = t_2, \cdots, f_n(\bar{x}_n) = t_n\}$$

where $n > 0$; the set F of function symbols $\{f_1, f_2, \cdots, f_n\}$ is disjoint from $G \cup R$; $\bar{x}_1, \bar{x}_2, \cdots, \bar{x}_n$ are lists of variables; and $t_1, t_2, \cdots, t_n$ are terms in the language $L_D \cup F$ such that each term t_i contains no variables other than those in $\bar{x}_i$. The intended *meaning* of the n-tuple of function symbols $[f_1, \cdots, f_n]$ introduced in the program P is the least fixed-point of the functional

$$\mathbf{P} = \lambda f_1, \cdots, f_n \cdot [\lambda \bar{x}_1 \cdot t_1, \cdots, \lambda \bar{x}_n \cdot t_n]$$

corresponding to P.

By Kleene's recursion theorem (most broadly formulated by Tarski [27]), **P** must have a least fixed-point $[\mathbf{f}_1, \cdots, \mathbf{f}_n]$, because it is a continuous mapping from the cpo$(|D|^{\#f_1} \to |D|) \times \cdots \times (|D|^{\#f_n} \to |D|)$ into itself. A proof that **P** is continuous can be found in either Cadiou [5] or Vuillemin [28].

Although continuity ensures that recursive programs are well-defined, it does not guarantee that they can be implemented on a machine. For this reason, program data domains typically satisfy several additional constraints which we lump together under the label *arithmeticity*. The most important difference between an arithmetic domain and a continuous domain is that the former must be finitely generated. First order programming logic critically depends on this property, because it presumes that the domain obeys the principle of structural induction. The remaining properties that distinguish arithmetic domains from continuous ones (items (i) and (iii) in the definition below) are not essential; they are included solely to simplify the exposition.

DEFINITION. A structure **D** is *flat* iff it is continuous under the binary relation $\subseteq$ defined by the identity

$$a \subseteq b \equiv [a = b \vee a = \bot].$$

DEFINITION. A continuous function $f: A_1 \times \cdots \times A_m \to B$ is *strict* iff $f(x_1, \cdots, x_m) = \bot$ when any argument $x_i = \bot$.

DEFINITION. Let **D** be a data domain (structure) compatible with the language L_D determined by the function symbols G and the predicate symbols R. **D** is an *arithmetic domain* iff it satisfies the following three properties:

(i) **D** is flat.

(ii) **D** is finitely generated. Hence, every element $d \in |D|$ has at least one *name* consisting of a variable-free term α such that $\mathbf{D}[\alpha] = d$. Note that the finite generation property implies that **D** obeys induction on the structure of names (often called "structural induction" or "generator induction"). We can formulate this principle as follows. Let Gen $= \{g_1, \cdots, g_k\}$ denote a minimal subset of G (the function symbols of L) that generates $|D|$. Generator induction asserts that for every unary predicate $\varphi(x)$ over **D**,

$$(*) \quad \left[\bigwedge_{1 \leq i \leq k} \forall x_1, \cdots, x_{\#g_i}[\varphi(x_1) \wedge \cdots \wedge \varphi(x_{\#g_i}) \supset \varphi(g_i(x_1, \cdots, x_{\#g_i}))]\right] \supset \forall x\, \varphi(x).$$

In the literature on programming languages, the generator symbols Gen are often called constructors. Note that a minimal set of generators Gen for a domain **D** is not necessarily unique.

(iii) The set of functions **G** includes the constants {**true, false**} and the special function **if-then-else** which partitions $|D|$ into three nonempty disjoint subsets D_{true}, D_{false}, $D_\bot$ such that

$$\begin{aligned}
&\mathbf{true} \in D_{\text{true}}\\
&\mathbf{false} \in D_{\text{false}}\\
&\bot \in \mathrm{D}_\bot\\
&\mathbf{if}\ p\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta = \alpha \text{ if } p \in D_{\text{true}}\\
&\mathbf{if}\ p\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta = \beta \text{ if } p \in D_{\text{false}}\\
&\mathbf{if}\ p\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta = \bot \text{ if } p \in D_\bot.
\end{aligned}$$

All functions in **G** other than **if-then-else** must be strict.

With the exception of the induction principle (*) appearing in property (ii), the preceding list of conditions on **D** can be formally expressed by a finite set of sentences in the language L_D. The induction principle (*) cannot be expressed in L_D, because it

asserts that induction holds for all unary predicates—an uncountable set with many members that cannot be defined within L_D. We will explore this issue in depth in § 7.

Fortunately, confining our attention to arithmetic data domains does not significantly limit the applicability of first order programming logic. With the exception of domains including an extensional treatment of higher order data objects (such as functions), the data domain of any plausible recursive programming language has a natural formalization as an arithmetic structure. At the end of this section, we will discuss how to extend an arbitrary, finitely generated domain **D** excluding $\perp$ to form an arithmetic domain $\mathbf{D}'$ with universe $|D| \cup \{\perp\}$.

Before we state and prove the fundamental theorem of first order programming logic, we must resolve a subtle issue concerning the status of induction in arithmetic domains that are augmented by definitions. Formalizing induction in first order logic requires an axiom scheme: a template with a free formula parameter. The scheme represents the infinite recursive set of sentences consisting of all possible instantiations of the template. Let **D** be a structure that is finitely generated by the function symbols $\text{Gen} = \{g_1, \cdots, g_k\}$. Obviously, the corresponding induction principle (*) holds in **D**. In a first order axiomatization A_D for **D**, we typically include the following axiom scheme formalizing the induction principle (*)

$$(**) \quad \left[\bigwedge_{1 \leqq i \leqq k} \forall x_1, \cdots, x_{\#g_i}[\varphi(x_1) \wedge \cdots \wedge \varphi(x_{\#g_i}) \supset \varphi(g_1(x_1, \cdots, x_{\#g_i}))]\right] \supset \forall x\, \varphi(x)$$

where $\varphi(x)$ is an arbitrary formula in L_D defining a unary predicate. The scheme asserts that structural induction holds for every *definable* unary predicate in the domain. Any structure satisfying the structural induction scheme (**) is called an *inductive domain* with generator set Gen. The only difference between an inductive domain and a finitely generated domain is that induction may fail in an inductive domain for predicates that are not definable.

When we augment a finitely generated domain **D** by a definition Δ introducing new function and predicate symbols S, how should we interpret the induction scheme? Does the formula parameter in the scheme range over formulas in the augmented language or formulas in the original language? Assuming that we are interested in constructing the strongest possible theory for the expanded structure, the answer to the question is clear. Since the universe of the expanded structure is identical to the universe of the original domain, the induction scheme must hold for all predicates that are definable in the expanded structure (using the augmented language). Consequently, we follow the convention that a definition Δ over a finitely generated domain **D** implicitly includes all of the new instances of the structural induction scheme (**) for **D** corresponding to the language extension. In this context, $A_D \cup \Delta$ denotes the set of axioms containing A_D, Δ, and *all new instances* of the induction scheme (**). For reasons that will become clear when we discuss nonstandard models, we follow exactly the same convention for definitions over inductive domains: a definition Δ over an inductive domain **D** implicitly includes all new instances of the induction scheme (**) for **D**. To emphasize this convention, we will use the term *arithmetic definition* to refer to any definition that implicitly includes new instances of the corresponding induction scheme.

THEOREM (fundamental theorem). *Let P be a recursive program*

$$\{f_1(\bar{x}_1) = t_1, f_2(\bar{x}_2) = t_2, \cdots, f_n(\bar{x}_n) = t_n\}$$

over an arithmetic domain **D**, *and let F denote the least fixed-point of the functional of P. Then P is an arithmetic definition over* **D** *satisfying the model* $\mathbf{D} \cup \mathbf{F}$.

Proof. By Kleene's recursion theorem [27], the functional for P has a least fixed-point $\mathbf{F}=[\mathbf{f_1},\cdots,\mathbf{f_n}]$. Hence,

$$\bigwedge_{1\leqq i\leqq n} [\mathbf{f_i}=\lambda\bar{x}_i\cdot\mathbf{t_i}],$$

where $\mathbf{t_i}$ denotes the interpretation of t_i given that the primitive function symbols in t_i are interpreted by the corresponding functions in $\mathbf{D}$ and the function symbols $f_1,\cdots,f_n$ are interpreted by $\mathbf{f_1},\cdots,\mathbf{f_n}$, respectively. This fact can be restated in the form

$$\bigwedge_{1\leqq i\leqq n} \mathbf{D}\dagger[f_i(\bar{x}_i)][s]=\mathbf{D}\dagger[t_i][s],$$

where $\mathbf{D}\dagger$ denotes the structure $\mathbf{D}\cup\mathbf{F}$, and s is an arbitrary state over $|D\dagger|$. Since the universe $|D\dagger|$ is identical to the universe $|D|$, the induction principle (*) holds for all unary predicates $\varphi(x)$ over $|D\dagger|$ including those defined by formulas in $L_D\cup\{f_1,\cdots,f_n\}$. Hence, $\mathbf{D}\dagger$ is a model for $P\cup$(**) extending $\mathbf{D}$. □

This theorem formally establishes that we can interpret recursive programs as definitions in first order logic. Consequently, given a recursive program P over an arithmetic domain $\mathbf{D}$, we can prove properties of P by applying ordinary first order deduction to a suitable first order axiomatization A_D of $\mathbf{D}$ (including the structural induction scheme (**) for $\mathbf{D}$) augmented by the equations in P (which are simply first order formulas). Using this approach, we can prove almost any property of practical importance about P, including totality. Most proofs strongly rely on structural induction.

A minor impediment to the practical application of first order programming logic is the fact that most axiomatizations appearing in the literature (e.g., the first order formulation of Peano's axioms) specify domains that exclude the special object $\perp$. In structures that are not specifically intended to serve as domains for computation, an object representing a divergent computation is superfluous.

Fortunately, it is easy to transform a first order axiomatization A_D for a finitely generated data domain $\mathbf{D}$ that excludes $\perp$ into an axiomatization A'_D for a corresponding arithmetic domain $\mathbf{D}'$ that includes $\perp$. Although the syntactic details of the transformation are beyond the scope of this paper, the main features warrant discussion. The transformation breaks down into three parts.

First, to satisfy the continuity property required by Kleene's theorem, the transformation designates two distinct elements of $|D|$ as the constants {**true, false**}, adds the undefined object $\perp$ to $|D|$, and replaces the computable predicates of $\mathbf{D}$ (those that can appear in program text) of the data domain by corresponding strict boolean functions.

Second, the transformation extends each primitive function $\mathbf{g}$ in $\mathbf{D}$ to its strict analog over $\mathbf{D}\cup\{\perp\}$. Specifically, for each axiom in the original set A_D, the transformation generates a corresponding axiom with restrictive hypotheses that prevent variables from assuming the value $\perp$. The transformation also generates new axioms asserting that each function $\mathbf{g}$ is strict.

Third, to accommodate nontrivial recursive definitions, the transformation adds the standard ternary conditional function **if-then-else** to the collection of primitive functions. While **if-then-else** is not strict (since **if true then** x **else** $\perp = x$), it is continuous. Without **if-then-else**, every recursive function definition (that actually utilizes recursion) diverges for all inputs, because all the other primitive functions are strict.

The new data domain $\mathbf{D}'$ retains the structure of the original one, yet it is clearly an arithmetic domain. Appendix I presents a sample axiomatization for a data domain (the natural numbers) that does not include $\perp$ and transforms it into one for an arithmetic domain that does.

5. A sample proof. As an illustration of the utility of first order programming logic, consider the following simple example. Let flat and flat1 be recursively defined functions over the domain of LISP S-expressions defined by the following equations:

$$\text{flat}(x) = \text{flat1}(x, \text{NIL})$$
$$\text{flat1}(x, y) = \text{if atom } x \text{ then cons}(x, y) \text{ else flat1}(\text{car } x, \text{flat1}(\text{cdr } x, y)).$$

The function flat returns a linear "in-order" list of the atoms appearing in the S-expression x. For example

$$\text{flat}[(A \cdot B)] = (A\ B),$$
$$\text{flat}[(A \cdot (B \cdot A))] = (A\ B\ A),$$
$$\text{flat}[(A)] = (A),$$
$$\text{flat}[((A \cdot C) \cdot B)] = (A\ C\ B).$$

We want to prove that flat1 (x, y) terminates for arbitrary S-expressions x and y (obviously implying flat (x) is total for all S-expressions x). In the theory of S-expressions augmented by $\{\bot\}$, we can formally state and prove this property in the following way, given that Sexpr (x) abbreviates the formula $x \neq \bot$.

THEOREM. $\forall x, y : \text{Sexpr}[\text{flat1}(x, y) : \text{Sexpr}]$.

Proof. We prove the theorem by structural induction on x.

Basis: x is an atom.

Simplifying flat1 (x, y) yields cons (x, y) which must be an S-expression since x and y are S-expressions.

Induction step: x has the form cons (hd, tl) where hd:Sexpr and tl:Sexpr. Given the hypotheses

(a) $\forall y : \text{Sexpr}[\text{flat1}(\text{hd}, y) : \text{Sexpr}]$, and
(b) $\forall y : \text{Sexpr}[\text{flat1}(\text{tl}, y) : \text{Sexpr}]$,

we must show

$$\forall y : \text{Sexpr}[\text{flat1}(\text{cons}(\text{hd}, \text{tl}), y) : \text{Sexpr}].$$

Since hd, tl, and y are S-expressions,

$$\text{flat1}(\text{cons}(\text{hd}, \text{tl}), y) = \text{flat1}(\text{hd}, \text{flat1}(\text{tl}, y)).$$

By induction hypothesis (a), flat1 (tl, y) is an S-expression. Given this fact, we immediately deduce by induction hypothesis (b) that flat1 (hd, flat (tl, y)) is an S-expression. □

Some additional examples appear in Appendix II.

6. Computations in first order programming logic. Although we have shown that recursive programs can be interpreted as definitions over an arithmetic data domain, we have not yet established that there is a plausible definition of computation that is consistent with our logical interpretation. Since conventional first order logic does not include any notion of computation (proof is the closest analogue), we must invent one specifically for first order programming logic. Fortunately, there is a simple syntactic definition of the concept based on "term rewriting systems" that makes sense in the context of first order logic. The critical idea is that computation is a uniform (possibly nonterminating) procedure for transforming a variable-free term into its meaning[4] in the standard model using ordinary first order deduction.

[4] More precisely, into a "canonical" term denoting its meaning.

Term rewriting systems for recursive programs have been extensively investigated by Cadiou [4], Vuillemin [28], [29], Rosen [24], Downey and Sethi [13], and O'Donnell [22], but not in the context of first order theories of program data. The following formulation of first order computation is a distillation and adaptation of the work of Cadiou [5] and Vuillemin [28], [29] recast in the terminology of first order logic.

DEFINITION. A structure **D** with function set **G** is *effective* iff it satisfies the following two conditions:

(i) Every element $d \in |D|$ has a unique canonical name can (d) that is a variable-free term denoting d in the first order language L_D excluding if-then-else. For simplicity, we require that can $(\perp)$ be $\perp$. In addition, the set of canonical names must be a recursive subset of the set of all variable-free terms in L_D.

(ii) The graph of every function **g** in **G** is recursive, given that we denote objects of the universe $|D|$ by their canonical names.

All data domains in conventional programming languages satisfy these constraints.

DEFINITION. Let P be an arbitrary recursive program over an effective, arithmetic domain **D** defining the function symbols $F=\{f_1, \cdots, f_n\}$, and let L_P be the language $L_D \cup F$. A set Φ of *productions* (*or rewrite rules*) *over* L_P is a set of ordered pairs $u \to v$ where u and v are variable-free terms of L_P. Let Υ denote the set of left-hand sides of Φ: $\{u | u \to v \in \Phi\}$. Φ is *effective* iff the following three conditions hold.

(i) Every left-hand side in Υ corresponds to a unique production in Φ.

(ii) Υ is a recursive subset of L_P.

(iii) Every noncanonical variable-free term t in L_P contains a subterm in Υ.

DEFINITION. Let Φ be an effective set of productions in the language L_P. Given an arbitrary variable-free term t in L_P, the Φ-*reduction of* t is the countable (finite or infinite) sequence of variable-free terms $\tau = t_0, \cdots, t_k, \cdots$ such that $t_0 = t$, each term t_i has a successor if and only if it is noncanonical, and each successor term t_{i+1} is generated from its predecessor t_i by locating the *left-most* subterm that matches a left-hand side of some production α in Φ and replacing it by the right-hand side of α. The *result* of a reduction τ is the *meaning* in **D** of the last term t' of τ ($\mathbf{D}[t']$) if τ is finite and $\perp$ otherwise.

Remark. A reduction τ is finite iff the last term in τ is a canonical name.

6.1. Call-by-name computation. The reduction scheme that directly corresponds with the logical meaning of recursive programs (as defined in § 4) is called *call-by-name* computation.

DEFINITION. Let L_P be the first order language corresponding to an effective, arithmetic domain **D** augmented by a recursive program P defining the function symbols $F=\{f_1, \cdots, f_n\}$. The *call-by-name production set* Φ_P for P is the set $\Phi_G \cup \Phi_F \cup \Phi_{if}$, where Φ_G, Φ_F, and Φ_{if} are defined as follows.

(i) Φ_G is the set of productions

$$\{g(\bar{c}) \to v \mid g \in G - \{\text{if-then-else}\};\ \bar{c} \text{ is a } \#g\text{-tuple of canonical names}; \\ v \text{ is the canonical name for } \mathbf{g}(\bar{\mathbf{c}})\}$$

where **G** denotes the set of primitive functions of **D**. Since **D** is an effective domain, Φ_G is a recursive set.

(ii) Φ_{if} is the set of productions

$$\{\text{if } t \text{ then } u \text{ else } v \to u \mid t \in \text{can}(D_{\text{true}});\ u, v \text{ are variable-free terms in } L_P\} \cup$$
$$\{\text{if } t \text{ then } u \text{ else } v \to v \mid t \in \text{can}(D_{\text{false}});\ u, v \text{ are variable-free terms in } L_P\} \cup$$
$$\{\text{if } t \text{ then } u \text{ else } v \to \perp \mid t \in \text{can}(D_{\perp});\ u, v \text{ are variable-free terms in } L_P\}$$

where can (S) stands for $\{\gamma_d \mid \gamma_d$ is the canonical name for an element $d \in S\}$. Φ_{if}

specifies how to reduce conditional expressions where the first argument (the Boolean test) is in canonical form. It is clearly recursive.

(iii) Φ_F is the set of productions

$$\{f_i(\bar{u}) \to t_i(\bar{u}) \mid f_i(\bar{x}) = t_i(\bar{x}) \in P;\ \bar{u} \text{ is a } \#f_i\text{-tuple of variable-free terms in } L_P\}$$

specifying how to expand an arbitrary function application.

LEMMA. *Φ_P is effective.*

Proof. Immediate from the definition of Φ_P and the fact that **D** is effective. The proofs of conditions (i) and (ii) are trivial. Condition (iii) is a routine induction on the structure of terms. □

DEFINITION. The *call-by-name computation* (with respect to the program P) for a variable-free term t in L_P is the Φ_P-reduction of t.

The following theorem establishes that call-by-name computation transforms variable-free terms in L_P into their meanings in the structure $\mathbf{D} \cup \mathbf{F}$.

THEOREM. *Let **D** be an effective, arithmetic domain, and let P be an arbitrary recursive program over **D** defining function symbols F. For every variable-free term t in L_P, the result of the call-by-name computation for t with respect to P is identical to $\mathbf{D} \cup \mathbf{F}[t]$ where **F** denotes the least fixed-point of the functional **P** for P.*

Proof. See reference [9]. It is a straightforward but tedious induction on an appropriate measure of the complexity of the term t. □

6.2. Call-by-value computation. Up to this point, we have confined our attention to program semantics consistent with call-by-name computation. However, most practical programming languages (e.g., LISP, PASCAL, C) employ *call-by-value* computation which has slightly different semantics. Call-by-value computation is identical to call-by-name computation, except for the productions concerning the expansion of program functions F.

DEFINITION. Let L_P be the first order language corresponding to an effective, arithmetic domain **D** augmented by a recursive program P defining the function symbols $F = \{f_1, \cdots, f_n\}$. The *call-by-value production set* $\Phi_{P_\perp}$ for P is the set $\Phi_G \cup \Phi_{F_\perp} \cup \Phi_{if}$ where Φ_G and Φ_{if} are defined exactly as they are in call-by-name computation, and $\Phi_{F_\perp}$ is defined as the set of productions

$$\{f_i(\bar{c}) \to t_i(\bar{c}) \mid f_i(\bar{x}) = t_i(\bar{x}) \in P;\ \bar{c} \text{ is a } \#f_i\text{-tuple of canonical terms excluding can } (\perp)\}.$$

LEMMA. *$\Phi_{P_\perp}$ is effective.*

Proof. Immediate from the definition of effective production set. □

DEFINITION. The *call-by-value computation* (with respect to program P) for a variable-free term t in L_P is the $\Phi_{P_\perp}$-reduction of t.

The main consequence of this change is that an application of a program function f_i is not expanded until all the arguments are reduced to canonical form. Fortunately, there is a simple semantic relationship between call-by-value and call-by-name computations. In fact, it is trivial to transform a program P into a slightly different program $P_\perp$—called the *strict transform* of P—such that the call-by-name meaning of $P_\perp$ is identical to the call-by-value meaning of P.

DEFINITION. Let P be an arbitrary recursive program

$$\{f_1(\bar{x}_1) = t_1, \cdots, f_n(\bar{x}_n) = t_n\}$$

over an arithmetic domain **D**. The *strict transform* $P_\perp$ corresponding to P is the program

$$\{f_1(\bar{x}_1) = \text{if } \delta(\bar{x}_1) \text{ then } t_1 \text{ else } \perp, \cdots, f_n(\bar{x}_n) = \text{if } \delta(\bar{x}_n) \text{ then } t_n \text{ else } \perp\}$$

where δ is the primitive "is-defined" function[5] such that:

$$\delta(\bar{x}_i)=\begin{cases}\textbf{true} & \text{if } \perp \notin \bar{x}_i,\\ \perp & \text{otherwise.}\end{cases}$$

Given this transformation, the *call-by-value functional* $\mathbf{P}_\perp$ for P is simply the (call-by-name) functional for $P_\perp$:

$$\lambda f_1, \cdots, f_n \cdot [\lambda \bar{x}_1 \cdot \text{if } \delta(\bar{x}_1) \text{ then } t_1 \text{ else } \perp, \cdots, \lambda \bar{x}_n \cdot \text{if } \delta(\bar{x}_n) \text{ then } t_n \text{ else } \perp].$$

The following theorem establishes that call-by-value computation transforms variable-free terms t in L_P into meanings in the expansion of $\mathbf{D}$ determined by the functional $\mathbf{P}_\perp$.

THEOREM. *Let* $\mathbf{D}$ *be an effective, arithmetic domain, and let* P *be an arbitrary recursive program over* $\mathbf{D}$ *defining the function symbols* F. *For every variable-free term* t *in* L_P, *the result of the call-by-value computation for* t *is identical to* $\mathbf{D} \cup \mathbf{F}[t]$ *where* $\mathbf{F}$ *denotes the least fixed-point of the call-by-value functional* $\mathbf{P}_\perp$ *for* P.

Proof. A proof of this theorem, an induction on the complexity of t, appears in [9]; proofs of similar theorems appear in [4] and [26]. □

To avoid confusion between the call-by-name and call-by-value interpretations for recursive programs, we will use the following terminology. Unless we specifically use the qualifier "call-by-value", the intended meaning of a program P defining the function symbols F is the least fixed-point of the (call-by-name) functional $\mathbf{P}$ for P—the call-by-name interpretation for F. In contrast, the intended meaning of a *call-by-value* program P is the least fixed-point of the *call-by-value* functional $\mathbf{P}_\perp$ for P—the call-by-value interpretation for F.

6.3. Proving properties of call-by-value programs. Since it is trivial to translate call-by-value recursive programs into equivalent (call-by-name) recursive programs, first order programming logic obviously accommodates call-by-value semantics. Given a first order axiomatization A_D for the domain $\mathbf{D}$ and a call-by-value recursive program P over $\mathbf{D}$, we augment A_D by the recursion equations $P_\perp$ and the definition of the function δ

$$[x \neq \perp \supset \delta(x) = \text{true}] \wedge \delta(\perp) = \perp.$$

A logically equivalent but conceptually simpler approach is to directly augment A_D by a set of axioms P_{ax} characterizing P; it eliminates the function δ and the construction of the strict transform $P_\perp$. In the direct approach, each function definition

$$f_i(\bar{x}_i) = t_i$$

in the original program P, generates two axioms

$$x_i \neq \perp \wedge \cdots \wedge x_{\#f_i} \neq \perp \supset f_i(\bar{x}_i) = t_i,$$
$$x_i = \perp \vee \cdots \vee x_{\#f_i} = \perp \supset f_i(\bar{x}_i) = \perp$$

defining f in P_{ax}. Note that P_{ax} and $P_\perp$ are logically equivalent sets of formulas. A proof that the expanded domain $\mathbf{D} \cup \mathbf{F}$, where $\mathbf{F}$ denotes the least fixed-point of the call-by-value functional $\mathbf{P}_\perp$, is a model for the augmented axiomatization $A_D \cup P_{\text{ax}}$ appears in reference [7].

Call-by-value programs are an attractive alternative to call-by-name programs because they are easier to implement and programmers seem more comfortable with

[5] Technically, $\boldsymbol{\delta}$ is a countable family of functions $\boldsymbol{\delta}_m$, $m = 1, 2, \cdots$ where $\boldsymbol{\delta}_m$ is the m-ary version of the "is-defined" function. It should be obvious from context which instance of $\boldsymbol{\delta}$ is required.

their semantics. In addition, we will show in § 8 and Appendices III and IV that the complete recursive program corresponding to an arbitrary call-by-value program is easier to describe and understand than the equivalent construction for a call-by-name program.

7. Metamathematics of first order programming logic. Although the fundamental theorem of first order programming logic clearly establishes that recursive programs are arithmetic definitions over an arithmetic domain **D**, it ignores two important issues. First, can recursive programs be ambiguous (as definitions over the domain **D**)? Second, do recursive programs have a plausible interpretation in nonstandard models (of a first order theory for **D**)?

The answer to the first question is significant. In fact, it motivates one of the major technical results of this paper: the complete recursive program construction. Assume that we are given a recursive program P over the domain **D** defining the function symbols $F=\{f_1, \cdots, f_n\}$. If we augment **D** by any fixed-point **F** of the functional **P** for P, the expanded structure $\mathbf{D} \cup \mathbf{F}$ is a model for $P \cup (**)$. Interpreting P as an arithmetic definition for **F** over **D** captures the fact that **F** is a fixed-point of **P**, but not the fact that it is the *least* fixed-point.

What are the implications of this form of incompleteness? If every function in the least fixed-point of the functional for P is total, the problem does not arise because the least fixed-point is the only fixed-point. On the other hand, if some function in the least fixed-point is partial, there may or may not be additional fixed-points. In the former case, we cannot prove any property of the least fixed-point that does not hold for all fixed-points. For example, we cannot prove anything interesting about the function **f** defined by

$$(5) \qquad f(x)=f(x)$$

since any interpretation for f over the domain satisfies (5), not just the everywhere undefined function.

In contrast, the program

$$(6) \qquad f(x)=f(x)+1,$$

which determines exactly the same function **f**, is unambiguous. Consequently, given program (6), we can easily prove that

$$\forall x\, f(x)=\perp$$

in first order programming logic.

There are several possible solutions to this problem. John McCarthy [18] has suggested adding a "minimization" axiom scheme φ_P (containing a free function parameter for each function symbol) to the definition of a program P. The scheme φ_P asserts that **F** approximates every definable set of functions $\mathbf{F}'$ satisfying the equations P ($P_\perp$ if P is a call-by-value program). In this paper, we will develop a more direct approach to the problem: a method for mechanically translating an arbitrary recursive program into an equivalent recursive program with a unique fixed-point.

DEFINITION. A (call-by-name or call-by-value) recursive program P over the domain **D** is *complete* iff the corresponding functional has a unique fixed-point.

In the next section of this paper, we will prove that every recursive program can be effectively transformed into an equivalent complete recursive program. As a result, we can reason about recursive programs that define partial functions by first transforming the programs into equivalent complete programs. Fortunately, the transformation

process leaves the logical structure of the original program intact—so that the programmer can understand it.

The answer to the second metamathematical question raised at the beginning of this section is even more interesting than the first one, although its practical significance is less apparent. The behavior of recursive programs in nonstandard structures does not concern programmers interested in proving properties of recursive programs over the standard arithmetic domains supported by language processors. Regardless of the meaning of recursive programs in nonstandard models, first order programming logic provides a sound, yet intuitively appealing formal system for deducing the properties of recursive programs. On the other hand, as computer scientists interested in the deductive and expressive power of various logics, we can gain insight into the relative strength of first order programming logic by settling the question of how to interpret recursive programs over nonstandard structures.

Before we can make precise statements about nonstandard models, we need to introduce some additional terminology.

DEFINITION. Assume that we are given an arithmetic domain **D** generated by the finite set of function symbols Gen. A set T sentences in the language L_D compatible with **D** is an *arithmetically complete* axiomatization of **D** iff

(i) **D** is a model of T.

(ii) T logically implies all the sentences expressing the arithmetic properties of **D** (listed in the definition of arithmetic domain in § 4) except the structural induction principle (*) (which cannot be expressed within a first order theory).

(iii) T logically implies that the structural induction axiom scheme (**) for **D** (stated in § 4), holds for every formula $\varphi(x)$ in L_D.

(iv) For every pair of variable-free terms u and v in L_D, either the sentence $u = v$ or the sentence $u \neq v$ is derivable from T.[6]

Remark. The axiomatization of $\mathbf{N}^+$ in Appendix I is arithmetically complete. Given an arithmetic domain **D**, it is a straightforward but tedious (and error-prone) exercise to devise an effective, arithmetically complete axiomatization for **D**. Note that *Th* **D** is an arithmetically complete axiomatization for **D**. Unfortunately, Gödel's incompleteness theorem implies that for nontrivial domains **D**, *Th* **D** is not recursively enumerable.

An arithmetically complete axiomatization T for an arithmetic domain **D** has many distinct (nonisomorphic) models. The nonstandard models (models other than **D**) are not necessarily arithmetic, since induction may fail for unary predicates that are not definable. On the other hand, they are inductive, because they satisfy the structural induction scheme (**) for **D**.

DEFINITION. A structure $\mathbf{D}'$ is *weakly arithmetic* iff it is a model of an arithmetically complete set of sentences T.

The only difference between an arithmetic and a weakly arithmetic model of T is that induction may fail in a weakly arithmetic model for predicates that are not definable in T. Obviously, the nonstandard models corresponding to an arithmetic domain **D** are weakly arithmetic. Given a recursive program P defining the function symbols F over the arithmetic domain **D**, we can interpret P as a definition over an arbitrary nonstandard model $\mathbf{D}'$ if we can find an interpretation $\mathbf{F}'$ for F such that $\mathbf{D}' \cup \mathbf{F}'$ is a model for $P \cup$ (**).

[6] Although none of the theorems that we prove in this paper depend on this property, it ensures that an arithmetically complete axiomatization has a unique (up to isomorphism) arithmetic model. In addition, it guarantees that arithmetically complete axiomatizations for nontrivial arithmetic domains support elementary syntax (see below).

At first glance, the proof of the fundamental theorem appears to generalize without modification to nonstandard models, because it does not explicitly rely on the fact that the domain $\mathbf{D}$ is arithmetic rather than weakly arithmetic. Let P be a recursive program over an arithmetic domain $\mathbf{D}$ defining the function symbols F, let $\mathbf{D}'$ be a nonstandard model corresponding to $\mathbf{D}$, and let $\mathbf{P}'$ denote the functional for P over $\mathbf{D}'$. Then the least fixed-point $\mathbf{F}'$ of $\mathbf{P}'$ obviously satisfies the equations P, implying that $\mathbf{D}' \cup \mathbf{F}'$ is a model for P. Hence, the only remaining step in confirming that the proof generalizes to nonstandard models is to show that the induction scheme (**) holds for definable predicates in the extended language $L_D \cup F$—a property that appears plausible, if not obvious.

Nevertheless, there is a simple counterexample to this conjecture. Consider the following program defined over a nonstandard model $\bar{\mathbf{N}}$[7] of the natural numbers augmented by $\perp$ (axiomatized as in Appendix I):

(7) $\quad \text{zero}(n) = \text{if } n \text{ equal } 0 \text{ then } 0 \text{ else zero}(n-1).$

This program is essentially identical to the one that Hitchcock and Park [16] used to argue that first order logic was incapable of expressing and proving that the function defined by (7) is total. The least fixed-point of the corresponding functional is the function **zero** defined by:

$$\mathbf{zero}(x) = \begin{cases} \mathbf{0} & \text{if } x \text{ is a standard natural number,} \\ \perp & \text{otherwise } (x \text{ is } \perp \text{ or nonstandard}). \end{cases}$$

Yet, we have already established the fact in refuting Hitchcock and Park's argument (§ 3 and Appendix II) that we can prove (using structural induction) that the zero function defined in equation (7) is identically zero everywhere except at $\perp$.

Clearly, our naive approach to generalizing the proof of the fundamental theorem to nonstandard models will not work. Our assumption that a recursive program P over a weakly arithmetic structure $\mathbf{D}'$ can be interpreted as a definition introducing a set of functions $\mathbf{F}'$ that

(i) forms the least fixed-point of the functional for P, and
(ii) obeys the structural induction principle (**)

leads to a contradiction. Where did we go wrong?

Ironically, we made essentially the same mistake as Hitchcock and Park: we assumed that a recursive program over a nonstandard structure should be interpreted as the least fixed-point of the corresponding functional. In the preceding example, the least fixed-point of the functional for equation (7) over $\bar{\mathbf{N}}$ is not definable in $\bar{\mathbf{N}}$. For this reason, it need not obey the structural induction principle.

In order to generalize the fundamental theorem to nonstandard models, we must develop a more sophisticated interpretation for recursive programs than the least fixed-point of the corresponding functional. Since first order programming logic formalizes recursive programs as arithmetic definitions over the data domain, we must find an interpretation for recursive programs over nonstandard models that satisfies the structural induction principle. From the preceding example, it is obvious that the least fixed-point interpretation does not. What is a reasonable alternative? For induction to hold, the interpretation must be definable in the original domain $\mathbf{D}$. Hence, we must limit our attention to definable fixed-points of the functional for a recursive program—abandoning our reliance on the familiar least fixed-point construction from Kleene's recursion theorem. In its place, we must develop a new approach to constructing fixed-points of functionals that always determines definable functions.

[7] A model containing an element with infinitely many predecessors.

A detailed, systematic development of the subject of definable fixed-points is beyond the scope of this paper (the interested reader is encouraged to consult reference [10]). However, the correct formulation of the generalized fundamental theorem rests on a single lemma which is easy to explain and to justify. The critical lemma is a generalization of Kleene's recursion theorem; it asserts that every continuous functional over an appropriate domain **D** has a least definable-fixed-point. The lemma applies to weakly arithmetic domains that support what John McCarthy calls *elementary syntax.* Fortunately, nonstandard models of nontrivial, arithmetically complete theories possess this property.

Elementary syntax is a definition that introduces functions for encoding finite sequences over the universe as individual elements of the universe. We formalize the notion as follows.

DEFINITION. An arithmetically complete axiomatization T *supports elementary syntax* iff there exists an unambiguous definition Elem augmenting T introducing a set of functions and predicates **Seq** including the constant **0**; unary functions **last**, **mkseq**, **length**, and **suc**; the binary function $\circ$ (append); and unary predicates **seq** and **nat** satisfying the following sentences:

(a) $\mathrm{suc}(\bot)=\bot$
(b) $\forall x[x:\mathrm{nat}\,(x=0\oplus\exists! y[x=\mathrm{suc}\,(y)\wedge y:\mathrm{nat}])]$
(c) $\varphi(0)\wedge\forall x:\mathrm{nat}\,[\varphi(x)\supset\varphi(\mathrm{suc}\,(x))]\supset\forall x:\mathrm{nat}\,\varphi(x)$
for every formula $\varphi(x)$ in $L_D\cup\mathrm{Seq}$
(d) $\mathrm{mkseq}\,(\bot)=\bot$
(e) $\forall x[x:\mathrm{seq}\equiv\exists! d(d\neq\bot\wedge[\mathrm{mkseq}\,(d)=x\oplus\exists! y:\mathrm{seq}\;x=\mathrm{mkseq}\,(d)\circ y])]$
(f) $\forall x,y,z:\mathrm{seq}\,[x\circ(y\circ z)=(x\circ y)\circ z]$
(g) $\forall x,y[x=\bot\vee y=\bot\supset x\circ y=\bot]$
(h) $\forall d[\mathrm{last}\,(\mathrm{mkseq}\,(d))=d]$
(i) $\forall y:\mathrm{seq}\,\forall d[d\neq\bot\supset\mathrm{last}\,(\mathrm{mkseq}\,(d)\circ y)=\mathrm{last}\,(y)]$
(j) $\forall d[d\neq\bot\supset\mathrm{length}\,(\mathrm{mkseq}\,(d))=\mathrm{suc}\,(0)]$
(k) $\forall y:\mathrm{seq}\,\forall d[d\neq\bot\supset\mathrm{length}\,(\mathrm{mkseq}\,(d)\circ y)=\mathrm{suc}\,(\mathrm{length}\,(y))]$
(l) $\forall y[\varphi(\mathrm{mkseq}\,(y))]\wedge\forall x:\mathrm{seq}\,[\varphi(x)\supset\forall y\,\varphi(\mathrm{mkseq}\,(y)\circ x)]\supset\forall x:\mathrm{seq}\,\varphi(x)$
for every formula $\varphi(x)$ in $L_D\cup\mathrm{Seq}$.

Remark. Elem implicitly determines a representation function embedding the finite, nonempty sequences over $|D|$ in $|D|$. In the sequel, we will denote the element of $|D|$ representing the sequence $[s_1,\cdots,s_k]$ by $\langle s_1,\cdots,s_k\rangle$.

DEFINITION. A weakly arithmetic domain $\mathbf{D}'$ *supports elementary syntax* iff there exists a corresponding arithmetically complete axiomatization A_D that supports elementary syntax.

Given this terminology, we can succinctly state the lemma as follows.

LEMMA (generalized recursion theorem). *For every recursive program P over a weakly arithmetic domain $\mathbf{D}'$ supporting elementary syntax, the corresponding functional $\mathbf{P}'$ has a least definable-fixed-point.*

Proof. A detailed proof of this lemma appears [10]; we will only sketch the main ideas. Let **D** and T be the arithmetic domain and the arithmetically complete axiomatization, respectively, corresponding to $\mathbf{D}'$. By Kleene's least fixed-point theorem for continuous functionals, the functional **P** for P in **D** has a least fixed-point $[\mathbf{f_1},\cdots,\mathbf{f_n}]$. It is a straightforward, but tedious exercise to construct formulas (called *program formulas*) $\psi_1(\bar{x}_1,y),\cdots,\psi_n(\bar{x}_n,y)$ in $L_D\cup\mathrm{Seq}$ such that $\mathbf{D}[\psi_i(\bar{x}_1,y)][s]$ is **TRUE** iff $\mathbf{f_i}(s(\bar{x}_i))=s(y)$. The key idea underlying the construction of the program formulas is that for each pair $(\bar{a},b)$ in the graph of a program defined function $\mathbf{f_i}$, there exists a set of finite graphs $G_1\subset\mathrm{Graph}\,(\mathbf{f_1}),\cdots,G_n\subset\mathrm{Graph}\,(\mathbf{f_n})$, such that:

(i) $(\bar{a}, b) \in G_i$.

(ii) The collection of graphs $G_1, \cdots, G_n$ is *computationally closed under P*: for each pair $(\bar{u}, v)$ in a graph G_j, every reduction of a function application $f_k(\bar{t})$ in the (call-by-name) computation reducing $f_j(\bar{u})$ to v (according to the definition presented in § 6) has the property that $(\bar{\mathbf{t}}, \mathbf{f_k}(\bar{\mathbf{t}}))$ appears in G_k.

Since a finite graph $\{(\bar{a}_1, b_1), \cdots, (\bar{a}_m, b_m)\}$ can be represented by the sequence $\langle\langle\bar{a}_1\rangle, b_1, \cdots, \langle\bar{a}_m\rangle, b_m\rangle$, the formula $\psi_i(\bar{x}_i, y)$ can be expressed in the form

$$\exists g_1, \cdots, g_n(\mathrm{Clos}_P(g_1, \cdots, g_n) \wedge \mathrm{member}(\langle x_i\rangle, y, g_i))$$

where each g_j is a sequence $\langle\langle\bar{a}\rangle, b_1, \cdots, \langle\bar{a}_m\rangle, b_m\rangle$, $\mathrm{Clos}_P(g_1, \cdots, g_n)$ is a formula expressing the fact that $g_1, \cdots, g_n$ represent a set of finite graphs that are computationally closed under P, and member $(\langle\bar{x}\rangle, y, g)$ is a formula expressing the fact that $\langle\langle\bar{x}\rangle, y\rangle$ is an element of the graph represented by the sequence g. Roughly speaking, the formula Clos_P is generated from the text of P by replacing all references to program function symbols f_j by corresponding references to finite graphs g_j.

In the standard model $\mathbf{D}$, the formulas $\psi_1(\bar{x}_1, y), \cdots, \psi_n(\bar{x}_n, y)$ characterize the least fixed-point of $\mathbf{P}$. What do they mean in the weakly arithmetic model $\mathbf{D}'$? From T, we can prove that the n-tuple of functions $[\mathbf{f}'_1, \cdots, \mathbf{f}'_n]$ determined by $\psi_1(\bar{x}_n, y), \cdots, \psi_n(\bar{x}_n, y)$ satisfies the definition P. Moreover, given another collection of formulas $\varphi_1(\bar{x}_1, y), \cdots, \varphi_n(\bar{x}_n, y)$ determining an n-tuple of functions $[\tilde{\mathbf{f}}'_1, \cdots, \tilde{\mathbf{f}}'_n]$ over $|D'|$ that satisfies the definition P, we can prove that $[\mathbf{f}'_1, \cdots, \mathbf{f}'_n]$ approximates $[\tilde{\mathbf{f}}'_1, \cdots, \tilde{\mathbf{f}}'_n]$. Hence, the n-tuple of functions determined by $\psi_1(\bar{x}_1, y), \cdots, \psi_n(\bar{x}_n, y)$ must be the least definable-fixed-point of $\mathbf{P}'$. □

Given the generalized recursion theorem, the following generalization of the fundamental theorem is a simple consequence.

THEOREM (generalized fundamental theorem). *Let P be a recursive program*

$$\{f_1(\bar{x}_1) = t_1, f_2(\bar{x}_2) = t_2, \cdots, f_n(\bar{x}_n) = t_n\}$$

over a weakly arithmetic domain $\mathbf{D}'$ *supporting elementary syntax, and let* $\mathbf{F}'$ *denote the least definable fixed-point of the functional for P. Then P is an arithmetic definition over* $\mathbf{D}'$ *satisfying the model* $\mathbf{D}' \cup \mathbf{F}'$.

Proof. By the generalized recursion theorem, the functional $\mathbf{P}'$ over $\mathbf{D}'$ for P has a least definable fixed-point $[\mathbf{f}'_1, \cdots, \mathbf{f}'_n]$. Hence, the relationship

$$\bigwedge_{1 \le i \le n} \mathbf{f_1} = \lambda \bar{x}_i \cdot \mathbf{t_i}$$

holds where $\mathbf{t_i}$ denotes the interpretation of t_i given that the primitive function symbols in t_i are interpreted by the corresponding functions in $\mathbf{D}'$ and the function symbols $f_1, \cdots, f_n$ are interpreted by $\mathbf{f}'_1, \cdots, \mathbf{f}'_n$, respectively. We can restate this fact as follows

$$\bigwedge_{1 \le i \le n} \mathbf{D}'\dagger[f_i(\bar{x}_i)][s] = \mathbf{D}'\dagger[t_i][s]$$

where $\mathbf{D}'\dagger$ denotes the structure $\mathbf{D}' \cup \mathbf{F}'$ and s is an arbitrary state over $|D'\dagger|$. By the same construction used in the proof of the generalized recursion theorem, every formula over $\mathbf{D}'\dagger$ can be translated into an equivalent formula over $\mathbf{D}'$. Consequently, the induction principle (**) holds for all formulas over $\mathbf{D}'$, implying $\mathbf{D}'\dagger$ is a model for $P \cup (**)$ extending $\mathbf{D}'$. □

8. Construction of complete recursive programs. In this section, we will show how to construct a complete recursive program P^* equivalent to a given call-by-value program P. We will also verify that the constructed program P^* actually is complete

and equivalent to P. We relegate the analogous construction and proof for call-by-name programs to Appendix III, since they are similar but somewhat more complex.

The intuitive idea underlying the construction is to define for each function f in the original call-by-value program a corresponding function f^* such that $f^*(x_1, \cdots, x_n)$ constructs the computation sequence for the call-by-value evaluation of $f(x_1, \cdots, x_n)$. In fact, constructing the actual computation sequence really is not necessary; the values of the elements in the sequence, except for the final one (the value of $f(x_1, \cdots, x_n)$), are irrelevant. It is the expanding structure of the sequence that is significant, because it prevents an arbitrary fixed-point solution from filling in points where the computed (least) fixed-point diverges.

For example, consider the trivial program

(8) $\quad f(x) = \text{if } x \text{ equal } 0 \text{ then } 0 \text{ else } f(g(x))$

over the domain of LISP S-expressions where $\mathbf{g}$ is any unary function with fixed-points (i.e., $\mathbf{g}(y) = y$ for some S-expression y). The corresponding functional obviously has multiple fixed-points, although the intended meaning of the definition is the least fixed-point $\mathbf{f}$. If we define f^* by the program

(9) $\quad f^*(x) = \text{if } x \text{ equal } 0 \text{ then cons}\,(0, \text{NIL}) \text{ else cons}\,(g(x), f^*(g(x))),$

then f^* constructs a sequence containing the argument for each call on f in the call-by-value evaluation of $f(x)$, assuming that $f(x)$ terminates. If $\mathbf{f}(x)$ does not terminate (e.g., $\mathbf{g}(x) = x$), then every fixed-point $\mathbf{f}^*$ of the functional for definition (9) must be undefined ($\perp$) at x. Otherwise, $\mathbf{f}^*(x)$ has length greater than any integer which contradicts the fact that every sequence in the data domain is finite.[8] Given (9), we can redefine f by the recursion equation

(10) $\quad f(x) = \text{last}\,(f^*(x))$

where **last** is the standard LISP function that extracts the final element in a list. Now, by substituting the definition consisting of equations (9) and (10) for the original program (8), we can force f to mean $\mathbf{f}$. We generalize this idea to arbitrary recursive programs as follows.

DEFINITION. Let P be a call-by-value recursive program defining the function symbols F over the arithmetic domain $\mathbf{D}$ supporting elementary syntax. Let L_P and L_{D^*} denote the first order languages $L_D \cup F$ and $L_D \cup \text{Seq}$, respectively, and let $\mathbf{D}^*$ denote domain $\mathbf{D} \cup \mathbf{Seq}$. Let t be an arbitrary term in the language L_P. The *call-by-value computation sequence term* t^* (in the extended language $L_{P^*} = L_{D^*} \cup \{f_1^*, \cdots, f_n^*\}$) corresponding to t is inductively defined as follows:

(i) If t is a constant or a variable x,

$$t^* = \text{mkseq}\,(x).$$

(ii) If t has the form $g(u_1, \cdots, u_{\#g})$ where $g \in G - \{\text{if-then-else}\}$,

$$t^* = u_1^* \circ \cdots \circ u_{\#g}^* \circ \text{mkseq}\,(g(\text{last}\,(u_1^*), \cdots, \text{last}\,(u_{\#g}^*))).$$

(iii) If t has the form $f_i(u_1, \cdots, u_{\#f_i})$,

$$t^* = u_1^* \circ \cdots \circ u_{\#f_i}^* \circ f_1^*\,(\text{last}\,(u_1^*), \cdots, \text{last}\,(u_{\#f_i}^*)).$$

---

[8] Although this argument is cast in terms of standard S-expressions, it generalizes to arbitrary models of a simple first order theory of S-expressions. Given the usual recursive definition for **length**, the sentence $\forall y{:}\,\text{list}\ \exists n{:}\,\text{integer}\,[\text{length}\,(y) < n]$ is a theorem of the theory. Hence it must hold for arbitrary models—including those with infinite objects.

(iv) If t has the form if u_0 then u_1 else u_2,

$$t^* = u_0^* \circ (\text{if last}\,(u_0^*) \text{ then } u_1^* \text{ else } u_2^*).$$

The *call-by-value complete recursive program* P^* equivalent to P is the call-by-value program

$$\{f_1^*(\bar{x}_1) = t_1^*, \cdots, f_n^*(\bar{x}_n) = t_n^*\}$$

over $\mathbf{D}^*$.

A similar construction generates the complete recursive program equivalent to an arbitrary call-by-name recursive program; it appears in Appendix IV. The following theorem, called the *fixed-point normalization theorem*, shows that the complete recursive program construction preserves the meaning of the original program and produces a program that is in fact complete.

THEOREM (fixed point normalization theorem). *Let P be a call-by-value recursive program over an arithmetic data domain $\mathbf{D}$ supporting elementary syntax and let $[\mathbf{f}_1, \cdots, \mathbf{f}_n]$ denote the least fixed-point of the call-by-value functional $\mathbf{P}_\perp$ for P. The complete recursive program P^* equivalent to P has the following properties*:

(i) *P^* is complete, i.e. the corresponding call-by-value functional $\mathbf{P}_\perp^*$ has a unique fixed-point $[\mathbf{f}_1^*, \cdots, \mathbf{f}_n^*]$.*

(ii) *For $i = 1, \cdots, n$, $\mathbf{last}(\mathbf{f}_i^*(\bar{d})) = \mathbf{f}_i(\bar{d}))$ for all $\# f_i$-tuples $\bar{d}$ over $|D|$.*

Proof. See Appendix III. □

The corresponding theorem for call-by-name programs and a sketch of its proof appear in Appendix IV.

The fixed-point normalization theorem has an important corollary relating complete recursive programs to first order theories. In informal terms, the corollary asserts that a complete recursive program over an arithmetic domain $\mathbf{D}$ is an unambiguous, arithmetic definition augmenting a suitable first order theory for $\mathbf{D}$ (a theory closely resembling Peano arithmetic[9]).

COROLLARY. *Let A_D be an arithmetically complete first order axiomatization for the arithmetic domain $\mathbf{D}$. Then for every call-by-value program P over $\mathbf{D}$, the equivalent complete recursive program P^* (expressed in the form P^*_{ax}) is an unambiguous, arithmetic definition augmenting $A_D \cup$ Elem.*

Proof. The key idea underlying the proof of the corollary is to generalize the fixed-point normalization theorem to cover programs defined over weakly arithmetic (not just arithmetic) domains. Since all the models of $A_D \cup$ Elem are weakly arithmetic, the generalized normalization theorem implies that P^*_{ax} determines a unique expansion of every model of $A_D \cup$ Elem—immediately establishing the corollary. □

The only obstacle to extending the fixed-point normalization theorem to weakly arithmetic domains is the same complication that we encountered in generalizing the fundamental theorem of first order programming logic in § 7: the least fixed-point of the functional corresponding to a recursive program may not be definable. We overcame this problem in § 7 by substituting the notion of least definable fixed-point for the standard notion of least fixed-point. The same strategy works here.

THEOREM (generalized fixed point normalization theorem). *Let P be a call-by-value recursive program over a weakly arithmetic data domain $\mathbf{D}$ supporting elementary syntax and let $[\mathbf{f}_1, \cdots, \mathbf{f}_n]$ denote the least definable fixed-point of the call-by-value functional $\mathbf{P}_\perp$ for P. The call-by-value complete recursive program P^* equivalent to P has the following properties*:

[9] The first order theory generated by Peano's axioms for the natural numbers.

(i) *P^* is complete, i.e. the corresponding call-by-value functional $\mathbf{P}^*_\perp$ has a unique definable-fixed-point $[\mathbf{f}_1^*, \cdots, \mathbf{f}_n^*]$.*

(ii) *For $i=1, \cdots, n$, $\mathbf{last}\,(\mathbf{f}_i^*(\bar{d}))=\mathbf{f}_i(\bar{d}))$ for all $\# f_i$-tuples $\bar{d}$ over $|D|$.*

Proof. The proof of the generalized fixed-point normalization theorem is essentially identical to the proof of the original one, except that it must invoke the generalized recursion theorem described in § 7 instead of Kleene's recursion theorem—substituting the notion of least definable-fixed-point for least fixed-point. □

We will use the term *extended first order programming logic* for P to refer to conventional first order programming logic for P augmented by Elem (the definition of functions **Seq**), the axioms defining the equivalent complete program P^*, and the axioms asserting that each function $\mathbf{f_i}$ is identical to $\mathbf{last} \circ \mathbf{f}_i^*$.

9. Simplifying complete recursive programs. If we carefully examine the proof of the fixed-point normalization theorem, it is clear that we can simplify the structure of the constructed program without affecting the proof of the theorem. It is easy to verify that the same proof works if we substitute the following definition of computation sequence term for the original one.

DEFINITION. Let t be arbitrary term in the language L_D. The *simplified computation sequence term* t^* corresponding to t is given by the inductive definition:

(i) If t is a constant or a variable x,

$$t^* = \text{mkseq}\,(x).$$

(ii) If t has the form $g(u_1, \cdots, u_{\# g})$ where $g \in G - \{\text{if-then-else}\}$,

$$t^* = u^*_{i_1} \circ \cdots \circ u'_{i_k} \circ \text{mkseq}\,(g(\text{last}\,(u^*_1), \cdots, \text{last}\,(u^*_{\# g}))),$$

where $u_{i_1}, \cdots, u_{i_k}$ are all the arguments containing invocations of some program function f_i.

(iii) If t has the form $f_i(u_1, \cdots, u_{\# f_i})$,

$$t^* = u^*_{i_1} \circ \cdots \circ u^*_{i_k} \circ f^*_i\,(\text{last}\,(u^*_1), \cdots, \text{last}\,(u^*_{\# f_i})),$$

where $u_{i_1}, \cdots, u_{i_k}$ are all the arguments containing invocations of some program function f_i, except when this list is empty. In this case, $k=1$ and $u^*_{i_1}$ is mkseq (true).

(iv) If t has the form if u_0 then u_1 else u_2,

$$t^* = \text{if last}\,(u^*_0) \text{ then } u^*_1 \text{ else } u^*_2,$$

when u^*_0 does not contain invocations of some program function f_i. Otherwise,

$$t^* = u^*_0 \circ (\text{if last}\,(u^*_0) \text{ then } u^*_1 \text{ else } u^*_2).$$

In subsequent examples, we will always use this construction since it produces simpler translations.

10. Sample proofs involving complete recursive programs. To illustrate how complete programs can be used to prove theorems about partial functions, we present two examples.

10.1. Example 1. A divergent function. Let f be the partial function on the natural numbers defined by the recursive program

(11) $\quad f(x)=f(x+1).$

We want to prove the following theorem.

THEOREM. $\forall x[f(x)=\perp]$.

Proof. Although the intended meaning of f (the least fixed-point of the functional for the program (11)) is everywhere undefined, we cannot establish this property using conventional first order programming logic since f is total in many other models. On the other hand, in extended first order programming logic, we can prove the theorem by using the equivalent complete program

$$f^*(x)=\text{mkseq}\,(x+1)\circ f^*(x+1)$$

and the axiom

$$\forall x\, f(x)=\text{last}\,(f^*(x))$$

relating it to the original program (11). Since last is strict, the theorem is an immediate consequence of the following lemma. □

LEMMA. $\forall x[f^*(x)=\perp]$.

Proof. We prove the lemma by structural induction on the (possibly $\perp$) sequence $f^*(x)$.

Basis: $f(x)=\perp$.

In this case, the theorem is true by assumption.

Induction step: $f^*(x)\neq\perp$.

By induction, we may assume that the lemma holds for all x_0 such that $f^*(x_0)$ is a proper tail of $f^*(x)$. Since $x\in N$, $f^*(x)$ expands into the expression mkseq $(x)\circ f^*(x+1)$. Instantiating the induction hypothesis for $x_0=x+1$ yields $f^*(x+1)=\perp$, implying $f^*(x)=\perp$. □

10.2. Example 2. Equivalence of two program schemes. A more interesting example is the proof that the following two iterative program schemes are equivalent.

Program A	Program B
{	{
$x\leftarrow f(x)$;	repeat $x\leftarrow f(x)$ until $p(x)$;
while $\neg p(x)$ do $x\leftarrow f(x)$;	return (x)
return (x)	
}	}

Expressed as recursive programs, program A and program B have the following form:

progA $(x)\leftarrow$ whileA $(f(x))$
whileA $(x)\leftarrow$ if $p(x)$ then x else whileA $(f(x))$

progB $(x)\leftarrow$ repeatB (x)
repeatB $(x)\leftarrow$ if $p(f(x))$ then $f(x)$ else repeatB $(f(x))$.

We wish to prove the following formal theorem.

THEOREM. $\forall x[\text{progA}\,(x)=\text{progB}\,(x)]$.

Proof. By simplification, the theorem trivially reduces to the statement

(12) $\forall x[\text{whileA}\,(f(x))=\text{repeatB}\,(x)]$.

If $f(x)$ is not total, this statement is not provable in conventional first order programming logic, because the recursive definitions of whileA and repeatB may have extraneous fixed-points. However, in extended first order programming logic, the proof is straightforward. Given the equivalent complete programs

whileA* $(x)\leftarrow$ if $p(x)$ then mkseq (x) else cons $(x, \text{whileA*}\,(f(x)))$
repeatB* $(x)\leftarrow$ if $p(f(x))$ then mkseq $(f(x))$ else cons $(x, \text{repeatB*}\,(f(x)))$

and the axioms (from extended first order programming logic)

$$\begin{aligned}&\text{cons}\,(x, y) = \text{mkseq}\,(x) \circ y\\&\text{last}\,(\text{mkseq}\,(x)) = x\\&x \neq \bot \supset \text{last}\,(\text{cons}\,(x, y)) = \text{last}\,(y)\\&\text{whileA}\,(x) = \text{last}\,(\text{whileA}^*\,(x))\\&\text{repeatB}\,(x) = \text{last}\,(\text{repeatB}^*\,(x))\end{aligned}$$

relating the complete program to the original one, statement (12) reduces to the sentence:

(13) $\forall x[\text{last}\,(\text{whileA}^*\,(f(x)) = \text{last}\,(\text{repeatB}^*\,(x))]$.

As in higher order logics based on fixed-point induction (e.g., Edinburgh LCF), the proof of (13) breaks down into two parts:

(13a) $\forall x[\text{last}\,(\text{whileA}^*\,(f(x)) \subseteq \text{last}\,(\text{repeatB}^*\,(x))]$

(13b) $\forall x[\text{last}\,(\text{repeatB}^*\,(x)) \subseteq \text{whileA}^*\,(f(x))]$

where $\alpha \subseteq \beta$ intuitively means "α approximates β" (as defined in § 8) and formally abbreviates the formula

$$\alpha = \bot \vee \alpha = \beta.$$

The proof of (13a) proceeds as follows.

First, we can assume that $\text{last}\,(\text{whileA}^*\,(f(x))) \neq \bot$; otherwise, (13a) trivially holds. Given this assumption and the fact that last is strict (which follows immediately from the definition of last), we can apply structural induction on $\text{whileA}^*\,(f(x))$. As the induction hypothesis we assume that

$$\text{last}\,(\text{whileA}^*\,(f(x'))) \subseteq \text{last}\,(\text{repeatB}^*\,(x'))$$

for all x' such that $\text{whileA}^*\,(f(x'))$ is a proper subtail of $\text{whileA}^*\,(f(x))$. By the definition of whileA*,

$$\text{whileA}^*\,(f(x)) = \text{if } p(f(x)) \text{ then mkseq}\,(f(x)) \text{ else cons}\,(f(x), \text{whileA}^*\,(f(f(x)))).$$

A three-way case analysis on the value of $p(f(x))$ completes the proof of (13a).

Case (a). $p(f(x)) \in D_{\bot}$.
In this case, $\text{whileA}^*\,(f(x)) = \bot$, which is a contradiction.

Case (b). $p(f(x)) \in D_{\text{true}}$.
By simplification,

$$\text{last}\,(\text{whileA}^*\,(f(x))) = f(x) \text{ and last}\,(\text{repeatB}^*\,(x)) = f(x)$$

establishing (13a).

Case (c). $p(f(x)) \in D_{\text{false}}$.
Obviously,

$$\text{whileA}^*\,(f(x)) = \text{cons}\,(f(x), \text{whileA}^*\,(f(f(x))))$$

and

$$\text{repeatB}^*\,(x) = \text{cons}\,(f(x), \text{repeatB}^*\,(f(x)))$$

implying that $\text{whileA}^*\,(f(f(x)))$ is shorter than $\text{whileA}^*\,(f(x))$. Consequently, the induction hypothesis holds for $x' = f(x)$, yielding the following chain of simplifications:

$$\begin{aligned}\text{last}(\text{whileA}^*(f(x))) &= \text{last}(\text{cons}(f(x), \text{whileA}^*(f(f(x)))))\\ &= \text{last}(\text{whileA}^*(f(f(x))))\\ &\subseteq \text{last}(\text{repeatB}^*(f(x))) \quad \text{(by induction)}\\ &= \text{last}(\text{repeatB}^*(x)),\end{aligned}$$

which proves (13a).

Since the proof of (13b) is nearly identical, it is omitted. □

An interesting property of proofs based on complete recursive programs is their similarity to the corresponding proofs based on fixed-point induction in higher order logics. Although fixed-point induction is an awkward rule for reasoning about total functions, it appears well suited to proving many properties of partial functions.

11. Advantages of first order programming logic. Although first order programming logic is narrower in scope than higher order logics (since it does not accommodate functions that take functions as arguments), it is a powerful, yet natural formalism for proving properties of recursive programs. The primary advantage of first order programming logic is that it relies on the familiar principle of structural induction, the most important proof technique in discrete mathematics. In first order programming logic, programmers can develop proofs that are direct formalizations of familiar informal structural induction arguments. In contrast, higher order logics for recursive programs (e.g. Edinburgh LCF [15]) typically rely on fixed-point induction, a rule that is more obscure and difficult to use.

A particularly troublesome aspect of fixed-point induction is that it is valid only for admissible formulas.[10] Edinburgh LCF [15], for example, restricts the application of fixed-point induction to formulas that pass a complex syntactic test ensuring admissibility. Unfortunately, it is often difficult to predict whether a particular formula will pass the test. Moreover, the test does not necessarily produce consistent results on logically equivalent formulas.

As an illustration, consider the sample proof presented in § 5: the termination of the recursive program flat. The proof using first order programming logic is a direct translation of the obvious informal proof. In contrast, a proof of the same theorem in Edinburgh LCF (or similar higher order logic) must introduce a retraction characterizing the domain of S-expressions and simulate structural induction by performing fixed-point induction on the retraction. In the fixed-point induction step, the programmer (or theorem prover) must check that the formula is admissible (by applying the syntactic test) before applying the rule.

For the reasons cited above, we believe that first order programming logic—rather than a higher order logic—is the appropriate formal system for proving properties of recursive programs. Both Boyer and Moore [3], [4] and Cartwright [6], [7] have successfully applied first order programming logic to prove the correctness of sophisticated LISP programs with relative ease. On the other hand, the practical utility of extended first order programming logic for reasoning about non-total functions (such as interpreters) has yet to be determined. Moreover, it is not obvious that the particular complete recursive program construction presented in this paper is the best way to translate an arbitrary program into an equivalent complete program. There are many different equivalence preserving program transformations that generate complete programs. The few examples that we have done manually are encouraging, but we

[10] The formula $\alpha[f]$ in the language $L_D \cup \{f\}$ is *admissible with respect to f over the continuous domain* $\mathbf{D}$ iff for every ascending chain $\mathbf{f}_0 \subseteq \mathbf{f}_1 \subseteq \cdots \subseteq \mathbf{f}_k \subseteq \cdots$ over $\mathbf{D}$, $\mathbf{D} \cup \mathbf{f}[\alpha[f]]$ is TRUE for all functions $\mathbf{f}$ in the chain implies that $\mathbf{D} \cup \mathbf{f}[\alpha[f]]$ is **TRUE** for $\mathbf{f}$ defined as $\mathbf{lub}\{\mathbf{f}_k | k \geqq 0\}$.

cannot reach a firm conclusion until we build a machine implementation and experiment with various schemes for translating arbitrary programs into complete ones.

12. Related and future research. A group of Hungarian logicians—Andreka, Nemeti, and Sain—have independently developed a programming logic [1] with metamathematical properties similar to first order programming logic, although the pragmatic details are completely different. Their logic formalizes flowchart programs as predicate definitions within a first order theory of the data domain excluding $\perp$. Given a flowchart program P, they generate a formula $\pi_P(x, y)$ that is true (in the standard model of the data domain) iff y is the output produced by applying program P to input x. In contrast to first order programming logic, the notion of computation embedded in their logic applies to *all* models of the data domain theory.[11] We are confident that an analogous result holds for first order programming logic; we intend to formulate and prove it in a subsequent paper.

As a formal system for reasoning about recursive programs, the major limitation of first order programming logic as formulated in this paper is that it does not accommodate "higher order" data domains—structures that are not flat. In practice, this restriction may not be very important since higher order objects can always be modeled by intensional descriptions (e.g., computable functions as program text). Nevertheless, we believe that an important direction for future research is to extend first order programming logic to "higher-order" domains. With this objective in mind, we are exploring the implications of allowing lazy (nonstrict) constructors (e.g., lazy cons in LISP) in the data domain.

Appendix I. Sample first order axiomatizations. A conventional first order axiomatization A for the structure $\mathbf{N}$, the natural numbers with functions $\{\mathbf{0}, \mathbf{suc}, +, \times\}$, is:

(1) $\forall x[x=0 \oplus \exists! y\, x=\mathrm{suc}\,(y)]$.
(2) $\forall y[0+y=y]$.
(3) $\forall x, y[\mathrm{suc}\,(x)+y=\mathrm{suc}\,(x+y)]$.
(4) $\forall x, y[0\times y=0]$.
(5) $\forall x, y[\mathrm{suc}\,(x)\times y=y+(x\times y)]$.
(6) $(\alpha(0)\wedge \forall x[\alpha(x)\supset \alpha(\mathrm{suc}\,(x))])\supset \forall x\alpha(x)$ for every formula $\alpha(x)$.

The corresponding axiomatization A^+ for the arithmetic domain $\mathbf{N}^+$, consisting of the universe $|N|\cup\{\perp\}$ and functions $\{\mathbf{0}, \mathbf{true}, \mathbf{false}, \mathbf{suc}, \mathbf{equal}, +, \times, \mathbf{if\text{-}then\text{-}else}\}$, is:

(1) $\forall x:N[x=0\oplus \exists! y:N x=\mathrm{suc}\,(y)]$.
(2) $\forall y:N[0+y=y]$.
(3) $\forall x, y:N[\mathrm{suc}\,(x)+y=\mathrm{suc}\,(x+y)]$.
(4) $\forall x, y:N[0\times y=0]$.
(5) $\forall x, y:N[\mathrm{suc}\,(x)\times y=y+(x\times y)]$.
(6) $\alpha(0)\wedge \forall x:N[\alpha(x)\supset \alpha(\mathrm{suc}\,(x))]\supset \forall x:N\alpha(x)$ for every formula $\alpha(x)$.
(7) $0:N$.
(8) $\forall x:N[\mathrm{suc}\,(x):N]$.
(9) $\mathrm{suc}\,(\perp)=\perp$.
(10) $\forall y[\perp+y=\perp\wedge y+\perp=\perp]$.
(11) $\forall x[x\times\perp=\perp\wedge \perp\times x=\perp]$.
(12) $\mathrm{true}=\mathrm{suc}\,(0)\wedge \mathrm{false}=0$.

[11] Presumably, their notion of nonstandard computation is closely related to the concept of least definable-fixed-points presented in this paper.

(13) $\forall x, y:N[(x=y \supset (x \text{ equal } y)=\text{true}) \wedge (x \neq y \supset (x \text{ equal } y)=\text{false})]$.
(14) $\forall x[(\perp \text{ equal } x)=\perp \wedge (x \text{ equal } \perp)=\perp]$.
(15) $\forall y, z[\text{if false then } y \text{ else } z=z]$.
(16) $\forall x:N\ \forall y, z[\text{if suc}(x) \text{ then } y \text{ else } z=y]$.
(17) $\forall y, z\ [\text{if } \perp \text{ then } y \text{ else } z=\perp]$.

where $x:N$ abbreviates the formula $x \neq \perp$.

Appendix II. Sample proofs in first order programming logic.

Example 1. *Termination of the countdown function.* Let the function zero over the natural numbers **N** augmented by $\{\perp\}$ be defined by the (call-by-name) recursive program:

$$\text{zero}(n)=\text{if } n \text{ equal } 0 \text{ then else zero}(n-1),$$

which is logically equivalent (on $|N|$) to the definition for zero in § 3:

$$\forall n[(n=0 \supset \text{zero}(n)=0) \wedge (n \neq 0 \supset \text{zero}(n)=\text{zero}(n-1))].$$

We will prove a theorem asserting that the function zero equals 0 for all natural numbers.

THEOREM. $\forall n[n \neq \perp \supset \text{zero}(n)=0]$.

Proof. The proof proceeds by induction on n.

Basis: $n=0$.
This case is trivial by simplification: $\text{zero}(n)=\text{if } 0 \text{ equal } 0 \text{ then } 0 \text{ else zero}(n-1)=0$.

Induction step: $n>0$.
We assume by hypothesis that the theorem holds for all $n'<n$. Since $n>0$

$$\text{zero}(n)=\text{if } n \text{ equal } 0 \text{ then } 0 \text{ else zero}(n-1)=\text{zero}(n-1)$$

which is 0 by hypothesis.

Example 2. *Termination of an Ackermann function.* Let the function ack over the natural numbers **N** augmented by $\{\perp\}$ be defined by the call-by-value recursive program:

$$\begin{aligned}\text{ack}(x, y)=\ &\text{if } x \text{ equal } 0 \text{ then suc}(y)\\ &\text{else if } y \text{ equal } 0 \text{ then ack}(\text{pred}(x), 1)\\ &\text{else ack}(\text{pred}(x), \text{ack}(x, \text{pred}(y))).\end{aligned}$$

We will prove that ack is total.

THEOREM. $\forall x, y[x \neq \perp \wedge y \neq \perp \supset \text{ack}(x, y) \neq \perp]$.

Proof. The proof proceeds by induction on the pair $[x, y]$.

Basis: $x=0$.
By assumption, $y \neq \perp$. Hence, $\text{ack}(x, y)=\text{suc}(y) \neq \perp$.

Induction step: $x>0$.
By hypothesis, we assume the theorem holds for all $[x', y']$ such that either $x'<x$ or $x'=x$ and $y'<y$. Since $y \neq \perp$ by assumption,

$$\begin{aligned}\text{ack}(x, y)=\ &\text{if } y \text{ equal } 0 \text{ then ack}(\text{pred}(x), 1)\\ &\text{else ack}(\text{pred}(x), \text{ack}(x, \text{pred}(y)))\end{aligned}$$

Case (a). $y=0$.
In this case, $\text{ack}(x, y)=\text{ack}(\text{pred}(x), 1)$ which by hypothesis is a natural number (not $\perp$).

Case (b). $y>0$.

In this case,

$$\text{ack}(x, y) = \text{ack}(\text{pred}(x), \text{ack}(x, \text{pred}(y))).$$

By hypothesis, ack $(x, \text{pred}(y))$ is a natural number implying (by the induction hypothesis) that ack (pred (x), ack $(x, \text{pred}(y)))$ is a natural number. □

Example 3. *McCarthy's 91-function.* Let the function $f91$ over the integers augmented by $\{\perp\}$ be defined by the (call-by-name) recursive program

$$f91(n) = \text{if } n > 100 \text{ then } n - 10 \text{ else } f91(f91(n+11)).$$

We will prove the following theorem implying $f91$ is total over the integers.

THEOREM. $\forall n[n \neq \perp \supset f91(n) = \text{if } n > 100 \text{ then } n - 10 \text{ else } 91]$.

Proof. The proof proceeds by induction on $101 \ominus n$ where the binary operator $\ominus$ (monus) is defined by the equation

$$x \ominus y = \text{if } (x - y) > 0 \text{ then } x - y \text{ else } 0.$$

Basis: $101 \ominus n = 0$.
Clearly, $n > 100$, implying $f91(n) = n - 10$, which is exactly what the theorem asserts.

Induction step: $101 \ominus n > 0$.
By hypothesis, we assume the theorem holds for n' such that $101 \ominus n' < 101 \ominus < n$, i.e. $n' > n$. By the definition of $f91$,

$$f91(n) = f91(f91(n+11)) = f91(\text{if } n+11 > 100 \text{ then } n+1 \text{ else } 91)$$
(by induction since $n + 11 > n$).

By assumption, $n \leqq 100$. Consequently, there are two cases we must consider.

Case (a). $n + 11 > 100$.
Obviously, $100 \geqq n > 89$, implying

$$f91(n) = f91(n+1) = \text{if } n+1 > 100 \text{ then } n-9 \text{ else } 91 = 91 \text{ (since } n \leqq 100).$$

Case (b). $n + 11 \leqq 100$.
By assumption, $n \leqq 89$, implying

$$f91(n) = f91(91) = \text{if } 91 < 100 \text{ then } 91 - 10 \text{ else } 91$$
(by induction since $91 > n) = 91$. □

Appendix III. Proof of call-by-value fixed point normalization theorem.

THEOREM. *Let P be a call-by-value recursive program over an arithmetic data domain* **D** *and let* **F** *(abbreviating* $[\mathbf{f}_1, \cdots, \mathbf{f}_n]$*) denote the least fixed-point of the call-by-value functional* $\mathbf{P}_\perp$ *for P. The complete recursive program* P^* *corresponding to P has the following properties*:

(i) P^* *is complete, i.e. the corresponding call-by-value functional* $\mathbf{P}'_\perp$ *as a unique fixed-point* $[\mathbf{f}_1^*, \cdots, \mathbf{f}_n^*]$.

(ii) *For* $i = 1, \cdots, n$, $\mathbf{last}(\mathbf{f}_i^*(\bar{d})) = \mathbf{f}_i(\bar{d})$ *for all* $\# f_i$*-tuples* $\bar{d}$ *over* $|D|$.

Proof. By definition, **F** is the least upper bound of the chain of approximating n-tuples of functions

$$\mathbf{F}^{(0)} \subseteq \mathbf{F}^{(1)} \subseteq \cdots \subseteq \mathbf{F}^{(k)} \subseteq \cdots$$

where $\mathbf{F}^{(k)} = [\mathbf{f}_1^{(k)}, \cdots, \mathbf{f}_n^{(k)}]$ is inductively defined $\forall k \geqq 0$ by

$$\mathbf{F}^{(0)} = [\lambda \bar{x}_1 \cdot \perp, \cdots, \lambda \bar{x}_n \cdot \perp]$$

$$\mathbf{F}^{(k+1)} = \mathbf{P}_\perp(\mathbf{F}^{(k)}).$$

Let $\mathbf{D}^{(k)}$ denote the structure $\mathbf{D} \cup \mathbf{F}^{(k)}$. In informal terms, $\mathbf{D}^{(k)}$ is $\mathbf{D}$ augmented by the call-by-value evaluation of P to depth k. Similarly, let $\mathbf{D}^{*(k)}$ denote the structure $\mathbf{D}^* \cup \mathbf{F}^{*(k)}$ where $\mathbf{F}^{*(k)}$ is the depth k approximation for P^* analogous to $\mathbf{F}^{(k)}$.

In the course of this proof, we will frequently employ the following lemma without explicitly citing it.

LEMMA 0. *For every term t in L_P, every state s over $|D|$, and all $k \geqq 0$, $\mathbf{D}^{*(k)}[\mathrm{seq}\,(t^*)][s] = \mathbf{TRUE}$, i.e., t^* denotes a sequence code in $\mathbf{D}^*$.*

Proof of Lemma 0. A routine induction on the structure of t. Omitted. □

Property (ii) of the main theorem is an immediate consequence of the following lemma.

LEMMA 1. *For every term t in L_P, every state s over $|D|$, and all $k \geqq 0$, $\mathbf{D}^{(k)}[t][s] = \mathbf{D}^{*(k)}[\mathrm{last}\,(t^*)][s]$.*

Proof of Lemma 1. The proof proceeds by induction on the pair $[k, t]$. By hypothesis, we may assume that the lemma holds for all $[q, u]$ where either $q < k$ and u is arbitrary, or $q = k$ and u is a proper subterm of t.

Case (a). t is a constant or a variable x. Then t^* has the form $\mathrm{mkseq}(x)$ implying $\mathbf{D}^{*(k)}[\mathrm{last}\,(t^*)][s] = \mathbf{D}^{*(k)}[x][s] = \mathbf{D}^{(k)}[t][s]$ for arbitrary $k \geqq 0$.

Case (b). t has the form $g(u_1, \cdots, u_{\#g})$, $g \in G \cup \mathrm{Seq}$. Then

$$t^* = u_1^* \circ \cdots \circ u_{\#g}^* \circ \mathrm{mkseq}\,(g(\mathrm{last}\,(u_1^*), \cdots, \mathrm{last}\,(u_{\#g}^*))).$$

By hypothesis, $\mathbf{D}^{*(k)}[\mathrm{last}\,(u_i^*)][s] = \mathbf{D}^{(k)}[u_i][s]$, for all $s, i = 1, \cdots, \#g$. If for some i, $\mathbf{D}^{(k)}[u_i][s] = \bot$, then $\mathbf{D}^{*(k)}[\mathrm{last}\,(u_i^*)] = \bot$, implying $\mathbf{D}^{(k)}[t][s] = \mathbf{D}^{*(k)}[\mathrm{last}\,(t^*)][s] = \bot$, since $\mathbf{g}$, $\circ$, and **last** are all strict functions. On the other hand, if $\mathbf{D}^{(k)}[u_i][s] \neq \bot$ for all i, we conclude by the induction hypothesis and the definition of last that $\mathbf{D}^{*(k)}[u_i^*][s] \neq \bot$ for all i, implying

$$\begin{aligned}\mathbf{D}^{*(k)}[\mathrm{last}\,(t^*)][s] &= \mathbf{D}^{*(k)}[g(\mathrm{last}\,(u_i^*), \cdots, \mathrm{last}\,(u_{\#g}^*))][s]\\ &= \mathbf{D}^{(k)}[g(u_1, \cdots, u_{\#g})][s] \quad \text{(by induction)}\\ &= \mathbf{D}^{(k)}[t][s].\end{aligned}$$

Case (c). t has the form $f_i(u_1, \cdots, u_{\#f_i})$. If $k = 0$, the proof is trivial. For positive k, the proof breaks down into two cases. First, if $\mathbf{D}^{(k)}[u_j][s] = \bot$ for some j, then the lemma holds by exactly the same reasoning as in case 2. On the other hand, given $\mathbf{D}^{*(k)}[u_j^*][s] \neq \bot$ for all j, we deduce that

$$\begin{aligned}\mathbf{D}^{*(k)}[\mathrm{last}\,(t^*)][s] &= \mathbf{D}^{*(k)}[\mathrm{last}\,(f_i^*\,(\mathrm{last}\,(u_1^*), \cdots, \mathrm{last}\,(u_{\#f_i}^*)))][s]\\ &= \mathbf{D}^{*(k)}[\mathrm{last}\,(f_i^*\,(\bar{x}))][s^*]\end{aligned}$$

where s^* is a state binding each variable x_j to $\mathbf{D}^{*(k)}[\mathrm{last}\,(u_j^*)][s] = \mathbf{D}^{(k)}[u_j][s]$, $j = 1, \cdots, \#f_i$. Since no x_j is bound to $\bot$, we can expand $f_i^*\,(\bar{x})$ to produce

$$\begin{aligned}\mathbf{D}^{*(k)}[\mathrm{last}\,(t^*)][s^*] &= \mathbf{D}^{*(k-1)}[\mathrm{last}\,(t_i^*)][s^*]\\ &= \mathbf{D}^{(k-1)}[t_i][s^*] \quad \text{(by induction)}\\ &= \mathbf{D}^{(k)}[f_i(\bar{x})][s^*]\\ &= \mathbf{D}^{(k)}[t][s].\end{aligned}$$

Case (d). t has the form if u_0 then u_1 else u_2. By definition $t^* = u_0^* \circ (\text{if last}\,(u_0^*)$ then u_1^* else $u_2^*)$. If $\mathbf{D}^{(k)}[u_0][s] = \bot$, then by induction $\mathbf{D}^{*(k)}[\mathrm{last}\,(u_0^*)] = \bot$, implying $\mathbf{D}^{*(k)}[\mathrm{last}\,(t^*)][s] = \bot$ and $\mathbf{D}^{(k)}[t][s] = \bot$. *On the other hand, if $\mathbf{D}^{(k)}[u_0][s] \neq \bot$*, then it

denotes either "true" or "false". If $\mathbf{D}^{(k)}[u_0][s]$ is a "true" element of $\mathbf{D}$ then $\mathbf{D}^{(k)}[t][s] = \mathbf{D}^{(k)}[u_1][s]$. By induction,

$$\mathbf{D}^{(k)}[u_0][s] = \mathbf{D}^{*(k)}[\text{last}\,(u_0^*)][s]$$

implying

$$\begin{aligned}\mathbf{D}^{*(k)}[t^*][s] &= \mathbf{D}^{*(k)}[\text{last}\,(u_1^*)][s]\\ &= \mathbf{D}^{(k)}[u_1][s] \quad \text{(by induction)}\\ &= \mathbf{D}^{(k)}[t][s].\end{aligned}$$

An analogous argument proves the "false" case. □ (Lemma 1)

We prove property (ii) of the theorem as follows. By Lemma 1,

$$\begin{aligned}&\mathbf{D}^{(k)}[f_i(x_1, \cdots, x_{\#f_i})][s]\\ &\quad = \mathbf{D}^{*(k)}[\text{last}\,(\text{mkseq}\,(x_1) \circ \cdots \circ \text{mkseq}\,(x_{\#f_i}) \circ f_i^*\\ &\qquad\qquad (\text{last}\,(\text{mkseq}\,(x_1)), \cdots, \text{last}\,(\text{mkseq}\,(x_{\#f_i})))))][s]\end{aligned}$$

for all $k \geqq 0$, all states s over $|D|$. Simplifying the right-hand side of the preceding equation yields

$$\mathbf{D}^{(k)}[f_i(x_1, \cdots, x_{\#f_i})][s] = \mathbf{D}^{*(k)}[\text{last}\,(f_i^*(x_1, \cdots, x_{\#f_i}))][s]$$

for all states s over $|D|$. Since $\mathbf{D}$ and $\mathbf{D}^*$ are both flat domains, the functions $\mathbf{f}_i$ and $\mathbf{f}_i^*$ have the following property. For any $\#f_i$-tuple d over $|D|$ there exists $p \geqq 0$ such that

$$\mathbf{f}_i^{(p)}(\bar{d}) = \mathbf{f}_i(\bar{d})$$

and

$$\mathbf{f}_i^{*(p)}(\bar{d}) = \mathbf{f}_i^*(\bar{d}).$$

Let s_d be a state mapping $\bar{x}$ into $\bar{d}$. Then

$$\begin{aligned}\mathbf{f}_i(\bar{d}) = \mathbf{f}_i^{(p)}(\bar{d}) &= \mathbf{D}^{(p)}[f_i(\bar{x})][s_d]\\ &= \mathbf{D}^{*(p)}[\text{last}\,(f_i^*(\bar{x}))][s_d] = \mathbf{last}\,(\mathbf{f}_i^{*(p)}(\bar{d})) = \mathbf{last}\,(\mathbf{f}_i^*(\bar{d}))\end{aligned}$$

proving property (ii).

To prove property (i), we must introduce some new definitions. Let $\mathbf{H}$ be an n-tuple of strict functions $[\mathbf{h}_1, \cdots, \mathbf{h}_n]$ over $\mathbf{D}$ corresponding to the n-tuple of function symbols $[f_1^*, \cdots, f_n^*]$. We define $\mathbf{D}_H^{*(k)}$ for all $k \geqq 0$ as the structure corresponding to L_P^* that is identical to $\mathbf{D}^{*(k)}$ except that $\mathbf{D}_H^{*(k)}$ interprets $[f_1^*, \cdots, f_n^*]$ by $\mathbf{F}_H^{*(k)}$ where $\mathbf{F}_H^{*(k)}$ is inductively defined by

$$\mathbf{F}_H^{*(0)} = \mathbf{H},$$

$$\mathbf{F}_H^{*(k+1)} = \mathbf{P}_\perp(\mathbf{F}_H^{*(k)}).$$

Informally $\mathbf{f}_{i_H}^{*(k)}$ is the function computed by applying call-by-value evaluation to the recursion equation for f_i where invocations of f_j, $j = 1, \cdots, n$, at depth k are interpreted by the function $\mathbf{h}_j$ instead of ordinary evaluation. If all the functions $\mathbf{h}_j$ in $\mathbf{H}$ are everywhere undefined, then $\mathbf{F}_H^{*(k)} = \mathbf{F}^{*(k)}$.

Property (i) is a simple consequence of the following lemma.

LEMMA 2. *Let t be any term in L_P. Let $\mathbf{H}$ be an n-tuple of strict functions $[\mathbf{h}_1, \cdots, \mathbf{h}_n]$ corresponding to $[f_1^*, \cdots, f_n^*]$. Then for any $k \geqq 0$ and any state s over $|D|$, $\mathbf{D}^*[t^*][s] = \perp$ implies either $\mathbf{D}_H^{*(k)}[t^*][s] = \perp$, or* **length** $(\mathbf{D}_H^{*(k)}[t^*][s]) \geqq k$.

Proof of Lemma 2. In the course of the proof, we will use the following lemmas which are easily proven by structural induction on t.

LEMMA 2a. *For any term t in L_{P^*}, $\mathbf{D}^{*(k)}[t][s] \neq \perp$ implies $\mathbf{D}^{*(k)}[t][s] = \mathbf{D}_H^{*(k)}[t][s]$.*

LEMMA 2b. *For any term t in L_{P^*} not containing any recursive function symbol f_i^*, $\mathbf{D}_H^{*(k)}[t][s] = \mathbf{D}^{*(k)}[t][s]$ for arbitrary state s.*

Proof of Lemmas 2a *and* 2b. Omitted. □

To prove lemma 2, we apply induction on the pair $[k, t]$.

Basis: $k = 0$. In this case, the lemma is a trivial consequence of the definition of **length** and the fact that the meaning of any computation sequence term t^* under any state s in the structure $\mathbf{D}^*$ is either $\perp$ or a sequence code.

Induction step: $k > 0$. We perform a case split on the structure of t.

Case (a). t is a constant or variable x. Then $t^* = \text{mkseq}(x)$, implying by Lemma 2b that $\mathbf{D}_H^{*(k)}[t^*][s] = \mathbf{D}^{*(k)}[t^*][s]$ which is $\perp$ by assumption.

Case (b). t has the form $g(u_1, \cdots, u_{\#g})$ where $g \in G$. In this case, $t^* = u_1^* \circ \cdots \circ u_{\#g}^* \circ \text{mkseq}(g(\text{last}(u_1^*), \cdots, \text{last}(u_{\#g}^*)))$. If $\mathbf{D}^{*(k)}[u_j^*][s] = \perp$, then by induction, either $\mathbf{D}_H^{*(k)}[u_j^*][s] = \perp$ or $\mathbf{length}(\mathbf{D}_H^{*(k)}[u_j^*][s]) \geqq k$, implying the lemma holds (since $\circ$ is strict). On the other hand, if $\mathbf{D}^{(k)}[u_j^*][s] \neq \perp$ for all j, then by Lemma 2a, $\mathbf{D}_H^{*(k)}[u_j^*][s] = \mathbf{D}^{*(k)}[u_j^*][s]$ for all j. Consequently,

$$\begin{aligned}\mathbf{D}_H^{*(k)}[t^*][s] &= \mathbf{D}_H^{*(k)}[u_1^* \circ \cdots \circ u_{\#g}^* \circ \text{mkseq}(g(\text{last}(u_1^*), \cdots, \text{last}(u_{\#g}^*)))][s] \\ &= \mathbf{D}^{*(k)}[u_1^* \circ \cdots \circ u_{\#g}^* \circ \text{mkseq}(g(\text{last}(u_1^*), \cdots, \text{last}(u_{\#g}^*)))][s] \\ &= \mathbf{D}^{*(k)}[t^*][s].\end{aligned}$$

implying the lemma holds.

Case (c). t has the form $f_i(u_1, \cdots, u_{\#f_i})$. If $\mathbf{D}^{*(k)}[u_j^*][s] = \perp$ for some j, the proof is identical to the analogous section of the previous case. On the other hand, when $\mathbf{D}^{*(k)}[u_j^*][s] \neq \perp$ for all j,

$$\begin{aligned}\mathbf{D}_H^{*(k)}[t^*][s] &= \mathbf{D}^{*(k)}[u_1^*][s] \circ \cdots \circ \mathbf{D}^{*(k)}[u_{\#f_i}^*] \\ &\quad [s] \circ \mathbf{f}_{i_H}^{*(k)}(\mathbf{last}(\mathbf{D}^{*(k)}[u_1^*][s]), \cdots, \mathbf{last}(\mathbf{D}^{*(k)}[u_{\#f_i}^*][s])) \\ &= \mathbf{D}^{*(k)}[u_1^*][s] \circ \cdots \circ \mathbf{D}^{*(k)}[u_{\#f_i}^*][s] \circ \mathbf{D}_H^{*(k)}[f_i^*(\bar{x})][s^*] \\ &= \mathbf{D}^{*(k)}[u_1^*][s] \circ \cdots \circ \mathbf{D}^{*(k)}[u_{\#f_i}^*][s] \circ \mathbf{D}_H^{*(k-1)}[t_i^*][s^*]\end{aligned}$$

where s^* maps x_j into $\mathbf{last}(\mathbf{D}^{*(k)}[u_j^*][s])$, $j = 1, \cdots, \#f_i$. By induction, either $\mathbf{D}_H^{*(k-1)}[t_i^*][s^*] = \perp$ or $\mathbf{length}(\mathbf{D}_H^{*(k-1)}[t_i^*][s^*]) \geqq k-1$. Hence the lemma clearly holds (since $\mathbf{length}(u_j^*) \geqq 1$ for all j).

Case (d). t has the form if u_0 then u_1 else u_2. If $\mathbf{D}^{*(k)}[u_0^*][s] = \perp$, the proof is identical to the analogous section of case (b). On the other hand, when $\mathbf{D}^{(k)}[u_0][s] \neq \perp$, $\mathbf{D}^{(k)}[u_0][s]$ is either a "true" element of $|D|$ or a "false" one. In the former case, $\mathbf{D}^{*(k)}[t][s] = \mathbf{D}^{*(k)}[u_1^*][s]$ and $\mathbf{D}_H^{*(k)}[t^*][s] = \mathbf{D}_H^{*(k)}[u_1^*][s]$. By induction, $\mathbf{D}^{*(k)}[u_1^*][s] = \perp$, simplying either $\mathbf{D}_H^{*(k)}[u_1^*] = \perp$ or $\mathbf{length}(\mathbf{D}_H^{*(k)}[u_1^*]) \geqq k$. Hence the lemma holds in this case. An analogous argument holds for the "false" case. □ (Lemma 2)

Given Lemma 2, we prove that property (i) holds by the following argument. Let $\mathbf{H} = [\mathbf{h}_1, \cdots, \mathbf{h}_n]$ be any fixed-point of the call-by-value function $\mathbf{P}_\perp$ for the complete recursive program P^*, i.e., for all i

$$\mathbf{D}_H^{*(0)}[f_i^*(\bar{x}_i)][s] = \mathbf{D}_H^{*(0)}[t_i^*][s].$$

By induction on $[k, t]$ we can easily show for all $k \geqq 0$, all terms t in L_{P^*}, and all states s that

$$\mathbf{D}_H^{*(k)}[t^*][s] = \mathbf{D}_H^{*(0)}[t^*][s].$$

Now assume $\mathbf{H}$ is not the least fixed-point of $\mathbf{P}_\perp$, i.e., $\mathbf{f}_i^*(\bar{d}) = \perp$ but $\mathbf{h}_i(\bar{d}) \neq \perp$ for some i and some $\#f_i$-tuple $\bar{d}$ over $|D|$. Then,

$$\forall k \geqq 0[\mathbf{D}_H^{*(k)}[f_i^*(\bar{x}_i)][s_d] = \mathbf{D}_H^{*(0)}[f_i^*(\bar{x}_i)][s_d] = \mathbf{h}_i(\bar{d})]$$

where s_d binds $\bar{x}_i$ to $\bar{d}$. By Lemma 2, the length of $\mathbf{h}_i(\bar{d})$ is greater than any number k, contradicting the fact that

$$\exists k[\mathbf{length}\,(\mathbf{D}_H^{*(0)}[f_i^*(\bar{x}_i)](s_d)) \leqq k]. \quad \square$$

Appendix IV. Call-by-name complete recursive programs. The call-by-value recursive program construction described in § 8 exploited the idea of defining a new function f_i^* for each function f_i in the original program such that f_i^* constructs the call-by-value computation sequence for f_i. We will utilize essentially the same idea to construct complete call-by-name programs.

Unfortunately, call-by-name computation sequences have a more complex structure than their call-by-value counterparts. The chief complication is that the collection of arguments in recursive call $f_i(\bar{t})$ that are actually evaluated depends on the particular arguments $\bar{t}$. To accommodate this complication, we adopt the convention that the new functions f_i^* in the complete recursive program take computation sequences corresponding to the arguments of f as inputs instead of the arguments themselves. Hence, the original functions $f_1, \cdots, f_n$ are related to the new functions $f_1^*, \cdots, f_n^*$ by the equations

$$f_i(x_1, \cdots, x_{\#f_i}) = \text{last}\,(f_1^*(\text{mkseq}\,(x_1), \cdots, \text{mkseq}\,(x_{\#f_i}))), i = 1, \cdots, n$$

instead of

$$f_i(x_1, \cdots, x_{\#f_i}) = \text{last}\,(f^*(x_1, \cdots, x_{\#f_i})), i = 1, \cdots, n,$$

which hold for call-by-value complete recursive programs.

This convention gives the body of each new function f_i^* control over the process incorporating particular argument evaluations into the output computation sequence. If a particular argument is never evaluated, its computation sequence is discarded. Recall that in the call-by-value construction, the computation sequences for the arguments of a function call are unconditionally inserted in the computation sequence by the calling expression.

We construct the call-by-name complete recursive program P^* corresponding to P as follows.

DEFINITION. Let $\mathbf{D}$, L_D, P, L_P, $\mathbf{F}$, $\mathbf{D}^{(k)}$, and $\mathbf{D}^*$ (including **Seq**) be defined as in § 8 except that P is a call-by-name program and, consequently, $\mathbf{F}$ is the least fixed-point of the (call-by-name) functional $\mathbf{P}$ for P over $\mathbf{D}$. Let t be an arbitrary term in the language L_P. The *call-by-name computation sequence term* t^* (in the extended language L_{P^*}) corresponding to t is inductively defined by:

(i) If t is a variable v,

$$t^* = v.$$

(ii) If t is a constant symbol c,

$$t^* = \text{mkseq}\,(c).$$

(iii) If t has the form $g(u_1, \cdots, u_{\#g})$ where $g \in G$,

$$t^* = u_1^* \circ \cdots \circ u_{\#g}^* \circ \text{mkseq}\,(g(\text{last}\,(u_1^*), \cdots, \text{last}\,(u_{\#g}^*))).$$

(iv) If t has the form $f_i(u_1, \cdots, u_{\#f_i})$,

$$t^* = \text{mkseq}\,(\text{true}) \circ f_i^*\,(u_1^*, \cdots, u_{\#f_i}^*).$$

(v) If t has the form if u_0 then u_1 else u_2,

$$t^* = u_0^* \circ (\text{if last}\,(u_0^*) \text{ then } u_1^* \text{ else } u_2^*).$$

The *complete recursive program P^** corresponding to P is the (call-by-name) program

$$\{f_1^*\,(\bar{x}_1) = t_1^*, \cdots, f_n^*\,(\bar{x}_n) = t_n^*\}$$

over $\mathbf{D}^*$.

THEOREM (call by-name fixed point normalization theorem). *Let P be a call-by-name recursive program over an arithmetic data domain $\mathbf{D}$ supporing elementary syntax and let $[\mathbf{f}_1, \cdots, \mathbf{f}_n]$ denote the least fixed-point of the corresponding (call-by-name) functional $\mathbf{P}$. The complete recursive program P^* corresponding to P has the following properties*:

(i) *P^* is complete, i.e. the corresponding (call-by-name) functional $\mathbf{P}^*$ has a unique fixed-point $[\mathbf{f}_1^*, \cdots, \mathbf{f}_n^*]$.*

(ii) For $i = 1, \cdots, n$, $\mathbf{last}\,(\mathbf{f}_i^*(\mathbf{mkseq}\,(d_1), \cdots, \mathbf{mkseq}\,(d_{\#f_i}))) = \mathbf{f}_i(\bar{d})$ for all $\#f_i$-tuples $\bar{d}$ over $|D|$.

Proof. The proof follows the same outline as the proof of the corresponding theorem for call-by-value fixed-points in § 8. Recall that $\mathbf{F}$ is the least upper bound of the chain of function n-tuples

$$\mathbf{F}^{(0)} \subseteq \mathbf{F}^{(1)} \subseteq \cdots \subseteq \mathbf{F}^{(k)} \subseteq \cdots$$

where $\mathbf{F}^{(k)} = [\mathbf{f}_1^{(k)}, \cdots, \mathbf{f}_n^{(k)}]$ is defined $\forall k \geqq 0$ by

$$\mathbf{F}^{(0)} = [\lambda \bar{x}_1 \cdot \bot, \cdots, \lambda \bar{x}_n \cdot \bot],$$

$$\mathbf{F}^{(k+1)} = \mathbf{P}(\mathbf{F}^{(k)}),$$

and that $\mathbf{D}^{(k)}$ denotes the structure $\mathbf{D} \cup \mathbf{F}^{(k)}$. Let $\mathbf{D}^{(k)*}$ denote the structure $\mathbf{D} \cup \mathbf{F}^{*(k)}$ for the call-by-name program P^* over $\mathbf{D}^*$ analogous to $\mathbf{D}^{(k)}$ for the program P over $\mathbf{D}$.

Property (ii) is a simple consequence of the following lemma.

LEMMA 1. For every term t in L_P, every state s over $|D^*|$, and all $k \geqq 0$, $\mathbf{D}^{(k)}[t][s] = \mathbf{D}^{*(k)}[\text{last}\,(t^*)][s]$.

Proof (*of Lemma* 1). The proof is essentially identical to the proof of the corresponding lemma in § 8, which proceeds by induction on the pair $[k, t]$. The details are left to the reader. □

Property (ii) follows immediately from Lemma 1 by the following argument. For any function f_i and $\#f_i$-tuple $\bar{d} = [d_1, \cdots, d_{\#f_i}]$ over D there exists $p \geqq 0$ such that

$$\mathbf{f}_i^{(p)}(\bar{d}) = f_i(\bar{d})$$

and

$$\mathbf{f}_i^{*(p)}(\mathbf{mkseq}\,(d_1), \cdots, \mathbf{mkseq}\,(d_{\#f_i})) = \mathbf{f}_i^*(\mathbf{mkseq}\,(d_1), \cdots, \mathbf{mkseq}\,(d_{\#f_i})).$$

Let s be a state mapping $\bar{x}$ into $\bar{d}$, and s_{mkseq} be the state mapping each variable x_j

into **mkseq** $(s(x_j))$. Then

$$\begin{aligned}\mathbf{f}_i(\bar{d}) &= \mathbf{f}_i^{(p)}(\bar{d})\\ &= \mathbf{D}^{(p)}[f_i(\bar{x})][s]\\ &= \mathbf{D}^{*(p)}[\text{last (mkseq } (d^*) \circ f_i^* (\bar{x}))][s_{\text{mkseq}}]\\ &= \mathbf{D}^{*(p)}[\text{last } (f_i^* (\bar{x}))][s_{\text{mkseq}}]\\ &= \textbf{last } (\mathbf{f}_i^{*(p)}(\textbf{mkseq } (d_1), \cdots, \textbf{mkseq } (d_{\#f_i})))\\ &= \textbf{last } (\mathbf{f}_i^*(\textbf{mkseq } (d_1), \cdots, \textbf{mkseq } (d_{\#f_i})))\end{aligned}$$

proving property (ii).

To prove property (i), we must utilize the definitions introduced in the analogous proof in Appendix II. Let $\mathbf{H}$, $\mathbf{D}_H^{*(k)}$, and $\mathbf{F}_H^{*(k)}$ be defined exactly as in §8, except substitute call-by-name semantics for call-by-value semantics in the definition of $\mathbf{F}_H^{*(k)}$ and $\mathbf{D}_H^{*(k)}$. In other words,

$$\mathbf{F}_H^{*(0)} = \mathbf{H},$$

$$\mathbf{F}_H^{*(k+1)} = \mathbf{P}(\mathbf{F}_H^{*(k)}).$$

Since the remaining details of the proof of property (i) are nearly identical to those found in the corresponding proof in Appendix III, they are omitted. □

It is a straightforward exercise to formulate and prove call-by-name analogues to the corollary and the generalization of the fixed-point normalization theorem presented in § 8 for call-by-value programs. The details are left to the reader.

Acknowledgments. After studying a very early draft of the paper, Mike O'Donnell suggested that the fixed-point normalization theorem might generalize to all models (not just the standard ones) of suitable theories, motivating the corollary and generalization of the theorem. Two of my graduate students, Alex and Vladimir Yakhnis, discovered numerous errors and omissions in earlier drafts of the paper. Immediately prior to publication, Albert Meyer suggested several technical refinements, noticed the connection between extended first order programming logic and recent research by Andreka, Nemeti and Sain [1], and proposed many improvements in the presentation.

REFERENCES

[1] H. Andreka, I. Nemeti and I. Sain, *A complete logic for reasoning about programs via nonstandard model theory*, Theoret. Comput. Sci., 17 (1982), pp. 193–212, 259–278.

[2] J. Bell and M. Machover, *A Course in Mathematical Logic*, North-Holland, New York, 1977, pp. 316–324.

[3] R. Boyer and J. Moore, *Proving theorems about* LISP *functions*, J. Comput. Mach., 22 (1975), pp. 129–144.

[4] R. Boyer and J. Moore, *A Computational Logic*, Academic Press, New York, 1979.

[5] J. Cadiou, *Recursive definitions of partial functions and their computations*, Technical Report AIM-163, Computer Science Dep., Stanford Univ., Stanford, CA, 1972.

[6] R. Cartwright, *User defined data types as an aid to verifying* LISP *programs*, Proc. Third International Colloquium on Automata, Languages, and Programming, S. Michaelson and R. Milner, eds., Edinburgh Press, Edinburgh, 1976, pp. 228–256.

[7] ———, *A practical formal semantic definition and verification system for* TYPED LISP, Stanford A.I. Memo AIM-296, Stanford Univ., Stanford, CA, 1976 (also published as a monograph in the Outstanding Dissertations in Computer Science series, Garland Publishing Company, New York, 1979).

[8] ———, *First order semantics: A natural programming logic for recursively defined functions*, Technical Report TR 78–339, Computer Science Dept., Cornell University, Ithaca, NY, 1978.

[9] ———, *Computational models for programming logics*, Technical Report, Computer Science Program, Mathematical Sciences Department, Rice University, 1983.

[10] ———, *Nonstandard fixed-points*, Proc. of Logics of Programs Workshop, Carnegie-Mellon Univ., June 1983, Lecture Notes in Computer Science, Springer-Verlag, New York, 1983.

[11] J. W. DEBAKKER, *The Fixed Point Approach in Semantics: Theory and Applications*, Mathematical Centre Tracts 63, Free University, Amsterdam, 1975.

[12] J. W. DEBAKKER AND W. P. DEROEVER, *A calculus for recursive program schemes*, Automata, Languages, and Programming, M. Nivat, ed., North-Holland, Amsterdam, 1973, pp. 167–196.

[13] P. DOWNEY AND R. SETHI, *Correct computation rules for recursive languages*, this Journal, 5 (1976), pp. 378–401.

[14] H. ENDERTON, *A Mathematical Introduction to Logic*, Academic Press, New York, 1972.

[15] M. GORDON, R. MILNER AND C. WADSWORTH, *Edinburgh* LCF, Edinburgh Technical Report CSR-11-77, University of Edinburgh, Edinburgh, 1977.

[16] P. HITCHCOCK AND D. M. R. PARK, *Induction rules and proofs of program termination.* Proc. First International Colloquium on Automata, Languages, and Programming, M. Nivat, ed., North-Holland, Amsterdam, 1973, pp. 225–251.

[17] Z. MANNA, *Introduction to the Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

[18] J. MCCARTHY AND R. CARTWRIGHT, *Representation of recursive programs in first order logic*, Stanford Artificial Intelligence Memo AIM-324, Stanford Univ., Stanford, CA, 1979.

[19] R. MILNER, *Logic for computable functions—description of a machine implementation*, Stanford A.I. Memo AIM-169, Stanford Univ., Stanford, CA, 1972.

[20] ———, *Models of* LCF, Stanford A.I. Memo AIM-186, Stanford Univ., Stanford, CA, 1973.

[21]. R. MILNER, L. MORRIS AND M. NEWEY, *A logic for computable functions with reflexive and polymorphic types*, Proc. Conference on Proving and Improving Programs, Arc-et-Senons, 1975.

[22] M. O'DONNELL, *Computing in Systems Described by Equations*, Lecture Notes in Computer Science, 58, Springer-Verlag, New York, 1977.

[23] D. M. R. PARK, *Fixpoint induction and proof of program semantics*, in Machine Intelligence 5, B. Meltzer and D. Michie, eds., Edinburgh Press, Edinburgh, 1970, pp. 59–78.

[24] B. K. ROSEN, *Tree-manipulating systems and Church–Rosser theorems*, J. Assoc. Comput. Mach., 20 (1973), pp. 160–187.

[25] D. SCOTT AND J. W. DEBAKKER, *A theory of programs*, unpublished notes, IBM seminar, Vienna, 1969.

[26] D. SCOTT, *Data types as lattices*, this Journal, 5 (1976), pp. 522–586.

[27] A. TARSKI, *A lattice-theoretical fixpoint theorem and its applications*, Pacific J. Math., 5 (1955), pp. 285–309.

[28] J. VUILLEMIN, *Proof techniques for recursive programs*, Stanford A.I. Memo AIM-218, Stanford Univ., Stanford, CA, 1973.

[29] ———, *Correct and optimal implementations of recursion in a simple programming language*, J. Comput. Syst. Sci., 9 (1974), pp. 332–354.

SIAM J. COMPUT.
Vol. 13, No. 2, May 1984

© 1984 Society for Industrial and Applied Mathematics
012

SIMULATION OF PARALLEL RANDOM ACCESS MACHINES BY CIRCUITS*

LARRY STOCKMEYER† AND UZI VISHKIN‡

Abstract. A relationship is established between (i) parallel random-access machines that allow many processors to concurrently read from or write into a common memory including simultaneous reading or writing into the same memory location (CRCW PRAM), and (ii) combinational logic circuits that contain AND's, OR's and NOT's, with no bound placed on the fan-in of AND-gates and OR-gates. Parallel time and number of processors for CRCW PRAM's are shown to correspond respectively (and simultaneously) to depth and size for circuits, where the time-depth correspondence is to within a constant factor and the processors-size correspondence is to within a polynomial. By applying a recent result of Furst, Saxe and Sipser, we obtain the corollary that parity, integer multiplication, graph transitive closure and integer sorting cannot be computed in constant time by a CRCW PRAM with a polynomial number of processors. This is the first nonconstant lower bound on the parallel time required to solve these problems by a CRCW PRAM with a polynomial number of processors. We also state and outline the proof of a similar result, due to W. L. Ruzzo and M. Tompa, that relates time and processor bounds for CRCW PRAM's to alternation and space bounds for alternating Turing machines.

Key words. synchronous parallelism, parallel time complexity, circuit complexity, relating complexity measures, alternating Turing machines

1. Introduction and statements of results. Our main motivation for this work was the goal of proving nontrivial lower bounds on the time required by a parallel random-access machine to solve certain problems when the parallel RAM model allows simultaneous reading from and writing into a common memory. Following Snir [25], we call this model a *CRCW PRAM* (for concurrent-read concurrent-write parallel RAM). A CRCW PRAM has a sequence of RAM's R_1, R_2, R_3, ... operating synchronously in parallel. Each individual RAM is similar to a standard one-processor RAM (cf. [1, Chap. 1]). Each RAM has its own *local* infinite random-access memory and has instructions for addition, subtraction, conditional branches based on the predicates = and <, and reading and writing into its local memory. (For the moment we assume that there are no multiplication, division or Boolean instructions.) The RAM's also have access to a *common memory*, and each RAM has instructions for reading from and writing into the common

*Received by the editors June 3, 1982, and in revised form July 29, 1983. Portions of this paper have been reprinted, with permission, from "A complexity theory for unbounded fan-in parallelism" by A. K. Chandra, L. J. Stockmeyer and U. Vishkin, appearing in the Proceedings of the 23rd Annual Symposium on Foundations of Computer Science, 1982, pp.1-13, © 1982 IEEE. This paper was typeset at the IBM San Jose Research Laboratory.

†Computer Science Department, IBM Research Lab. K51/281, San Jose, California 95193. Work performed in part at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

‡Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, New York 10012. Work performed while the author was visiting the IBM Thomas J. Watson Research Center, while on leave from the Department of Computer Science, Technion, Haifa, Israel.

memory using one of its local registers to specify the common memory address. If more than one processor attempts to write into the same location in common memory at the same time, the lowest numbered processor succeeds. Each processor R_i has an instruction which loads its number i into a specified local register. All processors have the same program. Without loss of generality we assume that instructions are of the following forms, where M_1, M_2, M_3, ... denote local memory registers, and res (result), op1 and op2 (operand 1 and 2) are positive integers.

$M_{\mathrm{res}} \leftarrow$ constant
$M_{\mathrm{res}} \leftarrow$ processor number
$M_{\mathrm{res}} \leftarrow M_{\mathrm{op1}}$
$M_{\mathrm{res}} \leftarrow M_{\mathrm{op1}} @ M_{\mathrm{op2}}$, $@ \in \{+,-\}$
$M_{\mathrm{res}} \leftarrow *M_{\mathrm{op1}}$ {local,common} (indirect memory READ)
The contents of the {local,common} location whose address is in register M_{op1} is read into local register M_{res}.
$*M_{\mathrm{res}} \leftarrow M_{\mathrm{op1}}$ {local,common} (indirect memory WRITE)
The contents of local register M_{op1} is written into the {local,common} register whose address is in M_{res}.
GOTO label
GOTO label if $M_{\mathrm{op1}} @ M_{\mathrm{op2}}$, $@ \in \{=, <\}$
HALT

By convention, zero is the value read by an indirect READ using a nonpositive address. An indirect WRITE using a nonpositive address has no effect. In addition to the program, another part of the specification of a particular CRCW PRAM is a function $P(n)$ from positive integers to positive integers called the *processor bound*. An input of *size n* consists of n binary words, each of length at most n. A CRCW PRAM is given an input of size n by placing the n words in the first n locations of common memory, and the first $P(n)$ processors $R_1, \ldots, R_{P(n)}$ are started. Each instruction takes one time unit (uniform cost criterion). The computation halts when $R_1, \ldots, R_{P(n)}$ have all halted. The machine *operates in time* $T(n)$ if it halts within $T(n)$ steps on any input of size n. Output conventions are not critical, but we assume that when the computation halts, the output is in an initial contiguous block of common memory of length at most n locations.

This definition is based largely on the definition of Shiloach and Vishkin [24] (simultaneous writing as defined above is defined in [11], [26]); they use this model to give upper bounds on parallel time for several problems. [24] contains many references to other papers that give algorithms that can be implemented on a CRCW PRAM or restricted versions of it. The model is essentially identical to the SIMDAG of Goldschlager [11] and similar to the P-RAM of Fortune and Wyllie [9]. The P-RAM of [9] allows simultaneous reading but no simultaneous writing to the same common memory location; we call this model a *CREW PRAM* (for concurrent-read exclusive-write). (Whereas [9] and [11] characterized the power of these models when the number of processors grows exponentially in n, we are concerned mainly with the case of a polynomially bounded number of processors.) The model is also mentioned by Schwartz [23];

although he notes the physical difficulties of implementing large fan-in, he does state that such models "can play a useful role as theoretical yardsticks for measuring the limits of parallel computation."

For some models of parallel computation, fan-in considerations lead easily to a lower bound of $\Omega(\log n)$ on parallel time. For the CREW PRAM, the fan-in argument is considerably subtle, but Cook and Dwork [7] and Reischuk [19] succeed in proving $\Omega(\log n)$ lower bounds for problems such as computing the OR of n bits or the minimum of n numbers. For the CRCW PRAM, arguments based on bounded fan-in no longer work. For example, Shiloach and Vishkin [24] show that a CRCW PRAM with $O(n^2)$ processors can find the minimum of n numbers in constant time. However, Vishkin and Wigderson [28] recently proved nonconstant lower bounds in the case where the size of the common memory is limited. In fact, they prove a trade-off of $mT^2 = \Omega(n)$ for problems such as parity of n bits in the CRCW PRAM, where m is the size of the common memory, T is the parallel time and the input is in a read-only common memory. This result holds for any number of processors.

To aid our understanding of the power of CRCW PRAM's and to facilitate proofs of lower bounds, a characterization in terms of circuits would seem useful. There is considerable precedent for circuit characterizations of other computational models. Pippenger and Fischer [16] show a correspondence between serial (e.g., Turing machine) time and circuit size, and Borodin [2] shows a correspondence between serial space and circuit depth. Correspondences involving simultaneous resource bounds are given by Dymond and Cook [8], Hong [12], Pippenger [15], and Ruzzo [20]. Lev [13] shows a correspondence between time on a parallel RAM that does not allow simultaneous writing and circuit depth (unlike our result, the time-depth correspondence involves a $\log n$ factor rather than a constant factor). Whereas these correspondences use circuits with fan-in bounded by 2, the "correct" circuit analogue for CRCW PRAM's is the unbounded fan-in circuit studied by Furst, Saxe and Sipser [10]. These are acyclic circuits containing AND-gates, OR-gates, and NOT-gates.

More precisely, a *circuit* is an acyclic directed graph. Each node of the graph is labeled as either an input node, an AND-gate, an OR-gate, or a NOT-gate. Input nodes must have fan-in zero, and NOT-gates must have fan-in one. In addition, certain nodes are designated as output nodes. An assignment of Boolean values to all input nodes extends, in the obvious way, to Boolean values associated with all nodes. The *size* of a circuit is the number of edges (i.e., wires). The *depth* of a circuit is the length of a longest path from some input to some output.

Our main result is the following:

THEOREM 1. *There is a constant c and a function $q(P, T, n)$ bounded above by a polynomial in P, T, n, such that the following holds. Let R be a* CRCW PRAM *with processor bound $P(n)$ that operates in time $T(n)$. There is a constant d_R and, for each n, a circuit C_n of size $d_R \cdot q(P(n), T(n), n)$ and depth $c \cdot T(n)$ such that C_n realizes the input-output behavior of R on inputs of size n.*

Remarks.

1. $q(P,T,n) = O(PTL(L^2 + PT))$ where $L = O(n + T + \log P)$.

2. Theorem 1 remains true if the PRAM has instructions for bitwise Boolean operations and, or, negation, etc., as in vector machines [17].

3. Theorem 1 is not true if the individual RAM's have a unit cost multiplication instruction, since [10] shows that multiplication of n-bit numbers cannot be realized by a circuit of polynomial size and constant depth. However, Theorem 1 is true (with a different polynomial q) if multiplication and division are restricted to $O(\log n)$-bit numbers.

4. As noted above, concurrent write conflicts into the same common memory location are resolved by allowing the lowest numbered processor to write. One can imagine other reasonable alternatives. For example, one can assume that the program works correctly no matter which processor succeeds in writing, or that the program is such that whenever more than one processor attempts to write into the same location concurrently they are all writing the same thing (this latter assumption is used in [24]). Clearly Theorem 1 holds for CRCW PRAM's with either of these alternate assumptions. Theorem 1 holds for the CREW PRAM as well.

One advantage of circuits is that lower bounds may be easier to prove in the context of the fixed structure of circuits rather than the dynamic nature of a program. In this regard, since Furst, Saxe, and Sipser [10] have shown that polynomial-size and constant-depth circuits cannot compute parity, an immediate corollary is that a CRCW PRAM with a polynomially-bounded number of processors cannot compute parity in constant time. The same corollary holds for any function to which parity is reducible by a polynomial-size constant-depth circuit. Two examples of such functions from [10] are integer multiplication and graph transitive closure. Other examples from [5], [6] are determining whether a graph has a perfect matching and sorting (binary representations of) positive integers.

COROLLARY 1. *A* CRCW PRAM *with a polynomially-bounded number of processors that operates in constant time cannot compute parity, multiply integers, find the transitive closure of a graph, determine whether a graph has a perfect matching, or sort integers.*

Even given the elegant proof technique of [10], a direct proof of Corollary 1 would probably be very cumbersome.

To provide evidence that we have found the "correct" circuit analogue of CRCW PRAM's we can give a converse to Theorem 1. To avoid the usual mismatch of uniform programs with nonuniform circuits, we now allow programs to be nonuniform. In the nonuniform case, each R_i can have a different program, and the programs can depend on the size of the input. Theorem 1 remains true for nonuniform CRCW PRAM's although the polynomial q now depends also on the size of the programs, where the *size* of a program is the number of bits needed to write the program when constants and indices of registers mentioned in the program are written in binary.

THEOREM 2. *There are constants c_1, c_2 and c_3 such that the following holds. Let C be a circuit of size S and depth T that computes a function f having n inputs and at most n outputs. There is a nonuniform* CRCW PRAM *with $c_1(S+n)$ processors and program size $c_2\log(S+n)$ that runs in time $c_3(T+1)$ and computes f.*

Remark. Theorem 2 remains true under the two alternate concurrent write assumptions for CRCW PRAM's mentioned in Remark 4 above. However, we do not see how to prove Theorem 2 for the CREW PRAM. Since [7], [19] shows that an n-bit OR requires time $\Omega(\log n)$ on a CREW PRAM, Theorem 2

does not hold literally for the CREW PRAM; however, a time bound of $O(T + \log n)$ would not contradict [7], [19]. We leave circuit characterizations of the CREW PRAM and EREW PRAM [14], [25] as interesting open questions (the EREW PRAM allows neither simultaneous reading nor writing to the same location). If natural circuit characterizations of the CREW PRAM and EREW PRAM cannot be found, this may strengthen [26], [27] in supporting the CRCW PRAM model of computation if some kind of simultaneous access to a common memory is to be allowed.

Before giving the proofs, a few remarks on our view of the significance of this work should be made. First, a few words should be said about how realistic is the CRCW PRAM model. Certainly any physically realizable computer using current technology cannot have completely unbounded fan-in. On the other hand, fan-in bounded by 2 is probably too restrictive. For example, a realistic situation could have many processors writing onto a bus; this would be, in effect, an OR with fan-in equal to the number of processors. Lower bounds on CRCW PRAM time are very strong since the model is so powerful. Second, it should be noted that Furst, Saxe, and Sipser were led to unbounded fan-in circuits by the desire to separate the polynomial-time hierarchy from PSPACE by an oracle. Starting from a quite different motivation, we were led to exactly the same type of circuits. This provides further evidence that lower bounds on the depth and size of unbounded fan-in circuits is an important area of study. Finally, a common thread running throughout the history of theoretical computer science is the search for the "right" computational models. One way to support a model or models as being right is to show equivalence of seemingly different models. This has occurred in attempts to model "effective computation" (Turing machines, recursion equations, RAM's, etc. are equivalent), "serial time" (Turing machine time, serial RAM time, and circuit size are equivalent to within a polynomial), and "parallel time" (vector machine time, SIMDAG time, alternating Turing machine time, and circuit depth are equivalent to within a polynomial). By the equivalence between CRCW PRAM's and circuits stated in Theorems 1 and 2, these two models support each other as being right models of unbounded fan-in parallelism.

Another model of parallelism is the alternating Turing machine [4], although when viewed as a model of parallelism the fan-in is bounded. Ruzzo and Tompa [22] have shown that if one considers the alternation depth of the alternating machine, that is, the number of times that the machine switches from an existential state to a universal state or vice versa along any computation path, then this complexity measure corresponds quite closely to parallel time in the unbounded fan-in sense. Specifically, if $T(n)$ and $S(n)$ are suitably well behaved functions with $S(n) \geq \log n$ and $\log T(n) \leq S(n) \leq T(n)$, then the class of languages accepted by alternating Turing machines that are simultaneously $O(T(n))$ alternation bounded and $O(S(n))$ space bounded is precisely the class of languages accepted by CRCW PRAM's that are simultaneously $O(T(n))$ time bounded and $2^{O(S(n))}$ processor bounded. An advantage of this result, as compared to Theorems 1 and 2, is that both models are uniform. A disadvantage is that the relationship does not hold for constant $T(n)$; the parity function is computable by a deterministic Turing machine in space $\log n$ (i.e., $T(n) \equiv 1$ and $S(n) = \log n$), but as noted in Corollary 1 above, parity is not computable in constant time by a CRCW PRAM

with a polynomial number of processors. In § 3, we review the definition of alternating Turing machines, state Ruzzo and Tompa's result more precisely, and outline the proof.

2. Proofs of Theorems 1 and 2.

Proof of Theorem 1. We give the proof for a nonuniform CRCW PRAM. We have made no special attempt to control the size of the constant c or the polynomial q in the statement of the theorem; we have opted instead for a simple description. Fix an input size n. Let P, T and S be the processor bound, running time and program size, respectively. We can take the word length, i.e., the maximum length of the binary representations of all addresses of registers and values stored in registers, to be

$$L = \max(n, S, \log P) + T + 1.$$

This is true because initially the length of all words in the memory or the program is $\max(n, S, \log P)$, and each instruction can at most double the magnitude of a value (i.e., add one to its length). One extra bit is added to L to allow for negative numbers. So that addition and subtraction can be treated uniformly, arithmetic is done modulo 2^{L+1} and a negative number $-z$ is stored as $(2^{L+1} - z) \bmod 2^{L+1}$.

In our circuit simulation we represent local and common memories by sets of triples

$$(a_\ell(p,k,t), v_\ell(p,k,t), w_\ell(p,k,t)) \text{ and } (a_c(k,t), v_c(k,t), w_c(k,t))$$

where $a_\ell(p,k,t)$, $v_\ell(p,k,t)$, $a_c(k,t)$ and $v_c(k,t)$ are L-bit binary words, and $w_\ell(p,k,t)$ and $w_c(k,t)$ are single bits. If $w_\ell(p,k,t) = 1$, the triple $(a_\ell(p,k,t), v_\ell(p,k,t), w_\ell(p,k,t))$ means that the register with address $a_\ell(p,k,t)$ in the local memory of processor p contains the word $v_\ell(p,k,t)$ at step t. If $w_\ell(p,k,t) = 0$, the triple means nothing. The triples $(a_c(k,t), v_c(k,t), w_c(k,t))$ mean the same for common memory. If for some a_0, p and t there is no k such that $w_\ell(p,k,t) = 1$ and $a_\ell(p,k,t) = a_0$, then local register a_0 of processor p contains zero at step t, and similarly for common memory triples. Since a processor can write into at most one local or common register at each step, we can take $1 \le k \le T$ for local triples and $1 \le k \le n+PT$ for common triples. Let $s(p)$ be the number of instructions in the program of processor p (certainly $s(p) \le S$). The circuit simulation also computes, for each processor p, each step t, and each j with $1 \le j \le s(p)$, a bit $\mathrm{ic}(p, j, t)$ (instruction counter) which is 1 iff processor p should execute the jth instruction of its program at step t. Let $x_1, \ldots, x_n$ be the input words stored in the first n locations of common memory, each padded out to L bits. At the start of the computation $(t = 0)$ set

$$(a_c(k,0), v_c(k,0), w_c(k,0)) = (k, x_k, 1) \quad \text{for } 1 \le k \le n,$$

$$\mathrm{ic}(p, j, 0) = \begin{cases} 1 & \text{if } j = 1 \\ 0 & \text{if } j \ne 1. \end{cases}$$

All other memory triples with $t = 0$ are set to (0,0,0).

The proof will be completed by showing that changes to the memory triples and ic bits caused by one step of the PRAM can be computed by a circuit of

constant depth and size polynomial in P, T, S and n. The inputs to this circuit are the local and common memory triples and the ic bits before the step and the outputs are the same information after the step. We restrict our description to a "projection" of the general circuit on one processor. The general outline of the circuit for one processor is shown in Fig. 1. We now describe the pieces of Fig. 1 and explain how each can be implemented by a constant-depth polynomial-size circuit. A subcircuit that is used often is the circuit EQ that takes two L-bit binary words, say y and z, and produces 1 (*true*) iff $y = z$. This circuit has depth 4 and size $O(L)$ since

$$\mathrm{EQ}(y, z) = \bigwedge_{i=1}^{L} (y_i z_i \vee (\sim y_i)(\sim z_i)).$$

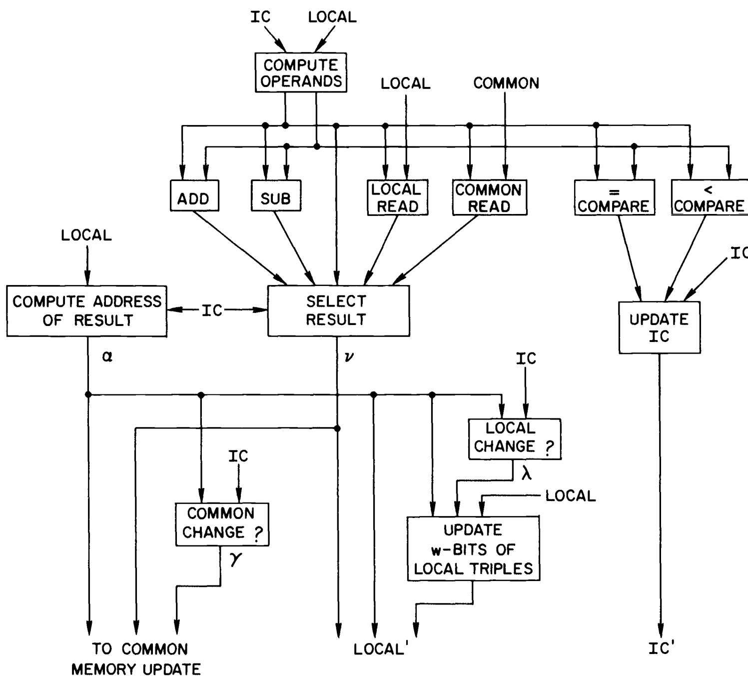


FIG. 1. *The high level outline of the circuit that simulates one step of one processor.*

Fix a processor p, $1 \le p \le P$, and a step t, $0 \le t < T$.

(1) Compute operands.

The outputs of this circuit are the values contained in local registers $M_{\mathrm{op}1}$ and $M_{\mathrm{op}2}$ for the instruction to be executed next. Let $\mathrm{op1}(p,j)$ be the L-bit binary representation of op1 in the jth instruction of the program of processor p; for an instruction with no op1 (e.g. HALT) we can take $\mathrm{op1}(p,j) = 0$. The first operand is computed as

$$\bigvee_{j=1}^{s(p)} \bigvee_{k=1}^{T} \mathrm{ic}(p,j,t) \wedge \mathrm{EQ}(\mathrm{op1}(p,j),\, a_\ell(p,k,t)) \wedge w_\ell(p,k,t) \wedge v_\ell(p,k,t).$$

The rightmost $\wedge$ of this expression is computed bitwise over the bits of $v_\ell(p,k,t)$. This expression can be implemented as a circuit of constant depth and size $O(STL)$. The second operand is computed similarly.

After the operands are computed, the various operations are applied to the operands.

(2) Addition and subtraction.

Addition of two L-bit numbers $y = y_L ... y_2 y_1$ and $z = z_L ... z_2 z_1$ can be computed by a circuit of size $O(L^3)$ and constant depth. First compute the carry *generate* and *propagate* bits as $g_i = y_i \wedge z_i$ and $p_i = y_i \vee z_i$, respectively. The ith carry bit c_i is 1 iff there is a $j < i$ such that $g_j = 1$ and $p_k = 1$ for all k with $j < k < i$. Therefore, all the carry bits can be computed by a circuit of size $O(L^3)$ and constant depth. Finally, the ith bit of the sum is $y_i \oplus z_i \oplus c_i$. For subtraction, $y - z$, the binary representation of $2^{L+1} - z \bmod 2^{L+1}$ can be computed by complementing the bits of z and then adding 1. (Recently, Chandra, Fortune and Lipton [3] have significantly improved the $O(L^3)$ upper bound for the constant depth computation of addition.)

(3) Read from memory.

Let a_0 be the first operand computed in (1) above. The value read from common memory is

$$\bigvee_{k=1}^{n+PT} \mathrm{EQ}(a_0,\ a_c(k,t)) \wedge w_c(k,t) \wedge v_c(k,t).$$

The size is $O(L(n+PT))$. Reading from local memory is similar; the size is $O(LT)$.

(4) Comparison.

For nonnegative numbers y and z (i.e., $y_L = z_L = 0$), $y < z$ iff there is an i such that $z_i = 1$ and $y_i = 0$ and $y_j = z_j$ for all $j > i$. This can be computed by a circuit of size $O(L^2)$ and constant depth. There are three other cases depending on whether y or z or both are negative. Testing whether $y = z$ is done by EQ.

(5) Update instruction counter.

An example should suffice. If part of the program is

5: GOTO 7 if $M_{\text{op1}} < M_{\text{op2}}$
6: not a GOTO,

if 7 is not mentioned in any other GOTO's, and if c is the output of the $<$-comparison circuit, then

$$\mathrm{ic}(p,7,t+1) = (\mathrm{ic}(p,5,t) \wedge c) \vee \mathrm{ic}(p,6,t).$$

The total size is $O(S)$.

(6) Select result.

Based on which $\mathrm{ic}(p,j,t)$ is 1, the correct L-bit result $v(p,t)$ is selected from the outputs of (1), (2) or (3). For instructions that load a constant or load the processor number, the proper constant is selected. The constant-depth implementation should be obvious at this point. The size is $O(LS)$.

(7) Compute address of result.

For instructions other than memory writes, the L-bit address $\alpha(p,t)$ is given by the number "res" in the instruction being executed at step t. For memory writes, the address is computed as in (1). The size is $O(STL)$.

(8) Local change? and common change?

The local (common) change bit $\lambda(p,t)$ ($\gamma(p,t)$) is 1 iff processor p changes the local (common) memory at step t. Let $C(p)$ be the set of instruction numbers of common memory WRITE instructions in the program of processor p. Let $\text{POS}(\alpha)$ be a circuit that computes 1 iff the L-bit α is positive. Recall the convention that an indirect WRITE using a nonpositive address has no effect. Then

$$\gamma(p,\,t) = \Big(\bigvee_{j\in C(p)} \text{ic}(p,\,j,\,t)\Big) \wedge \text{POS}(\alpha(p,\,t)).$$

$\lambda(p,t)$ is computed similarly. The size is $O(S+L)$.

(9) Update local memory.

Local memory of processor p is updated by setting

$$\begin{aligned} a_\ell(p,t+1,t+1) &= \alpha(p,t),\\ v_\ell(p,t+1,t+1) &= \nu(p,t),\\ w_\ell(p,t+1,t+1) &= \lambda(p,t), \end{aligned}$$

$$\begin{aligned} a_\ell(p,k,t+1) &= a_\ell(p,k,t) \quad \text{for } 1\le k\le t,\\ v_\ell(p,k,t+1) &= v_\ell(p,k,t) \quad \text{for } 1\le k\le t. \end{aligned}$$

Also, for each $1\le k\le t$, $w_\ell(p,k,t+1)$ should be set to 0 if processor p is changing its local memory at this step ($\lambda(p,t)=1$) and the address $a_\ell(p,k,t)$ matches the address $\alpha(p,t)$ being changed at this step. Each w_ℓ can be updated by a constant depth circuit of size $O(L)$, or $O(TL)$ in all.

The circuitry of (1)-(9) must be replicated for each processor, so this gives size $O(P(STL + L^3 + L(n + PT)))$.

(10) Update common memory.

Next, the common memory triples are updated using $\alpha(p,t)$, $\nu(p,t)$ and $\gamma(p,t)$ from all the processors. First, $\gamma(p,t)$ must be set to 0 if there is a processor q, with $q<p$, attempting to write into the same address.

$$\gamma'(p,\,t) = \gamma(p,\,t) \wedge \sim\Big(\bigvee_{q<p} \gamma(q,\,t) \wedge \text{EQ}(\alpha(q,\,t),\,\alpha(p,\,t))\Big).$$

The total size for updating the γ's is $O(P^2L)$. The updating of common memory triples is similar to (9) except that a new block of P triples comes into play. For $1\le p\le P$,

$$\begin{aligned} a_c(n+tP+p,t+1) &= \alpha(p,t),\\ v_c(n+tP+p,t+1) &= \nu(p,t),\\ w_c(n+tP+p,t+1) &= \gamma'(p,t). \end{aligned}$$

The computation of $w_c(k, t+1)$ for $k \leq n+tP$ is similar to (9). The size is $O(PL(n+PT))$.

Since all the circuitry must be replicated T times, the total size of the simulating circuit is $O(PT(STL + L^3 + L(n + PT)))$.

Finally, in order to assign output nodes in the circuit, the contents of the ith location of common memory, $1 \leq i \leq n$, is computed as in (3) where now a_0 is the L-bit representation of i. Output nodes are now assigned to the results of this final computation depending on the output conventions of the particular problem being solved by the CRCW PRAM. □

As mentioned in Remark 3 above, Theorem 1 remains true if the PRAM has instructions for multiplication and division of $O(\log n)$-bit numbers. In fact, *any* Boolean function with $d \cdot \log_2 n$ inputs and m outputs can be computed by a circuit of depth 3 and size $O(mn^{d+1})$ by computing each output from its disjunctive normal form.

Proof of Theorem 2. By DeMorgan's laws we can assume that the circuit contains only OR-gates and NOT-gates. Given a circuit with S wires and N nodes, number the nodes from 1 to N such that the n input nodes are numbered from 1 to n. Let C_1, C_2, ... denote the common memory registers. Initially the ith input bit is placed in C_i for $1 \leq i \leq n$. In general, the Boolean value computed by node number i in the circuit is computed by the CRCW PRAM in register C_i, and the values of all nodes at a fixed depth d in the circuit are computed in parallel. There is a processor associated with every wire. Say that a wire is directed from node i to node j, and node j is at depth d in the circuit. The processor associated with this wire waits cd steps, where the constant c is large enough that the values computed by all nodes at depth $< d$ have already been computed by the PRAM before step cd. The waiting is done by incrementing and testing a counter. If node j is an OR-gate, the processor writes 1 in C_j if C_i contains 1, or does nothing if C_i contains 0. If node j is a NOT-gate, the processor writes the negation of the contents of C_i into C_j. Other processors are assigned to output nodes. After waiting cT steps, these processors in parallel move the output bits to an initial contiguous block of common memory. □

Regarding the first sentence of the remark following the statement of Theorem 2, note that if several processors write into C_j at the same step, then j is an OR-gate and all these processors are writing 1.

3. Alternating Turing machines and CRCW PRAM's. In this section, some familiarity with the concept of alternation is assumed; see, for example, [4]. An alternating Turing machine (*ATM*) is like a nondeterministic Turing machine except that the states are partitioned into existential states, universal states, accepting states and rejecting states. Accepting and rejecting states are halting states. We consider ATM's with a read-only input tape and one read/write worktape. A configuration of an ATM consists of the state, the positions of the heads on the input tape and worktape, and the nonblank contents of the worktape. We let $\vdash$ denote the one-step transition relation on configurations. Let $\mathrm{INIT}_M(x)$ denote the initial configuration of machine M on input x. An ATM M *accepts* input x iff there is a tree, called an *accepting computation tree of M on input x*, such that the root of the tree is $\mathrm{INIT}_M(x)$, all leaves are accepting

configurations, if the existential configuration α is a node of the tree then α has one son β and $\alpha \vdash \beta$, and if the universal configuration α is a node of the tree then the sons of α are all those β with $\alpha \vdash \beta$. If L is a set of words, M *accepts* L if, for all x, M accepts x iff $x \in L$. With no loss of generality we assume that transitions from existential configurations to universal configurations, or vice versa, occur only when the machine is in one of a distinguished class of states called *switching states*. If β is an existential switching (resp., universal switching) configuration then there is exactly one α such that $\alpha \vdash \beta$ and α is a universal (resp., existential) configuration; moreover, β is the only configuration such that $\alpha \vdash \beta$. An ATM M is $S(n)$ *space bounded* if any configuration reachable from $\text{INIT}_M(x)$ uses at most $S(|x|)$ cells on the worktape. An ATM M is $T(n)$ *alternation bounded* if any computation path

$$\text{INIT}_M(x) \vdash \alpha_1 \vdash \alpha_2 \vdash \dots \vdash \alpha_m$$

has at most $T(|x|)-1$ switching configurations. We let

$$\text{ATM-ALT-SPACE}(T(n), S(n))$$

be the class of languages $L \subseteq \{1,2\}^*$ accepted by ATM's that are simultaneously $T(n)$ alternation bounded and $S(n)$ space bounded.

A CRCW PRAM M *accepts* input $x \in \{1,2\}^*$ iff, when started with the ith digit of x in common register i for $1 \le i \le |x|$, M halts with 1 in common register 1. Let

$$\text{CRCW-TIME-PROC}(T(n), P(n))$$

be the class of languages $L \subseteq \{1,2\}^*$ accepted by CRCW PRAM's that are simultaneously $T(n)$ time bounded and $P(n)$ processor bounded.

The pair $(T(n), S(n))$ is *suitable* if

(1) $S(n) \ge \log n$ and $\log T(n) \le S(n) \le T(n)$,

(2) there is a deterministic Turing machine which, when given an input of length n, lays off a block of $S(n)$ tape cells and computes the binary representation of $T(n)$, and

(3) there is a (serial, uniform cost criterion) RAM which, when given an input of length n, halts within $O(T(n))$ steps with $S(n)$ and $T(n)$ in two registers.

For example, letting $\log n$ abbreviate $\lceil \log_2(n+1) \rceil$, the function $\log n$ is computable by a RAM in time $O(\log n)$: starting with $I=1$, the RAM successively doubles the variable I until the Ith common location contains a zero. Therefore, $(\log n, \log n)$ is suitable.

THEOREM 3 (Ruzzo, Tompa [22]). *Let $(T(n), S(n))$ be suitable.*

$$\text{ATM-ALT-SPACE}(O(T(n)), O(S(n))) = \text{CRCW-TIME-PROC}(O(T(n)), 2^{O(S(n))}).$$

Proof. Our goal is just to outline enough of the proof to allow the interested reader to complete the details. (The following proof, which uses the results of the previous section as a foundation, is due to the first author. Ruzzo and Tompa's original proof does not use unbounded fan-in circuits *per se*.)

1. ($\subseteq$) Let M be an ATM which is $O(T(n))$ alternation bounded and $O(S(n))$ space bounded. If the constant d is such that $d^{S(n)}$ is an upper bound

on the number of configurations of M on inputs of length n, then M accepts within time $d^{S(n)}$ [4, Thm. 2.6]. Fix an input x. A *timed configuration* (on input x) is a pair (α, t) where α is an $S(|x|)$ space bounded configuration of M and t is an integer with $0 \leq t \leq d^{S(n)}$. The relation $\vdash$ is extended to timed configurations by

$$(\alpha, t) \vdash (\beta, t') \text{ iff } \alpha \vdash \beta \text{ and } t' = t + 1.$$

The CRCW PRAM that simulates M first constructs the adjacency matrix A of an acyclic directed graph whose nodes are the timed configurations. If u and v are timed configurations, there is an edge directed from v to u (i.e., $A(v,u) = 1$) iff $u \vdash v$. The matrix A is computed and stored in common memory. To compute A, each processor views its processor number as a pair (u,v) of timed configurations. Since u and v are strings of length $O(S(n))$, time $O(S(n))$ is sufficient for each processor to "decode" its number to a pair (u,v) and check whether $u \vdash v$; the processor then writes 1 in $A(v,u)$ iff $u \vdash v$. Note that $2^{O(S(n))}$ processors are sufficient for this, and recall that $S(n) \leq T(n)$.

A timed configuration (u, t) is *special* if u is halting, u is switching, or $(u, t) = (\text{INIT}_M(x), 0)$. The next step is to partially transitively close the graph so that $A(v,u) = 1$ iff there is a sequence $w_1, ..., w_m$ of *nonspecial* timed configurations such that

$$(*) \qquad u \vdash w_1 \vdash w_2 \vdash \dots \vdash w_m \vdash v.$$

To compute this closure, each processor decodes its processor number to a triple (u,w,v) of timed configurations and checks whether w is special; this takes time $O(S(n))$. If w is special, the processor does not participate in computing the closure. If w is not special, the processor executes a sequence of phases. During a given phase, the processor writes 1 in location $A(v,u)$ iff both $A(v,w) = 1$ and $A(w,u) = 1$; each phase takes constant time. After t phases, all paths (*) of length $\leq 2^t$ will have been discovered, so $\log_2 d^{S(n)} = O(S(n))$ phases suffice.

Next, it is easy to transform this graph to a circuit as follows. Remove all nonspecial nodes, view halting configurations as input nodes, view existential switching (resp., universal switching) configurations as OR-gates (resp., AND-gates), and view $(\text{INIT}_M(x), 0)$ as the single output node. This circuit has size $2^{O(S(n))}$ and depth $O(T(n))$. If input nodes corresponding to accepting (resp. rejecting) halting configurations are given value 1 (resp., 0), the value computed at the output node is 1 iff M accept x. As shown in the proof of Theorem 2, a CRCW PRAM with $2^{O(S(n))}$ processors can simulate this circuit within time $O(T(n))$.

2. ($\supseteq$). Let M be a CRCW PRAM which is $O(T(n))$ time bounded and $P(n) = 2^{O(S(n))}$ processor bounded. For a given input x, consider the circuit described in the proof of Theorem 1. By the well known trick of computing, for each gate, both the value computed by the gate and the negation of that value, we can assume that the circuit does not contain NOT-gates. Removing NOT-gates in this way does not increase depth and at most doubles size. This circuit has depth $O(T(n))$, and since it has size polynomial in P, T, and n, the nodes of the circuit can be named by binary strings of length

$$O(\log P(n) + \log T(n) + \log n) \quad \text{which is} \quad O(S(n)).$$

The construction of this circuit is sufficiently uniform that, for a particular naming of the nodes, there is an $O(S(n))$ space bounded deterministic Turing machine which, when given x and the names of nodes u and v, determines whether there is an edge (i.e., wire) of the circuit from u to v, and determines the types of u and v, that is, input node assigned value 0, input node assigned value 1, output node, AND-gate, or OR-gate. This Turing machine is used as a subroutine in the following ATM program for simulating the circuit.

```
v ← the output node of the circuit;
while v is not an input node do
    begin
        if v is an AND-gate (resp., OR-gate) then universally (resp., exis-
        tentially) choose a node u such that there is an edge from u to v in
        the circuit;
        v ← u;
    end
if the input node v is assigned value 0 (resp., 1) then reject (resp., accept).
```

This ATM is $O(T(n))$ alternation bounded and $O(S(n))$ space bounded, and it accepts x iff M does. □

Whereas Theorem 1 was useful in proving nonconstant lower bounds on CRCW PRAM time, Theorem 3 appears to be more useful in proving upper bounds. An interesting class is the class of languages accepted by CRCW PRAM's in time $O(\log n)$ with a polynomial number of processors. By Theorem 3, this class is precisely

$$\text{ATM-ALT-SPACE}(O(\log n), O(\log n)).$$

It is implicit in a paper of Ruzzo [21, Example 1, Thm. 2] that any context free language is in ATM-ALT-SPACE($O(\log n), O(\log n)$). The following corollary is immediate.

COROLLARY 2 (Ruzzo). *If L is context free then L is accepted in time* $O(\log n)$ *by a* CRCW PRAM *with a polynomial number of processors.*

It is interesting to compare this corollary with the result of Reif [18] that any *deterministic* context free language is accepted by a CREW PRAM (i.e., no concurrent writing to the same location) in time $O(\log n)$ using a polynomial number of processors.

Acknowledgment. We thank Larry Ruzzo and Martin Tompa for allowing us to include their result relating alternating Turing machines to CRCW PRAM's.

REFERENCES

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

[2] A. BORODIN, *On relating time and space to size and depth*, this Journal, 6 (1977), pp. 733-744.

[3] A. K. CHANDRA, S. FORTUNE, AND R. J. LIPTON, *Unbounded fan-in circuits and associative functions*, Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 52-60.

[4] A. K. CHANDRA, D. C. KOZEN, AND L. J. STOCKMEYER, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114-133.

[5] A. K. CHANDRA, L. J. STOCKMEYER, AND U. VISHKIN, *A complexity theory for unbounded fan-in parallelism*, Proc. 23rd IEEE Symposium on Foundations of Computer Science, 1982, pp. 1-13.

[6] A. K. CHANDRA, L. J. STOCKMEYER, AND U. VISHKIN, *Constant depth reducibility*, this Journal, this issue, pp. xxx-xxx.

[7] 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.

[8] P. W. DYMOND AND S. A. COOK, *Hardware complexity and parallel computation*, Proc. 21st IEEE Symposium on Foundations of Computer Science, 1980, pp. 360-372.

[9] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, Proc. Tenth ACM Symposium on Theory of Computing, 1978, pp. 114-118.

[10] M. FURST, J. B. SAXE, AND M. SIPSER, *Parity, circuits and the polynomial-time hierarchy*, Proc. 22nd IEEE Symposium on Foundations of Computer Science, 1981, pp. 260-270.

[11] L. M. GOLDSCHLAGER, *A universal interconnection pattern for parallel computers*, J. Assoc. Comput. Mach., 29 (1982), pp. 1073-1086.

[12] J.-W. HONG, *On similarity and duality of computation*, Proc. 21st IEEE Symposium on Foundations of Computer Science, 1980, pp. 348-359.

[13] G. LEV, *Size bounds and parallel algorithms for networks*, Doctoral Thesis, Report CST-8-80, Dept. of Computer Science, University of Edinburgh, 1980.

[14] G. LEV, N. PIPPENGER, AND L. G. VALIANT, *A fast parallel algorithm for routing in permutation networks*, IEEE Trans. on Computers, C-30 (1981), pp. 93-100.

[15] N. PIPPENGER, *On simultaneous resource bounds*, Proc. 20th IEEE Symposium on Foundations of Computer Science, 1979, pp. 307-311.

[16] N. PIPPENGER AND M. J. FISCHER, *Relations among complexity measures*, J. Assoc. Comput. Mach., 26 (1979), pp. 361-381.

[17] V. R. PRATT AND L. J. STOCKMEYER, *A characterization of the power of vector machines*, J. Comput. Sys. Sci., 12 (1976), pp. 198-221.

[18] J. REIF, *Parallel time* O(log *n*) *acceptance of deterministic CFLs*, Proc. 23rd IEEE Symposium on Foundations of Computer Science, 1982, pp. 290-296.

[19] R. REISCHUK, *A lower time bound for parallel random access machines without simultaneous writes*, Report RJ 3431, IBM Research Lab., San Jose, CA, 1982.

[20] W. L. RUZZO, *On uniform circuit complexity*, Proc. 20th IEEE Symposium on Foundations of Computer Science, 1979, pp. 312-318.

[21] W. L. RUZZO, *Tree-size bounded alternation*, J. Comput. Sys. Sci., 21 (1980), pp. 218-235.

[22] W. L. RUZZO AND M. TOMPA, personal communication.

[23] J. T. SCHWARTZ, *Ultracomputers*, ACM Trans. on Programming Languages and Systems, 2 (1980), pp. 484-521.

[24] Y. SHILOACH AND U. VISHKIN, *Finding the maximum, merging and sorting in a parallel computation model*, J. Algorithms, 2 (1981), pp. 88-102.

[25] M. SNIR, *On parallel searching*, Proc. SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Ottawa, Canada, 1982, pp. 242-253.

[26] U. VISHKIN, *Implementation of simultaneous memory address access in models that forbid it*, J. Algorithms, 4 (1983), pp. 45-50.

[27] U. VISHKIN, *A parallel-design space distributed-implementation space general purpose computer*, Report RC 9541, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1982.

[28] 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.

SIAM J. COMPUT.
Vol. 13, No. 2, May 1984
© 1984 Society for Industrial and Applied Mathematics
013

CONSTANT DEPTH REDUCIBILITY*

ASHOK K. CHANDRA†, LARRY STOCKMEYER‡ AND UZI VISHKIN§

Abstract. The purpose of this paper is to study reducibilities that can be computed by combinational logic networks of polynomial size and constant depth containing AND's, OR's and NOT's, with no bound placed on the fan-in of AND-gates and OR-gates. Two such reducibilities are defined, and reductions and equivalences among several common problems such as parity, sorting, integer multiplication, graph connectivity, bipartite matching and network flow are given. Certain problems are shown to be complete, with respect to these reducibilities, in the complexity classes deterministic logarithmic space, nondeterministic logarithmic space, and deterministic polynomial time. New upper bounds on the size-depth (unbounded fan-in) circuit complexity of symmetric Boolean functions are established.

Key words. circuit complexity, unbounded fan-in circuits, reducibility, complete problems

1. Introduction. Reducibility is a key concept in the theory of computation. In recursion theory, effective reducibility is useful in proving problems decidable or undecidable. In complexity theory, polynomial-time reducibility is central to the concept of NP-completeness [5], [14], [9] and permits one to show that the complexities of different problems are related even though the exact complexities of the problems are unknown. Similarly, reducibility can be used to establish upper or lower bounds on computational complexity: If problem A is reducible to problem B, then a lower bound on the complexity of A translates to a similar lower bound on the complexity of B, and an upper bound on B translates to an upper bound on A; this requires that the computational resources used in computing the reducibility function be negligible compared to the upper or lower bounds being proved. For investigating certain problems such as the relationship between deterministic and nondeterministic logarithmic space, polynomial-time reducibility is too weak so it has been strengthened to logspace reducibility [13], [21].

The purpose of this paper is to study reducibilities that can be computed by combinational logic circuits containing AND's, OR's and NOT's with no bound placed on the fan-in of AND-gates and OR-gates (call these simply *circuits*) where the circuit has polynomial size and constant depth. We define two such reducibilities and investigate how several common problems such as parity,

*Received by the editors September 14, 1982, and in revised form July 25, 1983. Portions of this paper have been reprinted, with permission, from "A complexity theory for unbounded fan-in parallelism" by A. K. Chandra, L. J. Stockmeyer and U. Vishkin, appearing in the Proceedings of the 23rd Annual Symposium on Foundations of Computer Science, 1982, pp.1-13, © 1982 IEEE. This paper was typeset at the IBM San Jose Research Laboratory.

†Mathematical Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

‡Computer Science Department, IBM Research Lab. K51/281, San Jose, California 95193. Work performed in part at the IBM Thomas J. Watson Research Center.

§Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, New York 10012. Work performed while the author was visiting the IBM Thomas J. Watson Research Center, while on leave from the Department of Computer Science, Technion, Haifa, Israel.

sorting, integer multiplication, graph connectivity, bipartite matching and network flow, are related by these reducibilities. There are two motivations for studying constant-depth reducibility. Interest was drawn to size-depth complexity for circuits by a recent result of Furst, Saxe and Sipser [8] showing that the parity function cannot be computed by any polynomial-size constant-depth circuit. Furst, Saxe and Sipser also give a few polynomial-size constant-depth reductions from parity to other problems, thus showing that these other problems cannot be computed by polynomial-size constant-depth circuits. By giving a more detailed classification of a larger collection of problems, our results contribute toward the theory of size-depth complexity for circuits. It should be expected that constant-depth reducibility will refine and expose more structure in low-level complexity classes such as deterministic polynomial time and nondeterministic logarithmic space than do the commonly used polynomial-time and log-space reducibilities.

Another motivation is that circuits with unbounded fan-in can be viewed as a model for unbounded fan-in parallelism, where circuit depth corresponds to parallel time and circuit size corresponds to the number of processors in the parallel machine. This view was strengthened by a recent result of Stockmeyer and Vishkin [22] giving a correspondence between circuits and the concurrent-read concurrent-write parallel random-access machines of Goldschlager [10] and Shiloach and Vishkin [19], [23] (*CRCW PRAM* or, for short, *WRAM*) which have many random access machines operating in parallel and communicating via a shared common memory, with appropriate conventions for resolving writing conflicts into the common memory. Stockmeyer and Vishkin show that time and number of processors for WRAM's correspond respectively (and simultaneously) to depth and size for circuits, where the time-depth correspondence is to within a constant factor and the size-processors correspondence is to within a polynomial. (Technically, for the simulation of circuits by WRAM's, the WRAM is allowed to be nonuniform.) By this correspondence, our results can also be viewed as contributing toward the theory of time-processors complexity for (nonuniform) WRAM's.

Section 2 of the paper contains definitions, including definitions of our constant-depth reducibilities. In §3, we establish several reductions and equivalences among problems using these reducibilities. We also show certain problems to be complete, with respect to these reducibilities, in the complexity classes deterministic logarithmic space, nondeterministic logarithmic space, and deterministic polynomial time. In §4 we give upper bounds on the size-depth circuit complexity of symmetric functions and graph transitive closure that improve the obvious upper bounds.

2. Definitions. A *Boolean function* is a function of the form $f:\{0,1\}^n \to \{0,1\}^m$ where 0 and 1 denote Boolean values *false* and *true*, respectively; n is the number of *variables* or *inputs*, and m is the number of *outputs*. A *problem* is an infinite sequence of Boolean functions $\{ f_n \mid n \geq 1 \}$ such that f_n has n variables. For some of the specific problems discussed herein, we define f_n only for certain values of n; in this case we assume that any f_n not defined explicitly is identically zero.

A *circuit* is an acyclic directed graph. Each node of the graph is labeled as either an input node, an AND-gate or an OR-gate. Input nodes must have fan-in (i.e., in-degree) zero. In addition, certain nodes are designated as output nodes.

An assignment of Boolean values to the input nodes extends, in the obvious way, to Boolean values associated with all nodes; an AND-gate (resp., OR-gate) with fan-in zero is assigned value 1 (resp., 0). The *size* of a circuit is the number of edges (i.e., wires). The *depth* is the length of a longest path from some input to some output. The circuit C *computes* the function $f:\{0,1\}^n \to \{0,1\}^m$ if C has $2n$ input nodes and m output nodes and there is a 1-1 correspondence between input nodes and $x_1, \overline{x}_1, \ldots, x_n, \overline{x}_n$ ($\overline{x}_i$ denotes the negation of x_i) and a 1-1 correspondence between output nodes and $y_1, \ldots, y_m$ such that, for all $(x_1, \ldots, x_n) \in \{0,1\}^n$, the assignment $x_1, \overline{x}_1, \ldots, x_n, \overline{x}_n$ to the input nodes extends to the assignment $y_1, \ldots, y_m$ to the output nodes where $f(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$. (When input variables and their negations are available as inputs, it is well known that NOT-gates can be eliminated from circuits without increasing depth and at most doubling size, so we do not need NOT-gates in our formal definition of circuits. However, for convenience we sometimes use negations in describing circuits.)

DEFINITION. Let $S(n)$ and $D(n)$ be functions from positive integers to positive reals. The problem $F = \{f_n\}$ is in the complexity class SIZE-DEPTH$(S(n), D(n))$ if, for every $n \geq 1$, f_n is computed by a circuit with both size $\leq S(n)$ and depth $\leq D(n)$. Also define

$$\text{SIZE-DEPTH(poly, constant)} = \bigcup_{c,d,k \geq 0} \text{SIZE-DEPTH}(cn^k, d).$$

Similarly, SIZE-DEPTH(poly, $D(n)$) is the union of SIZE-DEPTH$(cn^k, D(n))$ over all constants c and k, and SIZE-DEPTH$(S(n)$, constant) is the union of SIZE-DEPTH$(S(n), d)$ over all constants d.

A *fan-in 2 circuit* is a circuit such that all AND-gates and OR-gates have fan-in 2.

Our first reducibility is the projection reducibility studied by Skyum and Valiant [20]. A problem $F = \{f_n\}$ is *projection reducible* to a problem $G = \{g_n\}$ ($F \leq_{\text{proj}} G$) if there is a function $p(n)$ bounded above by a polynomial in n, and for each $f_n \in F$ a mapping

$$\sigma_n: \{ y_1, \ldots, y_{p(n)} \} \to \{ x_1, \overline{x}_1, \ldots, x_n, \overline{x}_n, 0, 1 \}$$

such that $f_n(x_1, \ldots, x_n) = g_{p(n)}(\sigma_n(y_1), \ldots, \sigma_n(y_{p(n)}))$. We also use a weaker reducibility, *constant-depth truth-table reducibility*: $F \leq_{\text{cd-tt}} G$ if there is a polynomial $p(n)$ and a constant c such that each f_n is computed by a circuit of depth $\leq c$ and size $\leq p(n)$ containing "black boxes" which compute members g_j of G or their negations $\overline{g}_j$ with $j \leq p(n)$, where the size and depth of black boxes are counted as unity, and such that there is no path in the circuit from an output of one black box to an input of another black box.

We write $F \equiv_x G$ if both $F \leq_x G$ and $G \leq_x F$.

The following proposition is obvious.

PROPOSITION 2.1. (1) $\leq_{\text{proj}}$ *and* $\leq_{\text{cd-tt}}$ *are transitive relations.*

(2) $F \leq_{\text{proj}} G \Rightarrow F \leq_{\text{cd-tt}} G$.

(3) *If* $F \leq_{\text{proj}} G$ *or* $F \leq_{\text{cd-tt}} G$, *and* $G \in$ SIZE-DEPTH$(S(n), D(n))$ *where* S *and* D *are monotone nondecreasing, then*

$$F \in \text{SIZE-DEPTH}(p(n) \cdot S(p(n)), c \cdot D(p(n)))$$

for some polynomial p *and constant* c. *In particular,*

$$G \in \text{SIZE-DEPTH(poly, constant)} \Rightarrow F \in \text{SIZE-DEPTH(poly, constant)}.$$

(We are interested primarily in the case $D(n) = O((\log n)^k)$ for some constant k, and $c \cdot D(p(n)) = O((\log n)^k)$ in this case, justifying the name "constant depth reducibility".)

We use certain complexity classes defined by time or space bounded Turing machines (see, for example, [9], [12]). Let $\mathscr{L}$ (resp., $\mathscr{NL}$) be the class of languages $L \subseteq \{0,1\}^*$ which are accepted by deterministic (resp., nondeterministic) Turing machines with a read-only input tape and a $\log n$ bounded worktape. A language $L \subseteq \{0,1\}^*$ is in the class NONUNIF-$\mathscr{L}$ (NONUNIF-$\mathscr{NL}$) if there is a polynomial $p(n)$ and a $\log n$ space-bounded deterministic (nondeterministic) Turing machine M, such that for every n there is a binary word α_n with $|\alpha_n| \le p(n)$, such that for all x with $|x| = n$, M accepts $x\#\alpha_n$ iff $x \in L$; α_n is called the *advice* (for inputs of length n). Let $\mathscr{P}$ be the class of languages $L \subseteq \{0,1\}^*$ accepted by polynomial time-bounded deterministic Turing machines. A problem F is a *one-output problem* if every $f_n \in F$ has one output. The complexity classes $\mathscr{L}$, $\mathscr{NL}$, NONUNIF-$\mathscr{L}$, etc. can be viewed as classes of one-output problems by making the obvious correspondence to binary languages, namely, $x \in L$ iff $f_{|x|}(x) = 1$. If $\le$ is a reducibility, F is a one-output problem, and $\mathscr{C}$ is a class of binary languages, then $\mathscr{C} \le F$ if $G \le F$ for all $G \in \mathscr{C}$; F is $\le$*-complete in* $\mathscr{C}$ if both $\mathscr{C} \le F$ and $F \in \mathscr{C}$.

For several of the specific problems considered, inputs are natural numbers or graphs. Natural numbers are sometimes represented in unary notation and sometimes in binary. The *m-bit unary representation* of k $(0 \le k \le m)$ is the binary word $1^k 0^{m-k}$. Graphs are represented by adjacency matrices. An undirected graph with m vertices $v_1, \ldots, v_m$ is represented by $m(m-1)/2$ Boolean variables a_{ij} for $1 \le i,j \le m$ and $i < j$ such that $a_{ij} = 1$ iff there is an edge between v_i and v_j. A directed graph is represented by m^2 variables a_{ij} for $1 \le i,j \le m$ such that $a_{ij} = 1$ iff there is a directed edge from v_i to v_j. An undirected bipartite graph with $2m$ vertices $u_1, \ldots, u_m, v_1, \ldots, v_m$ is represented by m^2 variables a_{ij} for $1 \le i,j \le m$ such that $a_{ij} = 1$ iff there is an edge between u_i and v_j. In describing graph constructions we let $\{u,v\}$ denote the undirected edge between vertices u and v and we let (u,v) denote the directed edge from u to v.

Define the function

$$\lg(n) = \lceil \log_2(n+1) \rceil;$$

note that $\lg(n)$ is the length of the binary representation of n.

3. Constant depth reducibility among problems. Our results are summarized in Fig. 1; see the Appendix for precise definitions of the various problems.

THEOREM 3.1. *All the problems enclosed as a group in Fig.* 1 *are in the same* $\equiv_{\text{cd-tt}}$ *equivalence class. Groups with* $\equiv_{\text{proj}}$ *written to the left are in the same* $\equiv_{\text{proj}}$ *equivalence class. A problem in a group* $\mathscr{G}$ *is reducible to all problems in the group above* $\mathscr{G}$ *via the indicated reducibility. Problems in the lowest group in Fig.* 1 *are in the class* SIZE-DEPTH(poly, constant).

(We note that Furst, Saxe and Sipser [8] previously showed that PARITY is $\le_{\text{cd-tt}}$ to THRESHOLD, MULTIPLICATION, and UNDIR-ST-CONNECTIVITY.) Problems that are $\equiv_{\text{proj}}$ are, in a fairly strong sense, the "same" problem. Since Furst, Saxe and Sipser [8] show that PARITY is not in the class SIZE-DEPTH(poly, constant), we know that this class lies strictly below the class containing PARITY in the $\le_{\text{cd-tt}}$ ordering. We do not know that any of the

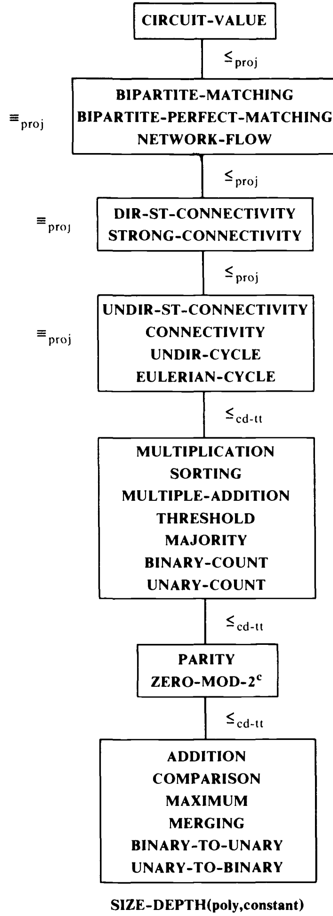


FIG. 1.

other relationships in Fig. 1 are strict in the $\leq_{cd\text{-}tt}$ ordering. Several of the reductions in Fig. 1 are not surprising. However, we can single out

$$\text{UNDIR-ST-CONNECTIVITY} \equiv_{proj} \text{CONNECTIVITY}$$
$$\text{MULTIPLICATION} \equiv_{cd\text{-}tt} \text{SORTING}$$

as being nonobvious.

Shortly we prove Theorem 3.1 by giving a series of reductions. We first prove the following completeness results; these results immediately yield most of the projection-equivalences for graph connectivity problems.

THEOREM 3.2.

(1) UNDIR-ST-CONNECTIVITY, CONNECTIVITY, *and* UNDIR-CYCLE *are* $\leq_{proj}$*-complete in* NONUNIF-$\mathscr{L}$.

(2) UNDIR-CYCLE *is* $\leq_{proj}$*-complete in* $\mathscr{L}$.

(3) DIR-ST-CONNECTIVITY *and* STRONG-CONNECTIVITY *are* $\leq_{proj}$*-complete in* NONUNIF-$\mathscr{NL}$ *and in* $\mathscr{NL}$.

(4) CIRCUIT-VALUE *is* $\leq_{proj}$*-complete in* $\mathscr{P}$.

Proof. The reductions are very similar to the known log-space reductions [11], [13], [15], [17].

(1) By a result of Aleliunas et al. [1], it easily follows that all three problems mentioned in part (1) belong to NONUNIF-$\mathscr{L}$; for inputs of length $n = m(m-1)/2$ corresponding to adjacency matrices for undirected graphs with m vertices, the advice α_n encodes a universal traversal sequence for such graphs. For the other direction we first show that

$$\text{NONUNIF-}\mathscr{L} \leq_{\text{proj}} \text{UNDIR-ST-CONNECTIVITY}.$$

Let M be a deterministic $\lg(n)$ space-bounded Turing machine. Fix an input length n, let α_n be the advice, and let $n' = |\alpha_n|$. The *configurations* of M are 4-tuples (q, i, β, j) where q indicates the state of the finite control, i with $0 \leq i \leq n+n'+2$ is the position of the input head in the string $¢x\#\alpha_n\$$ (where ¢ and \$ are left and right endmarkers), the word β with $|\beta| = \lg(n)$ is the content of the worktape, and j is the position of the head on the worktape. The machine has a unique initial configuration $c_0 = (q_0, 0, \#^{\lg(n)}, 1)$ where q_0 is the unique initial state and # is the blank tape symbol. Assume that M has a unique accepting configuration $c_a = (q_a, 0, \#^{\lg(n)}, 1)$ where q_a is the unique accepting state. Modify M so that it never halts, and that if configuration c_a is entered then M cycles forever in c_a. Let p be the number of configurations of M; p is bounded above by a polynomial in n. The number p can also be taken as an upper bound on the time for M to accept; i.e., if M accepts x of length n then M will be in configuration c_a at step p.

For each input x of length n we describe a graph $G(x)$ such that the adjacency matrix of $G(x)$ is obtained from x by a projection reduction. The vertices of $G(x)$ are all pairs (c, t) where c is a configuration of M and $0 \leq t \leq p$. Consider a vertex (c, t) with $c = (q, i, \beta, j)$ and $0 \leq t < p$. If $i = 0$ or $i > n$, then the configuration c' reached in one move of M is determined and $G(x)$ will have the edge $\{(c,t),(c',t+1)\}$. If $1 \leq i \leq n$, there will be two configurations c' and c'' such that c' (c'') is the configuration reached in one move if $x_i = 0$ ($x_i = 1$). Thus, $\overline{x}_i$ is the entry in the adjacency matrix corresponding to the edge $\{(c,t),(c',t+1)\}$, and x_i is the entry corresponding to the edge $\{(c,t),(c'',t+1)\}$. If $c' = c''$ then the entry corresponding to edge $\{(c,t),(c',t+1)\}$ is 1. Let $m = p(p+1)$ be the number of vertices of $G(x)$. Number the vertices so that $v_1 = (c_0, 0)$ and $v_m = (c_a, p)$. Since M is deterministic, for each fixed x and each vertex v of the form $(\cdot, p)$, the connected component of $G(x)$ containing v is a tree which can be rooted at v such that the sons of $(\cdot, t)$ in the tree are all of the form $(\cdot, t-1)$ for all t. Therefore, M accepts x iff there is a path from $v_1 = (c_0, 0)$ to $v_m = (c_a, p)$.

For CONNECTIVITY, we form $G'(x)$ from $G(x)$ by adding the edges of a spanning tree among the vertices

$$\{\ (c,p)\ |\ c \neq c_a\ \} \cup \{(c_0, 0)\}.$$

Recalling that M never halts, it is easy to see that $G'(x)$ is connected iff there is a path from $(c_0, 0)$ to (c_a, p). For UNDIR-CYCLE, we add to $G(x)$ the edge between $(c_0, 0)$ and (c_a, p).

(2) Since Hong [11] shows that UNDIR-CYCLE $\in \mathscr{L}$, the result is immediate from part (1).

(3) Obviously DIR-ST-CONNECTIVITY and STRONG-CONNECTIVITY belong to $\mathscr{NL}$. The proof that

$$\text{NONUNIF-}\mathcal{NL} \leq_{\text{proj}} \text{DIR-ST-CONNECTIVITY}$$

is very similar to the undirected case in (1). For example, if from configuration $c = (q, i, \beta, j)$ with $x_i = 0$ ($x_i = 1$) M can reach any configuration in the set C_0 (C_1) in one move, and if $B = C_0 \cap C_1$, then the entry of the adjacency matrix corresponding to the directed edge $((c,t),(c',t+1))$ is $\bar{x}_i$ if $c' \in C_0 - B$, x_i if $c' \in C_1 - B$, 1 if $c' \in B$, or 0 otherwise. So there is a directed path in $G(x)$ from $v_1 = (c_0, 0)$ to $v_m = (c_a, p)$ iff M accepts x.

For STRONG-CONNECTIVITY, form $G'(x)$ from $G(x)$ by adding the directed edges

$$\{ (v_m, u), (u, v_1) \mid u \neq v_1 \text{ and } u \neq v_m \}.$$

Then $G'(x)$ is strongly connected iff there is a directed path from v_1 to v_m in $G(x)$.

(4) As Ladner [15] points out, it is well known that for any deterministic Turing machine M there is a polynomial-size fan-in 2 circuit which takes as input a (binary representation of a) configuration of M and outputs the next configuration after one step. If M is $p(n)$ time-bounded, one places $p(n)$ copies of this circuit in series. By choosing the binary representation of tape symbols properly, the bits of the binary representation of the initial configuration are either constants or depend directly on the bits of the input x to M. Therefore, this is a projection reduction. □

The following useful lemma is proved by computing f from its disjunctive normal form.

LEMMA 3.3. *If f is a Boolean function with n inputs and m outputs, then f is computed by a circuit of size $\leq (m+n) \cdot 2^n$ and depth* 2.

Proof of Theorem 3.1. We first show that the problems in the lowest group in Fig. 1 are in SIZE-DEPTH(poly, constant).

ADDITION

The well known carry look-ahead method for the addition of m-bit numbers can be implemented as a circuit of size $O(m^3)$ and constant depth (see, e.g., the proof of Theorem 1 in [22]). This size bound is greatly improved in [4].

COMPARISON

Given two m-bit binary numbers $y = y_m \ldots y_2 y_1$ and $z = z_m \ldots z_2 z_1$, $y < z$ iff there is an i such that $y_i = 0$, $z_i = 1$, and $y_j = z_j$ for all $j > i$. This can be computed by a circuit of size $O(m^2)$ and depth $O(1)$.

MAXIMUM

Given m m-bit binary numbers $a_1, \ldots, a_m$, let $c_{ij} = 1$ if $a_i \geq a_j$, or 0 otherwise. Using COMPARISON, all the c_{ij} can be computed in size $O(m^4)$ and depth $O(1)$. Letting d_i be the AND of c_{ij} over all $j \neq i$, a_i is the maximum iff $d_i = 1$. The maximum number can be computed as

$$\bigvee_{1 \leq i \leq m} a_i \wedge d_i$$

where the $\wedge$ is computed bitwise over the bits of a_i.

MERGING

Let $a_1, ..., a_m$ and $b_1, ..., b_m$ be the lists of m-bit binary numbers, each list sorted in nondecreasing order. (Using COMPARISON, a polynomial-size constant-depth circuit can check that the lists are indeed sorted and can set all bits of the output to zero if not.) Let $c_{ij} = 1$ iff $b_i < a_j$. For each j, $c_{1j}c_{2j}...c_{mj}$ is a unary representation of the number of b's less than a_j. Let c_j be this unary representation with j 1's appended on the left and $m-j$ 0's appended on the right, so c_j is a $2m$-bit unary representation of a_j's position in the merged list. Similarly, let $d_{ij} = 1$ iff $a_i \leq b_j$. Appending j 1's to the left of $d_{1j}d_{2j}...d_{mj}$ and $m-j$ 0's to the right gives b_j's position in the merged list as a unary number d_j. For each k with $1 \leq k \leq 2m$, let $\mathrm{EQ}_k(z)$ be a one-output circuit whose output is 1 iff z is a unary representation of k; letting $z = z_1z_2...z_{2m}$ and $z_{2m+1} = 0$, $\mathrm{EQ}_k(z) = z_k \wedge \bar{z}_{k+1}$. Finally, the kth number in the merged list is

$$\bigvee_{1 \leq j \leq 2m} (EQ_k(c_j) \wedge a_j) \vee \bigvee_{1 \leq j \leq 2m} (EQ_k(d_j) \wedge b_j).$$

Remark. Shiloach and Vishkin [19] show that MAXIMUM and MERGING can be computed by a WRAM in constant time with a polynomial number of processors. Therefore, an alternate proof that MAXIMUM and MERGING are in SIZE-DEPTH(poly, constant) follows from this result and the Stockmeyer-Vishkin [22] simulation of WRAM's by circuits mentioned in the Introduction.

UNARY-TO-BINARY

Let $x = x_1...x_n$ be the given unary representation. (A polynomial-size constant-depth circuit can check whether $x_1...x_n \in 1^*0^*$ and set the output to zero if not.) Let $x_0 = 1$, $x_{n+1} = 0$ and $d_i = x_i \wedge \bar{x}_{i+1}$ for $0 \leq i \leq n$, so $d_i = 1$ iff x is the unary representation of i. For $1 \leq j \leq \lg(n)$, the jth bit of the binary representation is the OR of d_i over all i such that the jth bit of i is 1.

BINARY-TO-UNARY

Since the output depends only on the first $\lg(n)$ inputs, the polynomial-size constant-depth circuit is immediate from Lemma 3.3.

The remainder of the proof of Theorem 3.1 is a series of reductions.

PARITY $\leq_{\text{proj}}$ ZERO-MOD-2^c

PARITY on n variables $x_1, ..., x_n$ is reduced to ZERO-MOD-2^c on $2^{c-1}n$ variables y_{ij} for $1 \leq i \leq n$ and $1 \leq j \leq 2^{c-1}$. The projection reduction is $\sigma_n(y_{ij}) = x_i$ for all j.

ZERO-MOD-2^c $\leq_{\text{cd-tt}}$ PARITY

We show that ZERO-MOD-2^c $\leq_{\text{cd-tt}}$ ZERO-MOD-2^{c-1} for all $c \geq 2$. This suffices since c is constant and $\leq_{\text{cd-tt}}$ is transitive. Let $x_1, ..., x_n$ be the input bits and let $s = \Sigma x_i$. Compute $y_{ij} = x_i \wedge x_j$ for all i and j with $i < j$. Let $t = \Sigma y_{ij}$ and note that $t = s(s-1)/2$. It is easy to verify that $s \equiv 0 \pmod{2^c}$ iff both $s \equiv 0 \pmod{2^{c-1}}$ and $t \equiv 0 \pmod{2^{c-1}}$.

PARITY $\leq_{\text{proj}}$ MULTIPLICATION

This reduction is given in [8]. Given $x_1, ..., x_n$, let $k = \lg(n)$ and form the two binary numbers

$$A = \sum_{i=1}^{n} x_i 2^{k(i-1)}, \quad B = \sum_{i=1}^{n} 2^{k(i-1)}.$$

Writing $AB = \Sigma\, c_i 2^{k(i-1)}$, where the c_i are k-bit numbers, the parity of $\Sigma\, x_i$ is the low order bit of c_n.

The next seven reductions form a cycle showing $\equiv_{\text{cd-tt}}$ for the seven problems in the group containing MULTIPLICATION in Fig. 1.

MAJORITY $\leq_{\text{cd-tt}}$ MULTIPLICATION

The first stage of this reduction is identical to PARITY $\leq_{\text{proj}}$ MULTIPLICATION described above. The binary representation of $\Sigma\, x_i$ is c_n. These k bits are then compared with the k-bit binary representation of $\lceil n/2 \rceil$.

MULTIPLICATION $\leq_{\text{cd-tt}}$ MULTIPLE-ADDITION

Multiplication of m-bit numbers y and $z = \Sigma\, z_i 2^i$ is reduced to addition of m $2m$-bit numbers $a_0, \ldots, a_{m-1}$. For each i, $a_i = z_i 2^i y$, so each bit of each a_i is the AND of a bit of y and a bit of z.

MULTIPLE-ADDITION $\leq_{\text{cd-tt}}$ BINARY-COUNT

Let $A_1, \ldots, A_m$ be the m m-bit binary numbers to be added, and for $1 \leq i \leq m$ let

$$A_i = \sum_{j=0}^{m-1} a_{ij} 2^j \quad \text{where } a_{ij} \in \{0,1\}.$$

Let $L = m + \lg(m)$ and let $\ell = \lg(m)$. Note that L bits are sufficient for the sum $S = \Sigma\, A_i$. We describe the reduction as a sequence of stages.

The first stage. For $0 \leq j < m$, let b_j be the ℓ-bit result of applying BINARY-COUNT with inputs $a_{1j}, a_{2j}, \ldots, a_{mj}$. Note that $S = \Sigma\, b_j 2^j$. Since each b_j has only ℓ bits, the numbers $b_j 2^j$ can be "packed" into ℓ L-bit numbers $B_1, \ldots, B_\ell$ whose sum is S; that is for each i with $1 \leq i \leq \ell$ let

$$B_i = \sum_{k \geq 0} b_{i-1+k\ell} 2^{i-1+k\ell}.$$

The second stage. Let $\ell(2) = \lg(\ell)$ $(= \lg(\lg(m)))$. Similarly, by applying BINARY-COUNT to each bit position in $B_1, \ldots, B_\ell$ and then packing the $\ell(2)$-bit results into $\ell(2)$ L-bit numbers $C_1, \ldots, C_{\ell(2)}$, we have reduced the original problem to the problem of adding $\ell(2)$ L-bit numbers. Since each use of BINARY-COUNT in the second stage has only $\lg(m)$ inputs, each application of BINARY-COUNT is implemented directly by a circuit of polynomial size and constant depth (see Lemma 3.3) rather than by a "black box" as in the first stage.

The remaining stages. For $i \geq 3$, let $\ell(i) = \lg(\ell(i-1))$. By the same method of counting followed by packing, the ith stage produces $\ell(i)$ L-bit numbers whose sum is S. Let i^* be the smallest i such that $\ell(i) = 2$. The i^* stage produces two L-bit numbers, say Y and Z, which can be added directly by a circuit of polynomial size and constant depth to obtain S. If implemented as described, this would give a circuit of depth $> i^*$ (not constant). Actually, only the first two stages are implemented as described. For $i \geq 3$, note that each bit of each L-bit number produced at the ith stage depends on at most $\ell(i-1)$ bits of the numbers produced at the $(i-1)^{\text{th}}$ stage. Therefore, each bit of Y and Z depends on at most $t = \ell(2)\ell(3)\ldots\ell(i^*-1)$ bits of the numbers $C_1, \ldots, C_{\ell(2)}$ produced at the

second stage. Since $t = O(\lg n)$, Lemma 3.3 says that Y and Z can be computed from the C_i by a circuit of polynomial size and constant depth.

BINARY-COUNT $\leq_{\text{cd-tt}}$ SORTING

Let $x_1, ..., x_n$ be the input bits. Let a_i be an n-bit number whose highest order bit is x_i (the lower order bits of the a_i can be chosen to make the a_i distinct if desired). The highest order bits of the a_i in sorted order give an n-bit unary representation of Σx_i, from which the binary representation of Σx_i can be computed in polynomial size and constant depth.

SORTING $\leq_{\text{cd-tt}}$ UNARY-COUNT

Let $a_1, ..., a_m$ be the numbers to be sorted. Let $c_{ij} = 1$ iff either $a_i < a_j$ or ($a_i = a_j$ and $i \leq j$). For each fixed j, applying UNARY COUNT with inputs c_{ij} for all i gives a unary representation of a_j's position in the sorted list. The sorted list is then computed as in the proof that MERGING is in SIZE-DEPTH(poly, constant).

UNARY-COUNT $\leq_{\text{cd-tt}}$ THRESHOLD

Given $x_1, ..., x_n$, the ith bit of the unary representation of Σx_i is the result of applying THRESHOLD to $x_1, ..., x_n$ with threshold i.

THRESHOLD $\leq_{\text{cd-tt}}$ MAJORITY

Let $x_1, ..., x_m$ be the input bits to the threshold function and let $k_1 ... k_m$ be the m-bit unary representation of the threshold k. Let $y_i = x_i$ and $y_{m+i} = \overline{k}_i$ for $1 \leq i \leq m$. Now

$$\sum_{i=1}^{m} x_i \geq k \quad \text{iff} \quad \sum_{i=1}^{2m} y_i \geq m.$$

This completes the cycle for the $\equiv_{\text{cd-tt}}$-class containing MULTIPLICATION.

MAJORITY $\leq_{\text{proj}}$ CONNECTIVITY

Since MAJORITY $\in \mathscr{L}$, this is immediate from Theorem 3.2(1).

UNDIR-ST-CONNECTIVITY $\equiv_{\text{proj}}$ CONNECTIVITY
UNDIR-ST-CONNECTIVITY $\equiv_{\text{proj}}$ UNDIR-CYCLE

These are immediate from Theorem 3.2(1).

EULERIAN CYCLE $\leq_{\text{proj}}$ CONNECTIVITY

An undirected graph G has an Eulerian cycle iff G is connected except for isolated vertices and every vertex of G has even degree [2]. Since [1] shows that the connectivity test is in NONUNIF-$\mathscr{L}$, it follows that EULERIAN-CYCLE is in NONUNIF-$\mathscr{L}$, so the reduction follows from Theorem 3.2(1).

CONNECTIVITY $\leq_{\text{proj}}$ EULERIAN-CYCLE

Given an undirected graph G, we describe an undirected graph G' such that G is connected iff G' has an Eulerian cycle. If G has vertices $V = \{v_1, ..., v_m\}$, then G' has vertices

$$V \cup \{ u_{ij} \mid 1 \leq i < j \leq m \} \cup \{ y_i, z_i \mid 1 \leq i \leq m \}.$$

The edges $\{v_i, y_i\}$, $\{v_i, z_i\}$ and $\{y_i, z_i\}$ are in G' for all i. The edges $\{v_i, u_{ij}\}$, $\{u_{ij}, v_j\}$ and $\{v_i, v_j\}$ are present in G' iff the edge $\{v_i, v_j\}$ is present in G. Note that every vertex of G' has even degree, and G' is connected (except for isolated vertices) iff G is connected.

UNDIR-ST-CONNECTIVITY $\leq_{\text{proj}}$ DIR-ST-CONNECTIVITY
DIR-ST-CONNECTIVITY $\equiv_{\text{proj}}$ STRONG-CONNECTIVITY

These are immediate from Theorem 3.2(3).

DIR-ST-CONNECTIVITY $\leq_{\text{proj}}$ NETWORK-FLOW

There is a directed path from vertex v_1 to vertex v_m in G iff there is a flow of ≥ 1 from v_1 to v_m when all edges of G have unit capacity.

The next three reductions form a cycle.

BIPARTITE-PERFECT-MATCHING $\leq_{\text{proj}}$ BIPARTITE-MATCHING

This is trivial.

BIPARTITE-MATCHING $\leq_{\text{proj}}$ NETWORK-FLOW

This reduction is given in [6, Thm. 6.11]. Let G be the given bipartite graph with vertices $u_1, \ldots, u_m, v_1, \ldots, v_m$. Form a directed flow network G' with vertices $s, t, u_1, \ldots, u_m, v_1, \ldots, v_m$. The directed edges (s, u_i) and (v_i, t) in G' have capacity 1 for all $1 \leq i \leq m$. The directed edge (u_i, v_j) in G' has capacity 1 (resp., 0) if the edge $\{u_i, v_j\}$ is present (resp., not present) in G. All other edges of G' have capacity 0. There is a flow of f from s to t iff G has a matching of size f.

NETWORK-FLOW $\leq_{\text{proj}}$ BIPARTITE-PERFECT-MATCHING

Many of the details of this reduction were pointed out to us by N. Pippenger; the reduction was observed independently by T. Feather. A few additional details needed to make the reduction a projection are due to the authors. Let G be the given directed flow network with vertices $v_1, \ldots, v_m$. Assume for the moment that the edge capacities $c(i,j)$ and the flow f are valid unary representations. It is convenient to describe the reduction as a composition of two projection reductions. In the first reduction, we transform G to a directed graph G' with $2m^3-m^2$ vertices. For each vertex v_i of G, G' has m^2 "copies" of v_i, namely v_{ip} for $1 \leq p \leq m^2$. For all $1 \leq p \leq m^2$, the vertices v_{1p} are called *sources* of G' and the vertices v_{mp} are called *targets* of G'. For each pair (i,j) with $1 \leq i,j \leq m$ and $i \neq j$, G' also has vertices e_{ijk} for $1 \leq k \leq m$. For each i, j, k, p and q with $1 \leq i,j,k \leq m$ and $1 \leq p,q \leq m^2$, the directed edges (v_{ip}, e_{ijk}) and (e_{ijk}, v_{jq}) are present in G' iff $k \leq c(i,j)$. Since $c(i,j)$ is represented in unary, this is a projection reduction; that is, the entries of the adjacency matrix corresponding to the edges (v_{ip}, e_{ijk}) and (e_{ijk}, v_{jq}) are simply the kth bit of $c(i,j)$. There is a flow of f from v_1 to v_m in G iff

(*) There are f vertex disjoint paths in G', each path being from some source to some target, with the sources and targets of the paths also being disjoint.

In the second step we transform G' to an undirected bipartite graph G''. Replace each source u of G' by a vertex u_b. Replace each target u of G' by a vertex u_g. Replace each other vertex u of G' by the pair of vertices u_b and u_g with an edge between u_b and u_g. The edge $\{u_b, w_g\}$ is present in G'' iff the directed edge (u, w) is present in G'. Let $f_1 f_2 \ldots f_r$ be the unary representation of the flow f, where $r = m^2$. If u_b is the replacement for the source v_{1p} and w_g is the replacement for the target v_{mp}, then the edge $\{u_b, w_g\}$ is present in G'' iff $f_p = 0$. Thus, for each p with $1 \leq p \leq f$, the replacement u_b for v_{1p} cannot be matched with the replacement w_g for v_{mp}, but for $f < p \leq m^2$, u_b can be matched with w_g.

It is not difficult to verify that (*) holds in G' iff there is a perfect matching in G''. The correspondence between vertex disjoint directed paths in G' and perfect matchings in G'' is as follows: If (u, w) is not a source-target pair, the edge (u, w) of G' is used in one of the directed paths iff the edge $\{u_b, w_g\}$ of G'' is used in the matching; if the vertex u of G' is neither a source nor a target, then u does not appear on any of the directed paths iff the edge $\{u_b, u_g\}$ is used in the matching. Details are left to the reader.

Finally, for each word $c(i,j)$ and f we add a new component to G'' such that the added components all have perfect matchings iff all $c(i,j)$ and f are valid unary representations. In general, let $w = w_1...w_m$ be a binary word. We describe a bipartite graph $H(w)$ with vertices $a_1, ..., a_{m+1}$, $b_1, ..., b_{m+1}$ such that $H(w)$ is obtained from w by a projection reduction and $H(w)$ has a perfect matching iff $w \in 1^*0^*$. The edge $\{a_{m+1}, b_{m+1}\}$ and the edges $\{a_i, b_1\}$ are in $H(w)$ for all i with $2 \leq i \leq m+1$. For $1 \leq i \leq m$, the edge $\{a_i, b_{i+1}\}$ is in $H(w)$ iff $w_i = 1$, and the edge $\{a_i, b_i\}$ is in $H(w)$ iff $w_i = 0$. If $w \notin 1^*0^*$ then $w_i = 0$ and $w_{i+1} = 1$ for some i; this means that the vertex b_{i+1} has no edges adjacent to it, so $H(w)$ cannot have a perfect matching. Conversely, if $w = 1^p0^q$ then the perfect matching is $\{a_i, b_{i+1}\}$ for $1 \leq i \leq p$, $\{a_{p+1}, b_1\}$, and $\{a_i, b_i\}$ for $p+2 \leq i \leq m+1$.

BIPARTITE-MATCHING $\leq_{\text{proj}}$ CIRCUIT-VALUE

Since BIPARTITE-MATCHING $\in \mathcal{P}$, this is immediate from Theorem 3.2(4).

This completes the proof of Theorem 3.1.

4. Upper bounds. It is not hard to show that BINARY-COUNT is in SIZE-DEPTH(poly, O($\log n$)) and that DIR-ST-CONNECTIVITY is in SIZE-DEPTH(O(2^n), constant). These upper bounds can be improved. Regarding DIR-ST-CONNECTIVITY, Schorr [18] gives a simulation of nondeterministic Turing machines by circuits; the following Theorem 4.1 is implicit in the simulation.

THEOREM 4.1 (Schorr). *There is a constant b such that for any integer $k > 0$ there is a constant c such that*

$$\text{DIR-ST-CONNECTIVITY} \in \text{SIZE-DEPTH}(2^{cn^{1/k}}, bk).$$

Proof. We sketch the proof for completeness. Let G be the given directed graph with m vertices. Let $n = m^2$ be the number of bits in G's adjacency matrix. A circuit of constant depth b and size exponential in $n^{1/k}$ can compute the adjacency matrix of a new m-vertex directed graph G' such that, for each pair of vertices v_i and v_j, the edge (v_i, v_j) is present in G' iff there is a directed path of length $\leq \lceil m^{1/k} \rceil$ from v_i to v_j in G. Fix an i and j, and let $r = \lceil m^{1/k} \rceil$. A *possible path* is any sequence $w_0, ..., w_r$ of $r+1$ vertices such that $w_0 = v_i$ and $w_r = v_j$. For each possible path $\rho = (w_0, ..., w_r)$, let $\tau(\rho)$ be the AND of the entries in G's adjacency matrix corresponding to the edges (w_t, w_{t+1}) for $0 \leq t < r$. Assume that all (i,i)-entries have been set to 1. Now the (i,j)-entry in the adjacency matrix of G' is the OR of $\tau(\rho)$ over all possible paths ρ from v_i to v_j. By placing k copies of this adjacency matrix transformation in series, the $(1, m)$-entry in the final matrix is 1 iff there is a path (necessarily of length $\leq m$) from v_1 to v_m in G. □

THEOREM 4.2. *If F = BINARY-COUNT or F is a family of one-output symmetric Boolean functions, then for any $\varepsilon > 0$,*

$$F \in \text{SIZE-DEPTH}(O(n \cdot 2^{(\log n)^{\varepsilon}}), O(\log n/\varepsilon \cdot \log\log n)).$$

Define 2-SIZE-DEPTH($S(n), D(n)$) to be the class of problems F such that each $f_n \in F$ is computed by a fan-in 2 circuit of size $\leq S(n)$ and depth $\leq D(n)$. Since Muller and Preparata [16] show that BINARY-COUNT and all families of one-output symmetric Boolean functions are in 2-SIZE-DEPTH(O(n), O($\log n$)), Theorem 4.2 is immediate from the following:

THEOREM 4.3. *For any $\varepsilon > 0$,*

$$\text{2-SIZE-DEPTH}(S(n), D(n)) \subseteq \text{SIZE-DEPTH}(O(2^{(\log n)^{\varepsilon}} \cdot S(n)), O(D(n)/\varepsilon \cdot \log\log n)).$$

Proof. Any fan-in 2 circuit with one output and depth $\leq (\varepsilon/2) \cdot \log\log n$ can depend on at most $(\log n)^{\varepsilon/2}$ inputs, so by Lemma 3.3 it can be replaced by a circuit of depth 2 and size $O(2^{(\log n)^{\varepsilon}})$. Given a fan-in 2 circuit of depth D, partition its nodes into levels L_1, L_2, ... such that level L_i has nodes whose depth is between $(i-1)(\varepsilon/2)\log\log n$ and $i(\varepsilon/2)\log\log n$. View each node in L_1 as the output node of a fan-in 2 circuit. By the observation above, each such circuit can be replaced by a circuit of size $O(2^{(\log n)^{\varepsilon}})$ and depth ≤ 2. Regard the output nodes of these new circuits as the "inputs" to level L_2. Apply this process to L_2, L_3, ... in sequence, so each level is compressed to depth 2. □

By our reductions, the upper bound SIZE-DEPTH($2^{cn^{1/k}}$, constant) holds for any problem $F \leq_{\text{cd-tt}}$ DIR-ST-CONNECTIVITY, and the upper bound SIZE-DEPTH(poly, O($\log n/\log\log n$)) holds for any problem $F \leq_{\text{cd-tt}}$ BINARY-COUNT (e.g., MULTIPLICATION and SORTING). Theorems 4.1 and 4.2 could serve as an avenue for proving that certain of the $\equiv_{\text{cd-tt}}$-classes in Fig. 1 are distinct. For example, if

$$\text{DIR-ST-CONNECTIVITY} \notin \text{SIZE-DEPTH}(\text{poly}, O(\log n/\log\log n))$$

then the class containing BINARY-COUNT and the class containing DIR-ST-CONNECTIVITY are different. The following result shows that if these two classes are equal then Savitch's theorem [17] can be improved in the nonuniform case.

THEOREM 4.4. *If* BINARY COUNT $\equiv_{\text{cd-tt}}$ DIR-ST-CONNECTIVITY, *then*

$$\text{NONUNIF-}\mathcal{NL} = \text{NONUNIF-}\mathcal{L}.$$

Proof. In the constant-depth circuit that performs the reduction DIR-ST-CONNECTIVITY $\leq_{\text{cd-tt}}$ BINARY-COUNT, replace each gate with fan-in $r > 2$ by a tree of depth O($\log r$); since r is bounded above by a polynomial in n, $\log r$ = O($\log n$). Now replace each "black box" computing some member of BINARY-COUNT by a Muller-Preparata [16] fan-in 2 circuit of depth O($\log n$) and polynomial size. Thus

$$\text{DIR-ST-CONNECTIVITY} \in \text{2-SIZE-DEPTH}(\text{poly}, O(\log n)).$$

The conclusion NONUNIF-$\mathcal{NL}$ = NONUNIF-$\mathcal{L}$ now follows easily from two facts:

(1) NONUNIF-$\mathcal{NL}$ $\leq_{\text{proj}}$ DIR-ST-CONNECTIVITY (Theorem 3.2(3));

(2) The circuit value problem for fan-in 2 circuits of polynomial size and

depth $D(n)$ ($\geq \log n$) can be solved by a deterministic Turing machine in space $O(D(n))$ (see [3, Lemma 1]).

Details are left to the reader. □

5. Conclusion. One direction for future work is to add other problems to the classification begun in Fig. 1. A more fundamental open question is whether any two of the groups in Fig. 1 can be collapsed into one $\equiv_{\text{cd-tt}}$-class, or conversely whether the $\leq_{\text{cd-tt}}$ relation between the two groups is strict. In proving the latter, a stronger result would be obtained by replacing $\leq_{\text{cd-tt}}$ with constant-depth Turing reducibility, $\leq_{\text{cd-T}}$, defined like $\leq_{\text{cd-tt}}$ except that now in the circuit that performs the reduction there can be directed paths from the output of one black box to the input of another black box. (We did not define $\leq_{\text{cd-T}}$ above since we needed at most the power of $\leq_{\text{cd-tt}}$ for our specific reductions.)

Another interesting area is to improve the known lower bounds on size-depth circuit complexity. Furst, Saxe and Sipser [8] show that unbounded fan-in circuits can possibly be used as a tool to separate the polynomial-time hierarchy from PSPACE by an oracle. However, the lower bound for PARITY proved in [8], that any constant depth circuit must have size at least $n^{c \cdot \log^* n}$ for some constant c, is not sufficient to imply the existence of such an oracle. Our reducibilities suggest that an improved lower bound might be easier for a problem, such as CONNECTIVITY, that lies above PARITY in the $\leq_{\text{cd-tt}}$ ordering.

Appendix. In several of the following problems we require the input to be of a particular form. For example, in MERGING, the two input lists must be sorted, and in BIPARTITE-MATCHING, NETWORK-FLOW, THRESHOLD and UNARY-TO-BINARY, certain parts of the input must be valid unary representations, i.e., words in 1^*0^*. If the input is not of the correct form, all bits of the output are defined to be zero.

ADDITION (MULTIPLICATION)
INPUT: Two m-bit binary numbers ($n = 2m$).
OUTPUT: Their sum (product).

BINARY-COUNT (UNARY-COUNT)
INPUT: n bits $x_1, \ldots, x_n$.
OUTPUT: $\lg(n)$-bit binary representation of their sum Σx_i (n-bit unary representation of Σx_i).

BINARY-TO-UNARY
INPUT: n bits $x_1, \ldots, x_n$.
OUTPUT: $2^{\lg(n)}$-bit unary representation of the number whose binary representation is $x_1 x_2 \ldots x_{\lg(n)}$.

BIPARTITE-MATCHING
INPUT: Adjacency matrix of a $2m$-vertex bipartite graph and an m-bit unary number k.
OUTPUT: Does the graph have a matching with $\geq k$ edges?

BIPARTITE-PERFECT-MATCHING
INPUT: Adjacency matrix of a $2m$-vertex bipartite graph.
OUTPUT: Does the graph have a perfect matching?

CIRCUIT-VALUE
INPUT: Binary representation of a fan-in 2 circuit (i.e., every gate has fan-in at most 2) with one output node, and an assignment of Boolean values to all input nodes (the details of the binary representation are not crucial).
OUTPUT: The Boolean value computed by the output node.

COMPARISON
INPUT: Two m-bit binary numbers z_1 and z_2 $(n = 2m)$.
OUTPUT: Is $z_1 > z_2$?

CONNECTIVITY (STRONG-CONNECTIVITY)
INPUT: Adjacency matrix of an undirected (directed) graph.
OUTPUT: Is the graph connected (strongly connected)?

DIR-ST-CONNECTIVITY (UNDIR-ST-CONNECTIVITY)
INPUT: Adjacency matrix of an m-vertex directed (undirected) graph.
OUTPUT: Is there a directed path (a path) from v_1 to v_m?

EULERIAN-CYCLE
INPUT: Adjacency matrix of an undirected graph.
OUTPUT: Is there a cycle that traverses every edge exactly once?

MAJORITY
INPUT: n bits $x_1, \ldots, x_n$
OUTPUT: Is $\Sigma x_i \geq n/2$?

MAXIMUM
INPUT: A list of m m-bit binary numbers $(n = m^2)$.
OUTPUT: The largest number in the list.

MERGING
INPUT: Two lists of m m-bit binary numbers, each list sorted in nondecreasing order $(n = 2m^2)$.
OUTPUT: The merged list (a number which appears k times in the input lists will appear duplicated k times in the output list).

MULTIPLE-ADDITION
INPUT: A list of m m-bit binary numbers $A_1, \ldots, A_m$.
OUTPUT: The $(m + \lg(m))$-bit binary representation of the sum ΣA_i.

MULTIPLICATION (see ADDITION)

NETWORK-FLOW
INPUT: m-bit unary numbers $c(i,j)$ for each $1 \leq i,j \leq m$ with $i \neq j$, and an m^2-bit unary number f $(n = m^3)$.
OUTPUT: Is there an integral flow of $\geq f$ from v_1 to v_m in the directed flow network with m vertices and capacity $c(i,j)$ on edge (v_i, v_j) for each i and j?

PARITY (ZERO-MOD-2^c, for constant c)
INPUT: n bits $x_1, \ldots, x_n$.
OUTPUT: Is $\Sigma x_i \not\equiv 0 \bmod 2$ (mod 2^c)?

SORTING
INPUT: A list of m m-bit binary numbers $(n = m^2)$.
OUTPUT: The same list sorted in nondecreasing order (a number which appears k times in the input list will be duplicated k times in the output list).

STRONG-CONNECTIVITY (see CONNECTIVITY)

THRESHOLD
INPUT: m bits $x_1, \ldots, x_m$, and m-bit unary number k.
OUTPUT: Is $\Sigma x_i \geq k$?

UNARY-COUNT (see BINARY-COUNT)

UNARY-TO-BINARY
INPUT: n-bit unary representation of a number k.
OUTPUT: $\lg(n)$-bit binary representation of k.

UNDIR-CYCLE
INPUT: Adjacency matrix of an undirected graph.
OUTPUT: Is there a cycle in the graph?

UNDIR-ST-CONNECTIVITY (see DIR-ST-CONNECTIVITY)

ZERO-MOD-2^c (see PARITY)

Acknowledgment. We thank Amir Schorr for allowing us to include his proof of Theorem 4.1, and we thank Nicholas Pippenger for pointing out the reduction of network flow to bipartite perfect matching.

REFERENCES

[1] R. ALELIUNAS, R. M. KARP, R. J. LIPTON, L. LOVASZ AND C. RACKOFF, *Random walks, universal traversal sequences, and the complexity of maze problems*, Proc. 20th IEEE Symposium on Foundations of Computer Science, 1979, pp. 218-223.

[2] C. BERGE, *Graphs and Hypergraphs*, North-Holland, New York, 1973.

[3] A. BORODIN, *On relating time and space to size and depth*, this Journal, 6 (1977), pp. 733-744.

[4] A. K. CHANDRA, S. FORTUNE AND R. J. LIPTON, *Unbounded fan-in circuits and associative functions*, Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 52-60.

[5] S. A. COOK, *The complexity of theorem proving procedures*, Proc. Third ACM Symposium on Theory of Computing, 1971, pp. 151-158.

[6] S. EVEN, *Graph Algorithms*, Computer Science Press, Potomac, MD, 1979.

[7] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, Proc. Tenth ACM Symposium on Theory of Computing, 1978, pp. 114-118.

[8] M. FURST, J. B. SAXE AND M. SIPSER, *Parity, circuits and the polynomial-time hierarchy*, Proc. 22nd IEEE Symposium on Foundations of Computer Science, 1981, pp. 260-270.

[9] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability, A Guide to the Theory of* NP-*Completeness*, W. H. Freeman, San Francisco, 1979.

[10] L. M. GOLDSCHLAGER, *A universal interconnection pattern for parallel computers*, J. Assoc. Comput. Mach., 29 (1982), pp. 1073-1086.

[11] J.-W. HONG, *On some deterministic space complexity problems*, this Journal, 11 (1982), pp. 591-601.

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

[13] N. D. JONES, *Space-bounded reducibility among combinatorial problems*, J. Comput. Sys. Sci., 11 (1975), pp. 68-85.

[14] R. M. KARP, *Reducibility among combinatorial problems*, in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher eds., Plenum Press, New York, 1972, pp. 85-104.

[15] R. E. LADNER, *The circuit value problem is log space complete for* P, SIGACT News, 7 (1975), pp. 18-20.

[16] D. E. MULLER AND F. P. PREPARATA, *Bounds to complexities of networks for sorting and for switching*, J. Assoc. Comput. Mach., 22 (1975), pp. 195-201.

[17] W. J. SAVITCH, *Relationships between nondeterministic and deterministic tape complexities*, J. Comput. Sys. Sci., 4 (1970), pp. 177-192.

[18] A. SCHORR, *Super-circuits*, manuscript, Computer Science Dept., Tel Aviv Univ., Tel Aviv, Israel, Sept. 1981.

[19] Y. SHILOACH AND U. VISHKIN, *Finding the maximum, merging and sorting in a parallel computation model*, J. Algorithms, 2 (1981), pp. 88-102.

[20] S. SKYUM AND L. G. VALIANT, *A complexity theory based on Boolean algebra*, Proc. 22nd IEEE Symposium on Foundations of Computer Science, 1981, pp. 244-253.

[21] L. J. STOCKMEYER AND A. R. MEYER, *Word problems requiring exponential time: preliminary report*, Proc. 5th ACM Symposium on Theory of Computing, 1973, pp. 1-9.

[22] L. STOCKMEYER AND U. VISHKIN, *Simulation of parallel random access machines by circuits*, this Journal, this issue, pp. xxx-xxx.

[23] U. VISHKIN, *Implementation of simultaneous memory address access in models that forbid it*, J. Algorithms, 4 (1983), pp. 45-50.

SIAM J. COMPUT.
Vol. 13, No. 3, August 1984

© 1984 Society for Industrial and Applied Mathematics
001

AN ALGORITHM FOR THE GENERAL PETRI NET REACHABILITY PROBLEM*

ERNST W. MAYR†

Abstract. An algorithm is presented for the general Petri net reachability problem. It is based on a generalization of the basic reachability tree construction which is made symmetric with respect to the initial and final marking. Sets of transition sequences described by finite automata are used for approximations to firing sequences, and the approximation error is assessed by Presburger expressions. The approximation algorithm is iterated until a sufficient criterion for reachability is satisfied. The exact computational complexity of our algorithm is an open problem.

Key words. Petri net, vector replacement system, reachability problem, decidability

1. Introduction. Petri nets (due to C. A. Petri [34]) are a mathematical model for the representation and the analysis of parallel processes. This model has been widely generalized and investigated [4], [13], [16], [17], [21]. Especially in [13], the connections between different variants of the basic model—which can also be formulated in a purely algebraic way as vector addition systems or vector replacement systems—were investigated. Summaries can be found in [29] and [33] exhibiting the relation of Petri nets to numerous other models of parallel computation.

Petri nets and their equivalent concepts can be a convenient model if only a finite number of *types* of objects have to be considered, and if in addition the transitions in such a system are possible whenever some minimal number of objects of every type are present.

Problems analyzed in modelling parallel systems by Petri nets therefore usually deal with dynamic aspects of the control structure, since Petri nets tend to abstract away the individuality of data objects. Examples of such problems include (partial or total) deadlock freeness, and liveness of the system. Solutions to these example problems proclaim, in a sense, the absence of "difficulties" for all states that are reachable in the system. Another question is whether some arbitrary state can be reached from a fixed initial state, or whether there is some reasonable effective description of the set of all reachable states. The former, the so-called general reachability problem, is of basic importance for many others. It is, for instance, recursively equivalent to the liveness problem [12]. Hence, an algorithm for one of these two problems automatically solves the other. Moreover, a number of other problems in the representation of parallel and concurrent systems, in language generating systems, in algebra and in number theory can be shown to be effectively reducible or equivalent to the reachability problem. We note that the proof of the undecidability of the inclusion problem for Petri net reachability sets [2]—extended in [14] to the equality problem—implies that there cannot be any reasonable closed effective representation for reachability sets in general. For restricted classes of Petri nets, however, such representations have been given [1], [9], [27], [30].

For the reachability problem, restricted subclasses of Petri nets have been investigated (a summary of results can be found in [9]), and heuristic methods have also been proposed to simplify specific Petri nets to obtain sufficient conditions [3], [15], [23], [25]. In [37], the decidability of the reachability problem for Petri nets with at

* Received by the editors July 2, 1981, and in final revised form May 27, 1983. This work was supported in part by the Deutsche Forschungsgemeinschaft, under grant I3 Ma 870/1-1.

† Department of Computer Science, Stanford University, Stanford, California 94305.

most three places was established, and in [18], this result was extended to up to five places. These methods fail for the general case since they rely on semilinear reachability sets, and there are Petri nets with six places that have nonsemilinear reachability sets. The general reachability problem was claimed to be decidable in [36], but no complete or convincing proof was given.

The undecidability of the reachability set inclusion problem suggests (at least to the author) *not* to proceed by trying to find a representation for the set of all states reachable from the initial state, and then trying to determine whether the desired final state was in that set. Rather, a treatment of the reachability problem has been chosen which is symmetric with respect to the initial and final state. After giving the notation and basic concepts, a generalization of the fundamental reachability tree construction [20] is introduced which uses finite automata to restrict the set of transition sequences possible in the original construction, and which can be symmetrized in the above sense. These finite automata could actually be coded into the Petri nets. We avoid this encoding, however, for reasons of notational convenience and presentational clarity. In the iterative main algorithm, each successive automaton is a refinement of its predecessor, and is used to carry over from one step of the iteration to the next the information obtained so far about potential firing sequences. The firing sequences from the initial to the final state are thus approximated by the regular sets of transition sequences as determined by the more and more restrictive finite automata. The approximation error is evaluated by effectively constructible Presburger expressions. Whenever the iteration terminates successfully, the resulting regular set of transition sequences is guaranteed to contain a firing sequence.

The algorithm is formulated nondeterministically. In a deterministic implementation, every nondeterministic choice has to be replaced by a branching to all possibilities. The number of such branches will always be finite, and we show that any branch of the nondeterministic computation terminates.

The presentation is structured as follows. In § 2, we introduce some basic notation about Petri nets and related concepts, as well as some essential facts about Presburger Arithmetic and semilinear sets. We also present a slightly modified version of the original reachability tree construction due to Karp and Miller, which helps us find maximal covering markings. This algorithm is subsequently used as a subroutine. In § 3, we introduce the concept of regular constraint graphs. They play the role of controlling nondeterministic finite automata, and they contain information about reachable states. We first define more and more restrictive properties of these regular constraint graphs without actually detailing how to construct them. Some of the properties are used for mere technical convenience, whereas others guarantee better and better approximations for the description of firing sequences by the regular constraint graphs. In § 4, we show that the strictest such condition, which we call *consistency*, in fact suffices to construct firing sequences from a regular constraint graph. Section 5 deals with the construction of consistent regular constraint graphs. The construction is achieved by an iterative algorithm where each iteration refines the regular constraint graph obtained by the previous iteration, subject to certain conditions. These conditions are given by expressions in Presburger Arithmetic (resp., a restricted subset thereof). They describe, in a sense, how far away the current regular constraint graph is from being consistent.

The algorithm is composed of steps which clearly show that the nondeterministic computation will actually produce a final output if there is a firing sequence. Hence, a deterministic implementation of our algorithm provides a decision procedure for the general Petri net reachability problem. In the last section, we list some decision

problems reducible or equivalent to the reachability problem. We also mention some open questions in connection with the reachability problem and the complexity of the algorithm.

An earlier version of this paper was presented in [28]. While it was being refereed, another algorithm for the reachability problem was published in [22].

2. Notation and basic properties.

2.1. Petri nets and related concepts. Let $\mathbf{Z}$ denote the set of integers, $\mathbf{N}$ the subset of nonnegative integers, and, for $n \in \mathbf{N}$, let I_n be the (index) set $\{1, \cdots, n\}$. Further, let $\bar{\mathbf{N}} =_{\text{def}} \mathbf{N} \cup \{\omega\}$ stand for the set $\mathbf{N}$ augmented by the "infinite" number ω with the additional rules $\pm n + \omega = \omega \pm n = \omega$ and $n < \omega$ for all $n \in \mathbf{N}$.

A *Petri net P* is a triple (S, T, K) with

(i) $S = \{s_1, \cdots, s_v\}$ a finite set of *places*;

(ii) $T = \{t^1, \cdots, t^w\}$ a finite set of *transitions*, disjoint from S;

(iii) K a mapping from $(S \times T) \cup (T \times S)$ into $\mathbf{N}$, indicating the *multiplicity* of *directed edges* between places and transitions.

A Petri net hence is a directed bipartite multigraph.[1]

A *marking* (resp., *pseudomarking*) of a Petri net $P = (S, T, K)$ is a mapping $m : S \to \mathbf{N}$ (resp., $\bar{m} : S \to \bar{\mathbf{N}}$). It is naturally represented as a vector $m \in \mathbf{N}^v$ (resp., $\bar{m} \in \bar{\mathbf{N}}^v$), and will, for notational convenience, be written as a row vector $m = (m_1, \cdots, m_v)$. Of course, every marking is also a pseudomarking. We call a Petri net together with a marking or pseudomarking a *marked Petri net.*

We shall make use of the following relations defined for markings and pseudomarkings:

(i) $\bar{m} \leqq \bar{m}' \Leftrightarrow_{\text{def}} \bar{m}_i \leqq \bar{m}'_i$ for all $i \in I_v$.

The relation $\leqq$ is the usual partial order on vectors, given by the natural componentwise ordering. The relation $<$ will be its irreflexive subset.

In general it will be helpful to look at ω informally as a big but otherwise unspecified "number" or as a "wild card". Hence we may use another partial order based on the amount of lacking information:

(ii) $\bar{m} \preceq \bar{m}' \Leftrightarrow_{\text{def}}$ "$\bar{m}$ is more definite than $\bar{m}'$", i.e., $\bar{m}'_i$ finite (i.e. $\neq \omega$) implies $\bar{m}_i = \bar{m}'_i$, for all $i \in I_v$.

(iii) We say that $\bar{m}$ and $\bar{m}'$ are *compatible* if their finite coordinates are consistent, i.e., if $\bar{m}_i = \bar{m}'_i$ whenever both coordinates are finite.

If $\bar{m} \preceq \bar{m}'$ we say that $\bar{m}$ is *under* $\bar{m}'$ (and $\bar{m}'$ *over* $\bar{m}$). Again, $\prec$ denotes the irreflexive part of this relation. Thus, $\bar{m} = (2, 3, \omega, 1, \omega)$ is less than $\bar{m}' = (2, 4, \omega, \omega, \omega)$ but not under $\bar{m}'$, whereas $\bar{m}$ is under $\bar{m}'' = (2, 3, \omega, \omega, \omega)$, and hence of course also $\leqq \bar{m}''$. Another way to look at the relation $\preceq$ is to note that, if $\bar{m} \preceq \bar{m}'$, we can obtain $\bar{m}$ from $\bar{m}'$ by changing some (possibly zero) components of $\bar{m}'$ to finite values.

Figure 1 shows an example of a marked Petri net (P, m). The places are drawn as circles, with the numbers inside indicating the nonzero values of the marking, and the transitions are represented as bars. The marking m is $(0, 0, 1, 1, 0, 0)$, and the multiplicity of all edges is one.

Transitions in a Petri net $P = (S, T, K)$ with pseudomarking $\bar{m}$ may be *fired* and thus lead to new pseudomarkings. In particular, a transition $t \in T$ is *firable* at the pseudomarking $\bar{m}$ if the vector t^- which has as its ith component the multiplicity of the edge from the ith place of P to the transition t, is less than or equal to $\bar{m}$, i.e., if

$$t^- =_{\text{def}} (K(s_1, t), \cdots, K(s_v, t)) \leqq \bar{m}.$$

[1] As a matter of fact, there is no loss of generality if we restrict ourselves to bipartite digraphs (without multiple edges); see [13].

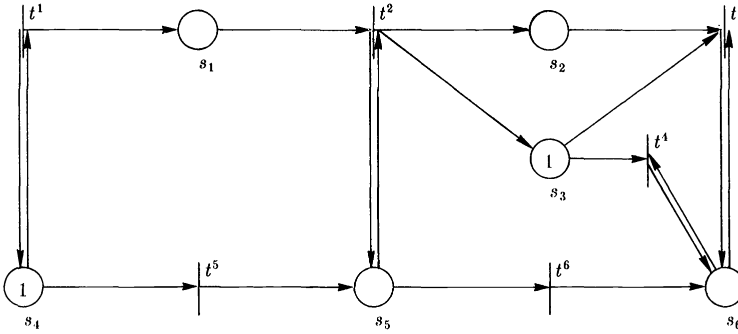


FIG. 1

The vector t^- can be considered the *prerequisite* for firing transition t. The *firing* of a firable transition $t \in T$ changes the pseudomarking $\bar{m}$ to the new pseudomarking

$$\bar{m} + (K(t, s_1) - K(s_1, t), \cdots, K(t, s_v) - K(s_v, t)).$$

In order to keep notation simple (without, hopefully, confusing the reader), we also use t for the *effect* of firing t, i.e., for $(K(t, s_1) - K(s_1, t), \cdots, K(t, s_v) - K(s_v, t))$. Hence the firing of t above changes $\bar{m}$ to $\bar{m} + t$. We write $\bar{m} \to^t \bar{m}'$ to denote the fact that the pseudomarking $\bar{m}$ is changed to $\bar{m}'$ by firing the firable transition t. For transition sequences $\tau = t^{i_1} \cdots t^{i_r} \in T^*$, we define *prerequisite*, *firability*, and *effect* inductively as follows:

(i) the empty sequence (with $r = 0$) has prerequisite $\tau^- =_{\text{def}} (0, \cdots, 0) \in \mathbf{N}^v$ and is firable at any $\bar{m} \in \bar{\mathbf{N}}^v$;

(ii) if $r > 0$, then the prerequisite is $\tau^- =_{\text{def}} \max\{(t^{i_1})^-, (t^{i_2} \cdots t^{i_r})^- - t^{i_1}\}$; (max componentwise), and τ is firable at $\bar{m}$ if t^{i_1} is firable at $\bar{m}$ and $t^{i_2} \cdots t^{i_r}$ is firable at $\bar{m} + t^{i_1}$.

It is now obvious that the effect of τ (denoted by the same letter) is $\tau = \sum_{j=1}^{r} t^{i_j}$, and that $\bar{m} \to^\tau \bar{m}'$ iff $\bar{m}' = \bar{m} + \tau$ and τ is firable at $\bar{m}$.

It should be noted (and could be established by an easy induction) that the prerequisite τ^- is the minimal marking at which τ is firable. This marking is well defined for all $\tau \in T^*$. A transition sequence $\tau \in T^*$ is called a *firing sequence* for some pseudomarking $\bar{m}$ if τ is firable at $\bar{m}$.

In the Petri net of Fig. 1, the sequence $\tau = t^1 t^1 t^5$ is firable at $m = (0, 0, 1, 1, 0, 0)$. The Petri net with the resulting marking $m + \tau$ is shown in Fig. 2.

The *reachability set* $\mathcal{R}(P, \bar{m})$ of a marked Petri net $(P, \bar{m})$ is defined as

$$\mathcal{R}(P, \bar{m}) =_{\text{def}} \{\bar{m}'; \bar{m} \to^\tau \bar{m}' \text{ for some } \tau \in T^*\}.$$

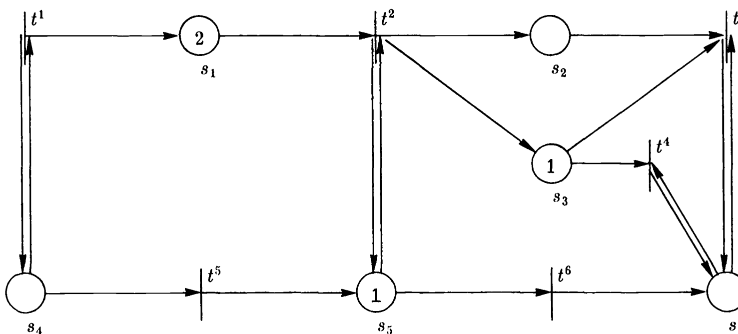


FIG. 2

If $\bar{m}' \in \mathcal{R}(P, \bar{m})$ we say that the pseudomarking $\bar{m}'$ is *reachable* (in P) from $\bar{m}$, and we also write $\bar{m} \rightarrow^* \bar{m}'$, the Petri net P being understood.

The *general reachability problem for Petri nets* is the decision problem for the set of all (well-formed) triples (P, m, m') such that m' is reachable from m in P.

There is another purely algebraic formulation of Petri nets, called *vector replacement systems* (VRS) [21]. A vector replacement system consists of a finite set of pairs $(u^i, v^i) \in \mathbf{N}^v \times \mathbf{Z}^v$ with $u^i + v^i \geqq 0$, an *initial vector* $\bar{m} \in \bar{\mathbf{N}}^v$,[2] and a *derivation rule*

$$\bar{m}' \rightarrow \bar{m}' + v^i \quad \text{whenever } \bar{m}' \geqq u^i.$$

Thus, $\bar{m}'$ can be changed to $\bar{m}' + v^i$ whenever it is not less than the *test vector* (or *prerequisite*) u^i. Note that $u^i + v^i \geqq 0$ implies that $\bar{m}' + v^i \geqq 0$. The *reachability set* of a VRS with initial vector $\bar{m}$ is then the set of all $\bar{m}'$ such that $\bar{m} \rightarrow^* \bar{m}'$. Here, $\rightarrow^*$ denotes the reflexive transitive closure of $\rightarrow$.

It should by now be clear that a Petri net $P = (S, T, K)$ with $|S| = v$ and $T = \{t^1, \cdots, t^w\}$ translates into an equivalent VRS with the pairs $((t^i)^-, t^i)$. The initial pseudomarking becomes the initial vector of the VRS, and the reachability relation is the same in both cases. Given a VRS, it is obvious how to construct a Petri net with the same reachability relation, by equating a pair (u^i, v^i) with $((t^i)^-, t^i)$.

2.2. Semilinear sets and paths in digraphs. A *linear set* $L \subseteq \mathbf{N}^w$ is a set of the form

$$L = \left\{ b + \sum_{i=1}^{r} n_i p^i ;\ (n_1, \cdots, n_r) \in \mathbf{N}^r \right\}$$

for some $r \in \mathbf{N}$ and vectors $b, p^1, \cdots, p^r \in \mathbf{N}^w$. The vector b is called the *base* of L.

A *semilinear set* is a finite union of linear sets.

Semilinear sets are precisely those sets definable by expressions in Presburger Arithmetic, i.e. the first order theory of the nonnegative integers with addition [8], [35]. Semilinear sets are, therefore, closed under Boolean operations. There are, furthermore, effective procedures to construct semilinear representations of the sets defined by Presburger expressions and to decide Presburger formulae [8], [31].

As an example, let $L = \{b + \sum_{i=1}^r n_i p^i;\ (n_1, \cdots, n_r) \in \mathbf{N}^r\}$ be a linear set. Then the statement "$(x_1, \cdots, x_w) \in L$" can be written as a Presburger expression as follows (we use the obvious abbreviations; remember that variables in Presburger Arithmetic range over $\mathbf{N}$):

$$(\exists n_1, \cdots, n_r)\left[\left(b_1 + \sum_{i=1}^{r} n_i p_1^i = x_1\right) \wedge \cdots \wedge \left(b_w + \sum_{i=1}^{r} n_i p_w^i = x_w\right)\right].$$

Also, the fact that the projection of L onto the first coordinate is unbounded, can be expressed in Presburger arithmetic in the following way:

$$(\forall a\ \exists d, n_1, \cdots, n_r)\left[a + d = b_1 + \sum_{i=1}^{r} n_i p_1^i\right].$$

Suppose that A is a digraph whose edges are labelled with elements from the set T of transitions of some Petri net $P = (S, T, K)$. Assume further that there are distinguished nodes z^0 and z^f (possibly $z^0 = z^f$) in A. We may look at A as defining a nondeterministic finite automaton over T, with initial state z^0 and final state z^f. It is well known that the set of the labelling sequences τ of all paths in A from z^0 to z^f is a regular subset L_A of T^*. If $T = \{t^1, \cdots, t^w\}$ and $\Phi(t^i, \tau)$ denotes the number of

[2] In the original definition [21], the initial vector is some $m \in \mathbf{N}^v$. Our version is a slight generalization.

occurrences of t^i in τ, for any $\tau \in T^*$ and $t^i \in T$, then it is also known [32] that the set

$$\{(\Phi(t^1, \tau), \cdots, \Phi(t^w, \tau)); \tau \in L_A\}$$

is semilinear.[3]

Consequently, we can express "$m' = m + \tau$ for some $\tau \in L_A$" in Presburger arithmetic (note that we are not claiming $m \to^{\tau} m'$). We only need to observe that

$$\tau = \sum_{i=1}^{w} \Phi(t^i, \tau) t^i.$$

In fact, an appropriate Presburger expression can be effectively constructed given P, m, m', and A.

A *simple* path in a digraph is a path such that all vertices on the path are distinct (and hence all edges are also distinct). A *cycle* is a closed path, i.e., its first and last node are the same. A *simple cycle* is a closed path that is simple except for the first and last node. If α is a path in a digraph whose edges are labelled (with elements from some set T), then $\tau(\alpha)$ will be used to denote the sequence in T^* given by the sequence of α's edge-labels. A *strongly connected component* or SCC of a digraph is a subgraph induced by a maximal subset of nodes in the digraph such that there is a path from every node in the subset to every other node in the subset.

Finally, we introduce the concept of *reversing* a digraph, and some related notions. Let A be a digraph, and let α be a path in A from the node *first*(α) to the node *last*(α). Then A_{rev} denotes the *reversed graph* of A obtained by reversing the orientation of all edges in A, α_{rev} denotes the *reversed path* in A_{rev} corresponding to α and leading from *last*(α) to *first*(α). In particular, if A is a Petri net $P = (S, T, K)$, then we refer to the transition in P_{rev} corresponding to t in P as t_{rev}. Note that the effect of t_{rev} is $-t$. For sequences $\tau = t^{i_1} \cdots t^{i_r} \in T^*$, we set $\tau_{\text{rev}} =_{\text{def}} t^{i_r}_{\text{rev}} \cdots t^{i_1}_{\text{rev}}$ (this is a slight misuse of notation since actually t as a node does not change when P is reversed to P_{rev}). If A is a digraph with edge-labels from a set T of transitions, the label t in A is replaced by t_{rev} in A_{rev}.

2.3. Maximum cover pseudomarkings. Suppose $\bar{m}$ is a pseudomarking of some Petri net $P = (S, T, K)$, and $\tau \in T^*$ is a firing sequence firable at $\bar{m}$ such that the resulting pseudomarking $\bar{m}' = \bar{m} + \tau$ is $\geqq \bar{m}$. Then clearly τ is also firable at $\bar{m}'$, and we may in fact, starting from $\bar{m}$, repeat firing τ arbitrarily many times. If τ_i, the ith component of the effect τ, is greater than zero, then, by firing τ over and over again, we can increase the ith component of the resulting pseudomarking as much as we like. In the constructions which we are about to discuss, we shall represent this fact by changing the ith component to ω. While these constructions are variants of the basic reachability tree constructions in [13], [14], [20], we shall only be interested in sequences τ which are also elements of a given regular set L_A. Intuitively, L_A can be thought of as representing the (partial) information or constraints which we have gathered so far on possible firing sequences of interest.

Formally, let A be some (nondeterministic) finite automaton over T. Let A be given by its state transition diagram, let z^0 be some state (= node) of A, and let $\bar{m}, \bar{m}' \in \mathbf{N}^v$ be pseudomarkings. The pseudomarking $\bar{m}'$ is called an (A, z^0)*-cover* of $\bar{m}$ if there is some transition sequence $\tau \in T^*$ such that

(i) there is a (closed) path in A from z^0 to itself whose edge labelling sequence is τ;

[3] Such a statement actually holds more generally for context-free languages, and is sometimes referred to as Parikh's Lemma.

(ii) τ is firable at $\bar{m}$;

(iii) $\bar{m} \leqq \bar{m} + \tau$; and

(iv) $\bar{m}'_i$ is ω whenever $\tau_i > 0$, and it is $\bar{m}_i$ otherwise, for $i = 1, \cdots, v$.

A pseudomarking $\bar{m}''$ is a *maximum* (A, z^0)*-cover* of $\bar{m}$ if it is an (A, z^0)-cover of $\bar{m}$ which is over all (A, z^0)-covers of $\bar{m}$. A corresponding transition sequence τ is called a *maximum* (A, z^0)*-covering sequence* for $\bar{m}$.

We now show that there is an effective procedure which, given A, z^0, and $\bar{m}$, determines a maximum (A, z^0)-cover together with a maximum (A, z^0)-covering sequence for $\bar{m}$. This construction varies from the reachability tree construction in [20] by taking into account the finite automaton A. The algorithm constructs a digraph $MC(P, \bar{m}, A, z^0)$ in which every node k has two labels attached to it, namely a label $\tilde{m}(k) \in \bar{\mathbf{N}}^v$ and a label $\tilde{z}(k)$ in the node set (= state set) of A. Also, every edge e of $MC(P, \bar{m}, A, z^0)$ carries a label $t(e) \in T$. The graph $MC(P, \bar{m}, A, z^0)$ is constructed as the labelled digraph MC in the following

ALGORITHM 1.

```
let MC initially consist of one node, the "root" r;
m̃(r) := m̄; z̃(r) := z⁰;
declare r "unfinished";
while there are "unfinished" nodes in MC do
  choose some "unfinished" node k, declare it "finished";
  for every edge (z̃(k), z') in A labelled t ∈ T with t firable at m̃(k) do
    add a new edge e to MC, with first(e) = k, and a new node k' as last(e);
    t(e) := t; m̃(k') := m̃(k) + t; z̃(k') := z';
    if there is some node k" on the simple path from r to k' (excluding k') such that
      m̃(k') = m̃(k"), and
      z̃(k') = z̃(k")
    then
      identify k' with k" co k' thus becomes "finished" oc
    else
      declare k' "unfinished";
      if there are nodes k" on the simple path from r to k' such that
        m̃(k') and m̃(k") have the same set of ω-coordinates,
        m̃(k') > m̃(k"), and
        z̃(k') = z̃(k") co such a k" need not be uniquely determined oc
      then
        choose such a k";
        for each i ∈ I_v such that (m̃(k'))_i > (m̃(k"))_i do (m̃(k'))_i := ω od
      fi
    fi
  od
od
end Algorithm 1.
```

We leave it to the reader to verify that for every node k in the constructed graph, there is indeed (as claimed in the algorithm) a unique simple path from the root r to k. The proof for the termination of Algorithm 1 is by contradiction and runs along the same lines as in [14], [20] but takes into account that there are only finitely many states in A. It is based on the observation that if the algorithm did not terminate, there would be an infinite simple path starting at the root. The sequence of vertices on this path contains an infinite subsequence such that all vertices in the subsequence have the same $\tilde{z}$-label. It is clear from the construction in the algorithm that if vertex k' comes after vertex k in this subsequence, the label $\tilde{m}(k')$ contains at least the same ω-coordinates as the label $\tilde{m}(k)$. Since there are only finitely many coordinates, there must now be an infinite subsequence of the above subsequence such that all of its

vertices have the same ω-coordinates in their $\tilde{m}$-label. But every such infinite sequence of finite dimensional vectors with nonnegative integer components (which we obtain by taking the projection onto the non-ω-coordinates) must contain an infinite subsequence that is nondecreasing with respect to $\leqq$ [7]. This now contradicts the definition of the algorithm. Indeed, if we let k' be the node belonging to the second element in this last sequence, either k' would have been identified with an earlier node on the simple path from the root r to k', or $\tilde{m}(k')$ would contain strictly more ω-coordinates than the $\tilde{m}$-labels of its predecessors on this simple path.

LEMMA 1.

(a) *The graph* $MC(P, \bar{m}, A, z^0)$ *contains a node* k *such that* $\tilde{z}(k)=z^0$ *and* $\tilde{m}(k') \leq \tilde{m}(k)$ *for every node* k' *in the graph with* $z^0=\tilde{z}(k')$ *and* $\bar{m} \leq \tilde{m}(k')$. *The label* $\tilde{m}(k)$ *of this node is a maximum* (A, z^0)*-cover for* $\bar{m}$.

(b) *From* $MC(P, \bar{m}, A, z^0)$, *a maximum* (A, z^0)*-covering sequence for* $\bar{m}$ *can effectively be determined.*

Proof. For a proof, we refer the reader to [13, § 3] and [14, § 3]. The proof of Lemma 1 is a straightforward generalization of the proofs given there. □

For example, consider once again the Petri net P of Fig. 1, but now with initial marking $\bar{m} = (\omega, 0, 0, 0, 1, 0)$. If we let A be the trivial automaton consisting of just one state, z^0, with a self-loop attached to it for every transition in the Petri net, then $MC(P, \bar{m}, A, z^0)$ contains exactly the following simple paths starting from the root (we let $\tilde{m}(k)$ stand for every node k):

$$(\omega, 0, 0, 0, 1, 0) \underset{t^2}{\rightarrow} (\omega, \omega, \omega, 0, 1, 0) \underset{t^6}{\rightarrow} (\omega, \omega, \omega, 0, 0, 1),$$

and

$$(\omega, 0, 0, 0, 1, 0) \underset{t^6}{\rightarrow} (\omega, 0, 0, 0, 0, 1).$$

This shows that $(\omega, \omega, \omega, 0, 1, 0)$ is the maximum (A, z^0)-cover, and $\tau = t^2$ is a maximum (A, z^0)-covering sequence for $\bar{m}$.

The maximum (A, z^0)-covering sequence in the above example happens to be the labelling sequence of a simple path in $MC(P, \bar{m}, A, z^0)$. This coincidence is not true in general. Rather, if τ is the labelling sequence of the simple path from the root to a node k whose $\tilde{m}(k)$ is a maximum (A, z^0)-cover for $\bar{m}$ and which has $\tilde{z}(k)=z^0$, then τ can be subdivided into $\tau=\tau^1\tau^2 \cdots \tau^h$, and a maximum (A, z^0)-covering sequence for $\bar{m}$ can be chosen to be of the form $(\tau^1)^{r_1}(\tau^2)^{r_2} \cdots (\tau^h)^{r_h}$ for appropriate r_i. Here, $(\tau^i)^{r_i}$ denotes the composition of r_i copies of τ^i. Note that the subdivision basically takes place at those nodes where the number of ω-coordinates of the $\tilde{m}$-labels increases, and at the corresponding nodes k'' as used in the above algorithm. The reason for the composition is that the segments of τ following such a node where the number of ω-coordinates increases, might actually have the effect of decreasing a component which changed to ω at this node. Of course, this fact does not become apparent in the corresponding $\tilde{m}$-labels because $\omega - n = \omega$ for all $n \in \mathbf{N}$. But the decrease can be balanced by repeating sufficiently many times earlier segments which do increase that coordinate. For more details (which we do not use), we refer again to [13, § 3] and [14, § 3].

3. Regular constraint graphs. We now introduce the concept of a *regular constraint graph* which is basically a controlling finite automaton as above. It can also be viewed as a generalization of the basic reachability tree construction in [20]. However, we shall first give a nonalgorithmic definition of regular constraint graphs that is motivated by some technically desirable properties, then show how to use them in the next chapter, and finally discuss a series of algorithms to construct them.

Regular constraint graphs are supposed to represent, in a canonical and technically convenient form, sets of transition sequences which are (under varying criteria) possible candidates for firing sequences.

DEFINITION 1. Let $P=(S, T, K)$ be a Petri net, and R a digraph with an edge label $t(e) \in T$ for every edge e of R. We call R a *regular constraint graph* for P if the following properties hold:

(i) R has two distinguished nodes: an *initial* node r and a *final* node q.

(ii) For every node k of R, there is a path from r to q containing k, and there is only one simple path from r to q; this path is called the *base path* of R.

(iii) The nodes r and q are SCC's by themselves, and every SCC of R has exactly one node in common with the base path.

Intuitively, R represents all transition sequences that are t-label sequences of paths from r to q. The traversal of an edge e with label $t(e)=t$ in a regular constraint graph thus corresponds to the firing of a transition $t \in T$. Assume that it takes some pseudomarking $\bar{m}$ to $\bar{m}'=\bar{m}+t$. If we want to associate this change of pseudomarkings with the edge e, it turns out to be very convenient to subdivide the firing of t into two phases: in the first phase, the pseudomarking $\bar{m}$ is decreased by t^-, in the second the resulting intermediate pseudomarking is increased by $t^+ := t + t^-$. (Hence $t = t^+ - t^-$, and both t^+ and t^- are nonnegative vectors.) To denote the change in these intermediate pseudomarkings effected by some transition sequence $\tau = t^{i_1} \cdots t^{i_r} \in T^*$ we set

$$\hat{\tau} =_{\text{def}} \tau + (t^{i_1})^- - (t^{i_r})^+,$$

i.e., we omit from the total effect τ the effect of the first phase of the first transition and the effect of the second phase of the last transition in τ.

Let R be a regular constraint graph, let $e^1, \cdots, e^p$ be the sequence of edges on the base path of R, and let $r = k^0, k^1, \cdots, k^p = q$ be the sequence of nodes. Note that all edges e^i connect different SCC's.

DEFINITION 2. We say that edge labels $\hat{m}(e) \in \bar{\mathbf{N}}^v$ for every edge e in R are *weakly consistent* with a pair of markings (m, m') if the following conditions hold:

(i) The marking determined by $\hat{m}$ for r (resp., q) is m (resp., m'), i.e.,

$$\hat{m}(e^1) + t^-(e^1) = m \text{ and } \hat{m}(e^p) + t^+(e^p) = m'.$$

(For notational convenience, we write $t^-(e)$ and $t^+(e)$ for $(t(e))^-$ and $(t(e))^+$, resp.)

(ii) Whenever there is a two-edge path ee' within an SCC or such that the SCC of $last(e)$ ($= first(e')$) contains no edges, then the labels of e and e' satisfy

$$\hat{m}(e') = \hat{m}(e) + \hat{\tau}(ee').$$

(iii) Whenever there is a two-edge path $e^i e'$ with e^i on the base path and e' within the SCC C^i of $k^i = last(e^i)$, then the labels of e^i and e' satisfy

$$\hat{m}(e^i) + \hat{\tau}(e^i e') \leq \hat{m}(e').$$

(iv) Whenever there is a two-edge path ee^i with e^i on the base path and e within the SCC C^{i-1} of $k^{i-1} = first(e^i)$, then the labels of e and e^i satisfy

$$\hat{m}(e^i) \leq \hat{m}(e) + \hat{\tau}(ee^i).$$

We note that as an immediate consequence of (ii) above, the $\hat{m}$-labels of all edges within the same SCC have the same set of ω-coordinates. Also, if $\hat{m}$ is a weakly consistent edge-labelling for R and (m, m'), then setting $\hat{m}(e_{\text{rev}}) =_{\text{def}} \hat{m}(e)$ for each edge e_{rev} in R_{rev} produces an $\hat{m}$-labelling for R_{rev} (considered as a regular constraint graph for P_{rev}) that is weakly consistent with (m', m).

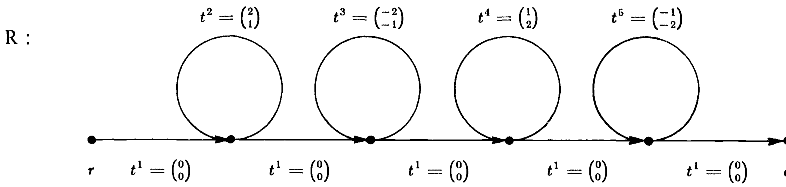


FIG. 3a. *Regular constraint graph.*

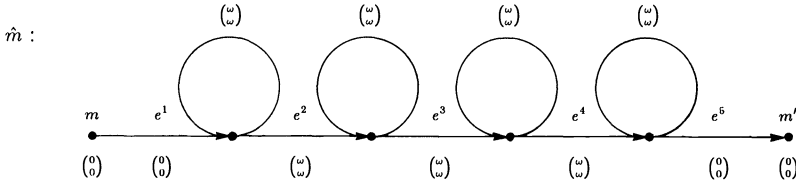


FIG. 3b. *Weakly consistent $\hat{m}$-labelling for R and R_{rev}.*

In the example in Fig. 3, we consider a Petri net with transitions $t^1, \cdots, t^5$ where (with transition t written as $t^+ - t^-$)

$$t^1 = \begin{pmatrix}0\\0\end{pmatrix} - \begin{pmatrix}0\\0\end{pmatrix}, \quad t^2 = \begin{pmatrix}2\\1\end{pmatrix} - \begin{pmatrix}0\\0\end{pmatrix}, \quad t^3 = \begin{pmatrix}0\\0\end{pmatrix} - \begin{pmatrix}2\\1\end{pmatrix}, \quad t^4 = \begin{pmatrix}1\\2\end{pmatrix} - \begin{pmatrix}0\\0\end{pmatrix}, \quad t^5 = \begin{pmatrix}0\\0\end{pmatrix} - \begin{pmatrix}1\\2\end{pmatrix}.$$

Let R be the regular constraint graph as given in Fig. 3a, and let $m = m' = (0, 0)$. It should be clear that the $\hat{m}$-labelling given in Fig. 3b is weakly consistent with (m, m').

Regular constraint graphs with weakly consistent $\hat{m}$-labels may sometimes provide good approximations for possible firing sequences. In general, however, the approximation is still too crude (otherwise they would not be termed *weakly* consistent). Informally, they are insufficient for basically two reasons. First, they may contain edges with t-labels that cannot occur in any firing sequence under consideration, and second, some edges may have $\hat{m}$-labels with ω-coordinates that are "unjustified" in the sense that the corresponding coordinates of every marking generated by a firing sequence when "passing through" such an edge are bounded. The $\hat{m}$-label of e^3 in Fig. 3b is an example of the latter possibility (here $\binom{0}{0}$ is the only possible intermediate marking when passing along e^3).

We shall now show how to remove these two deficiencies by taking a closer look at the t-label sequences of paths in a regular constraint graph. Again, let P be some Petri net, m and m' two (fixed) markings of P, and let R be a regular constraint graph for P, with $e^1, \cdots, e^p$ the sequence of edges and $r = k^0, k^1, \cdots, k^p = q$ the sequence of nodes on its base path. Also, let α be some arbitrary path in R from r to q.

DEFINITION 3.

(a) Let α_j denote the (uniquely determined) initial segment of α whose last edge is e^j. The last node of α_j is k^j. By convention, α_0 denotes the path consisting solely of k^0.

(b) Call α *admissible* iff

$$m + \tau(\alpha) = m' \quad \text{and} \quad (\forall j \in I_p)[\hat{m}(e^1) + \hat{\tau}(\alpha_j) \geqq 0],$$

i.e., iff the total effect $\tau(\alpha)$ is $m' - m$ (as desired, if we are interested in whether $m' \in \mathcal{R}(P, m)$), and all initial segments α_j of α satisfy the firability condition that $m = (\tau(\alpha_j) - t(e^j))$ is greater than or equal to the prerequisite $t^-(e^j)$. Note, however, that for other initial segments α' of α which do not end with some edge on the base path, the intermediate marking $\hat{m}(e^1) + \hat{\tau}(\alpha')$ may very well contain negative coordinates.

DEFINITION 4. Let the notation be as above. We say that $\hat{m}$ is *consistent* with (m, m') if the following properties hold:

(a) Suppose that the $\hat{m}$-labels within some SCC C^i of R contain strictly more ω-coordinates than does the label $\hat{m}(e^i)$ of the edge on the base path entering C^i in k^i. Let e' be an edge within C^i leaving k^i. Then the pseudomarking determined by e' for k^i is a maximum (C^i, k^i)-cover for the pseudomarking determined for k^i by $\hat{m}(e^i)$, i.e.,

$$\hat{m}(e')+t^-(e') \text{ is a maximum } (C^i, k^i)\text{-cover for } \hat{m}(e^i)+t^+(e^i).$$

(b) Suppose that the $\hat{m}$-labels within some SCC C^i of R contain strictly more ω-coordinates than does the label $\hat{m}(e^{i+1})$ of the edge on the base path leaving C^i (from k^i). Let e be an edge entering k^i within C^i. Then the pseudomarking determined by $\hat{m}(e)$ for k^i is a maximum (C^i_{rev}, k^i)-cover for the pseudomarking determined for k^i by $\hat{m}(e^{i+1})$, i.e.,

$$\hat{m}(e)+t^+(e) \text{ is a maximum } (C^i_{\text{rev}}, k^i)\text{-cover for } \hat{m}(e^{i+1})+t^-(e^{i+1}).$$

(Note that (b) is the equivalent of (a) for R_{rev}.)

(c) There are admissible paths α and α' such that the following conditions are satisfied:

(i) the path α' contains every edge within an SCC of R more often than does the path α;

(ii) for every edge e^j on the base path, $\hat{m}(e^1)+\hat{\tau}(\alpha_j)$ is under $\hat{m}(e^j)$;

(iii) for every $j \in I_p$, we have $\hat{\tau}(\alpha') \geqq \hat{\tau}(\alpha)$, and the ith coordinate of $\hat{\tau}(\alpha')$ is strictly greater than that of $\hat{\tau}(\alpha)$ if and only if the ith coordinate of $\hat{m}(e^j)$ equals ω, for all $i \in I_v$.

The paths α and α' are said to *justify* the labelling $\hat{m}$.

It follows immediately from this definition that if $\hat{m}$ is a consistent edge-labelling for some regular constraint graph R (with respect to a Petri net P and markings m and m'), then $\hat{m}$ is also consistent for R_{rev} (with respect to P_{rev} and markings m' and m).

In Fig. 3b, if we replace $\hat{m}(e^3)$ by $\binom{0}{0}$, the resulting $\hat{m}$-labelling becomes consistent as the reader can easily verify.

Let R be a regular constraint graph and $\hat{m}$ a consistent edge-labelling (with respect to m and m'). Suppose that the $\hat{m}$-labels within some SCC C^i of R contain strictly more ω-coordinates than does the label $\hat{m}(e^i)$ of the edge on the base path entering C^i in k^i. For the sake of brevity, we shall call k^i a (*forward*) ω-*node* and any (closed) path in C^i from k^i to itself whose t-label sequence is a maximum (C^i, k^i)-covering sequence for $\hat{m}(e^i)+t^+(e^i)$, a (*forward*) ω-*path* for k^i in $(R, \hat{m})$. Similarly, if the $\hat{m}$-labels within C^i contain strictly more ω-coordinates than does the label $\hat{m}(e^{i+1})$, we call k^i a *backward* ω-*node* and any (closed) path in C^i from k^i to itself which is the reversal of a maximum (C^i_{rev}, k^i)-covering sequence for $\hat{m}(e^{i+1})+t^-(e^{i+1})$, a *backward* ω-*path* for k^i in $(R, \hat{m})$.

LEMMA 2. *Suppose R is a regular constraint graph with consistent $\hat{m}$-labels. Let e^i be an edge on the base path of R entering the (forward) ω-node k^i, let $W = W(i)$ be the set of ω-coordinates of $\hat{m}(e^i)$, and let $W' = W'(i)$ be the set of ω-coordinates of the $\hat{m}$-labels in k^i's SCC C^i. Then there is a (closed) path γ from k^i to k^i such that the following properties hold*:

(i) *the sequence $\tau(\gamma)$ is firable at $\hat{m}(e^i)+t^+(e^i)$*;

(ii) *the ith coordinate of the effect $\tau(\gamma)$ is zero for all $i \in I_v - W'$, and it is positive for all $i \in W' - W$*;

(iii) *for every bound $H \in \mathbf{N}$, there is an $r \in \mathbf{N}$ such that the r-fold repetition $(\tau(\gamma))^r$ of $\tau(\gamma)$ is firable at $\hat{m}(e^i)+t^+(e^i)$, and such that, for all $i \in W'$, the ith coordinate of the resulting pseudomarking is $\geqq H$.*

Proof. Choose γ to be any (forward) ω-path in k^i. All properties claimed in the lemma are then immediate consequences of the definition of maximum covering sequences. □

It is clear that the statement analogous to Lemma 2 which takes into account direction reversal holds for backward ω-paths.

4. A sufficient condition for reachability. In this section, we show that regular constraint graphs with a consistent $\hat{m}$-labelling can be used to construct firing sequences from the initial marking m to the final marking m'.

As before, let R be a regular constraint graph for some Petri net $P=(S, T, K)$, and let $\hat{m}$ be a consistent edge labelling for R (with respect to two markings m and m' of P). Let $e^1, \cdots, e^p$ be the sequence of edges and $r=k^0, k^1, \cdots, k^p=q$ that of the nodes on the base path of R. Furthermore, let C^j be the SCC of R containing k^j, and let α and α' be two admissible paths which justify the $\hat{m}$-labels. Define, for $j=0, \cdots, p$, (closed) paths δ_j from k^j to k^j such that δ_j contains every edge within C^j just as many times as given by the "difference" between α' and α. By convention, these paths are trivial for $j=0$ and $j=p$. Because of the definition of α and α', the path δ_j is guaranteed to "cover" C^j, i.e., it contains every edge within C^j at least once. Since the indegree of the "difference" between α' and α is the same as its outdegree at every node in C^j, such paths δ_j exist. They can be constructed using an appropriate extension of an algorithm for Eulerian tours.

Let $\alpha^{(n)}$ be any path from r to q that contains every edge in R exactly as many times as does the multiset of edges consisting of α and n copies of all the δ_j. Clearly, $\alpha^{(n)}$ is an admissible path. In the following, we will show that, for sufficiently large n, there are paths $\alpha^{(n)}$ such that $\tau(\alpha^{(n)})$ is firable at m. It follows that $\tau(\alpha^{(n)})$ takes m to m'. We will find such paths by starting from some arbitrary path $\alpha^{(n)}$ which we then rearrange within every SCC of R. Note that for every $j=1, \cdots, p$, the intermediate marking $\hat{m}(e^1)+\hat{\tau}(\alpha_j^{(n)})$ does not depend on the specific choice of $\alpha^{(n)}$ as long as $\alpha^{(n)}$ is as defined above. As a matter of fact, the following relation holds:

$$\hat{m}(e^1)+\hat{\tau}(\alpha_j^{(n)})=m-t^-(e^1)+\hat{\tau}(\alpha_j)+n\sum_{k=1}^{j-1}\tau(\delta_j).$$

We may therefore look at each SCC C^j separately.

Let the notation be as above. We define, for $n \geqq 0$ and $j=0, \cdots, p$,

$$m^{(n)}(j)=_{\text{def}} m+\tau(\alpha_j^{(n)}),$$

and

$$m'^{(n)}(j)=_{\text{def}} m+\tau(\alpha_{j+1}^{(n)})-t(e^{j+1}),$$

which are, respectively, the marking defined by $\tau(\alpha^{(n)})$ for the moment when $\alpha^{(n)}$ first enters the SCC C^j, and the marking when it just leaves it. As argued above, these markings are well defined. Clearly $m^{(n)}(0)=m'^{(n)}(0)=m$ and $m^{(n)}(p)=m'^{(n)}(p)=m'$ for all n.

LEMMA 3. *Suppose $j \in I_{p-1}$. Then for all sufficiently large n, there is a closed path $\beta=\beta(j,n)$ in R from k^j to k^j such that*

(i) *the sequence $\tau(\beta)$ is firable at $m^{(n)}(j)$ and takes it to $m'^{(n)}(j)$;*

(ii) *the path β contains every edge within C^j exactly as many times as does the multiset of edges consisting of α and n copies of δ_j.*

Proof. For notational convenience, in this proof, we shall denote δ_j, $m^{(n)}(j)$, and $m'^{(n)}(j)$ simply by δ, $m^{(n)}$, and $m'^{(n)}$, respectively. We shall use $\bar{\alpha}$ for the maximal segment of the path α that lies within C^j. Note that $\bar{\alpha}$ starts and ends in k^j. Let I be a subset of I_v. We say, for some transition sequence τ and some pseudomarking $\bar{m} \in \bar{\mathbf{N}}^v$, that τ is I-firable at $\bar{m}$ if the projection of (the transitions of) τ onto the coordinates in I is a firing sequence for the projection of $\bar{m}$ onto the same coordinates. Clearly τ is firable at $\bar{m}$ iff it is I_v-firable.

Let W be the set of ω-coordinates of $\hat{m}(e^j)$ and W' that of $\hat{m}(e^{j+1})$. Let $\bar{W}$ be the set of ω-coordinates of the $\hat{m}$-label of any edge within C^j if C^j is nontrivial, and let $\bar{W}$ equal W otherwise (note that in the latter case $W = W'$). Also let γ_{up} be any forward ω-path for k^j, and let γ_{dn} be any backward ω-path for k^j, i.e., $(\gamma_{dn})_{\text{rev}}$ is a forward ω-path for k^j in $(R_{\text{rev}}, \hat{m})$. By convention, we define γ_{up} and γ_{dn} to be the empty path (consisting only of k^j) if $W = \bar{W}$ or $W' = \bar{W}$, respectively. Since the $\hat{m}$-labelling of R is consistent, the paths γ_{up} and γ_{dn} exist. Note that they are both closed paths from and to k^j. Since the (closed) path δ contains every edge within C^j at least once, there is another closed path γ_{lk} from k^j to k^j (lk for "link") such that the three paths γ_{up}, γ_{lk}, and γ_{dn} together contain every edge within C^j as often as do r_ω copies of δ, for some suitable integer $r_\omega \geqq 1$. From the definition of an ω-path we know that any iteration of $\tau(\gamma_{up})$ is $(I_v - W)$-firable at $m^{(n)}$, for every $n \geqq 0$. Furthermore, any iteration of $\tau((\gamma_{dn})_{\text{rev}})$ is $(I_v - W')$-firable at $m'^{(n)}$, again for every $n \geqq 0$. Since $\tau(\gamma_{up})$ increases all coordinates in $\bar{W} - W$ (as does $\tau((\gamma_{dn})_{\text{rev}})$ for all coordinates in $\bar{W} - W'$), we can by Lemma 2 choose some suitably large $s \in \mathbf{N}$ such that[4]

$$\tau(\gamma_{up}^s \bar{\alpha} \delta \gamma_{lk} \delta) \text{ is } (I_v - W)\text{-firable at } m^{(n)}$$

(for any $n \in \mathbf{N}$, because its coordinates in $I_v - W$ do not depend on n), and such that at the same time

$$(\tau(\bar{\alpha} \delta \gamma_{lk} \delta \gamma_{dn}^s))_{\text{rev}} \text{ is } (I_v - W')\text{-firable at } m'^{(n)}$$

(again for every $n \in \mathbf{N}$).

We claim that

$$\tau =_{\text{def}} \tau(\gamma_{up}^s \bar{\alpha} \delta \gamma_{lk}^s \delta \gamma_{dn}^s)$$

is $(I_v - (W \cup W'))$-firable at $m^{(n)}$, and that τ_{rev} is $(I_v - (W \cup W'))$-firable at $m'^{(n)}$. Set $I' =_{\text{def}} I_v - (W \cup W')$. Since $(m^{(n)} + \tau)_i = (m'^{(n)})_i$ for all $i \in I'$, it suffices to prove the first part of the claim. Note that the consistency conditions for the $\hat{m}$-labelling of R imply right away that τ is $(I_v - \bar{W})$-firable at $m^{(n)}$.

The sequence $\tau(\gamma_{up}^s \bar{\alpha} \delta)$ is I'-firable at $m^{(n)}$, and the sequence $\tau(\delta \gamma_{dn}^s)$ is I'-firable at $m'^{(n)} - \tau(\delta \gamma_{dn}^s)$ due to the choice of s. Consider the two markings $m^{(n)} + \tau(\gamma_{up}^s \bar{\alpha} \delta)$ and $m'^{(n)} - \tau(\delta \gamma_{dn}^s)$. If we restrict ourselves to the coordinates in I', we see above that the s-fold iteration of $\tau(\gamma_{lk})$ links the two markings together. Hence if $(\tau(\gamma_{lk}))^s$ were not I'-firable at $m^{(n)} + \tau(\gamma_{up}^s \bar{\alpha} \delta)$, then a violation of the nonnegativity condition for the intermediate markings (here restricted to the coordinates in I') would have to occur within the first or last copy of $\tau(\gamma_{lk})$. All other intermediate markings are linear interpolations of appropriate intermediate markings generated in the first and last copy of $\tau(\gamma_{lk})$. But by our construction, both copies contain no such violations. We conclude that the above sequence τ is therefore I'-firable at $m^{(n)}$, for arbitrary n. Note that τ has the same effect as given by $\tau(\bar{\alpha}) + (2 + sr_\omega)\tau(\delta)$. We should also like to point out here that the extended sequence $\tau(\gamma_{up}^s \bar{\alpha} \delta \gamma_{lk}^s)$ is not necessarily $(\bar{W} - W)$-firable at $m^{(n)}$ since some of these coordinates of $\tau(\gamma_{lk}^s)$ may have large negative values. A similar observation holds for $(\tau(\gamma_{lk}^s \delta \gamma_{dn}^s))_{\text{rev}}$.

[4] Remember that γ^r is the r-fold iteration of γ, i.e., $\gamma^1 = \gamma$, $\gamma^{r+1} = \gamma\gamma^r$.

However, we can choose $r\in\mathbf{N}$, $r\geqq sr_\omega$, big enough so that all of the following conditions are satisfied:

— the sequence $\tau(\gamma_{up}^s\bar{\alpha}\delta)$ is firable at $m^{(r)}$;

— the sequence $(\tau(\delta\gamma_{dn}^s))_{\text{rev}}$ is firable at $m'^{(r)}$;

— the marking $m''^{(r)}=_{\text{def}} m^{(r)}+\tau(\gamma_{up}^s\bar{\alpha})+\lfloor(r-sr_\omega)/2\rfloor\tau(\delta)$ is at least as big as the two prerequisites $\tau^-(\gamma_{lk}^s\delta)$ and $\tau^-(\delta_{\text{rev}})$.

Such a choice for r is possible because

(i) for all $i\in I_v-\bar{W}$, all pertinent conditions are already guaranteed by the consistency of the $\hat{m}$-labelling;

(ii) for all $i\in\bar{W}-(W\cup W')$, the ith coordinate of the effect $\tau(\delta)$ is zero, and because of the choice of s;

(iii) for all $i\in W$ (resp., W'), the ith coordinate of $m^{(r)}$ (resp., $m'^{(r)}$) increases with r; and therefore, finally, because

(iv) for all $i\in W\cup W'$ the ith coordinate of $m''^{(r)}$ increases with r.

Now we use the same linear interpolation argument as above. Since $\tau(\gamma_{up}^s\bar{\alpha}\delta)$ is firable at $m^{(r)}$ and since $\tau(\delta_{\text{rev}})$ is firable at $m''^{(r)}$, we conclude that in fact $\tau(\gamma_{up}^s\bar{\alpha}\delta^{\lfloor(r-sr_\omega)/2\rfloor})$ is firable at $m^{(r)}$. Similarly, because $\tau(\gamma_{lk}^s\delta)$ is firable at $m''^{(r)}$ and $(\tau(\delta\gamma_{dn}^s))_{\text{rev}}$ is firable at $m'^{(r)}$, we find that $\tau(\gamma_{lk}^s\delta^{\lceil(r-sr_\omega)/2\rceil}\gamma_{dn}^s)$ is firable at $m''^{(r)}$. Consequently we obtain that $\tau(\beta^{(r)})$, where

$$\beta^{(r)}=_{\text{def}}\gamma_{up}^s\bar{\alpha}\delta^{\lfloor(r-sr_\omega)/2\rfloor}\gamma_{lk}^s\delta^{\lceil(r-sr_\omega)/2\rceil}\gamma_{dn}^s,$$

is firable at $m^{(r)}$, and changes this marking to $m'^{(r)}$. The same argument actually shows that for every $n\geqq r$, $\tau(\beta^{(n)})$ is firable at $m^{(n)}$ and changes it to $m'^{(n)}$. □

THEOREM 1. *Suppose that there is a regular constraint graph R for the Petri net P with an $\hat{m}$-labelling which is consistent with respect to the two markings m and m' of P. Then m' is reachable from m.*

Proof. Choose n as in Lemma 3 sufficiently large for all SCC's of R and combine the paths within the SCC's with the edges on the base path of R. □

In the next section, we prove the following.

THEOREM 2. *Let P, m, and m' be given. There is an algorithm that constructs a regular constraint graph for P together with a consistent $\hat{m}$-labelling if $m\rightarrow^* m'$. Otherwise it determines that no such regular constraint graph exists.*

Combining Theorems 1 and 2 we therefore conclude

COROLLARY. *The general reachability problem for Petri nets and vector replacement systems is decidable.*

5. The construction of regular constraint graphs. In this section, we describe how to construct regular constraint graphs with consistent edge-labellings. Our algorithm performs a series of refinements on a given initial regular constraint graph (which is weakly consistent) until it obtains one with a consistent $\hat{m}$-labelling. However, if no such refinement is possible, the algorithm will determine this impossibility and stop. We present a nondeterministic algorithm to simplify the presentation. It will be clear from the description that whenever a nondeterministic step is performed, the number of possibilities for the step is finite, and hence the nondeterministic algorithm can be simulated by a deterministic algorithm enumerating all nondeterministic branches. When we prove the termination of the nondeterministic algorithm, we prove that in fact every nondeterministic branch of the algorithm terminates. A standard application of König's Infinity Lemma then implies the termination of the deterministic simulation.

We first introduce some additional notation. Suppose again that R is a regular constraint graph for the Petri net $P=(S,T,K)$, with $e^1,\cdots,e^p$ the sequence of edges

and $r = k^0, k^1, \cdots, k^p = q$ the sequence of nodes on its base path. Furthermore, let $e^{p+1}, \cdots, e^{p'}$ be the remaining edges of R (i.e., those within SCC's of R, in any order), and assume that $\hat{m}$ is an edge-labelling of R weakly consistent with respect to some fixed markings m and m' of P.

For every path α in R from r to q, we define the following $(p'-p+pv)$-dimensional integer vector:

$$\Phi'(\alpha) =_{\text{def}} (\Phi'(e^{p+1}, \alpha), \cdots, \Phi'(e^{p'}, \alpha), \hat{m}(e^1)+\hat{\tau}(\alpha_1), \cdots, \hat{m}(e^1)+\hat{\tau}(\alpha_p)).$$

The first $p'-p$ components $\Phi'(e^{p+i}, \alpha)$ give the number of occurrences of each edge e^{p+i} in the path α, while the next p blocks of v components each give the intermediate "marking" generated by $\tau(\alpha)$ when crossing the edge e^j on the base path of R, for $j = 1, \cdots, p$. (Note that for general α, these intermediate "markings" may have negative components.) From our discussion in § 2.2 and Definition 3, it follows that the set

$$AP(R) =_{\text{def}} \{\Phi'(\alpha); \alpha \text{ admissible path in } R\}$$

is an effectively constructible semilinear set. What is more, $AP(R)$ has the *slice property*, i.e., whenever x, $x+y$, $x+z \in AP(R)$ with $y, z \geqq 0$, then $x+y+z \in AP(R)$. A proof of this can be easily obtained by modifying the construction of Eulerian tours. We leave the details to the reader.

Now assume that α is a path such that $a =_{\text{def}} \Phi'(\alpha)$ is minimal in $AP(R)$. Because of the slice property of $AP(R)$, there is an admissible path α' such that

(i) $\Phi'(\alpha') \geqq a$, and

(ii) whenever $\Phi'(\alpha'') \geqq a$ for any admissible path α'', and the ith coordinate of $\Phi'(\alpha'')$ is strictly greater than the ith coordinate of a, then the ith coordinate of $\Phi'(\alpha')$ is strictly greater than the ith coordinate of $a = \Phi'(\alpha)$.

Informally, the path α' boosts the maximal set of coordinates that can be boosted by any path α'' with $\Phi'(\alpha'') \geqq a$.

DEFINITION 5. Let α, α', and $a = \Phi'(\alpha)$ be as above.

(i) The *count* $c_a(e^i)$ determined by a for the edge e^i of R, $i = p+1, \cdots, p'$, is

$$c_a(e^i) = \begin{cases} \Phi'(e^i, \alpha) & \text{if } \Phi'(e^i, \alpha) = \Phi'(e^i, \alpha'), \\ \omega & \text{otherwise.} \end{cases}$$

We extend this definition by setting $c_a(e^i) = 1$ for $i = 1, \cdots, p$.

(ii) The *restriction* $r_a(e^j)$ determined by a for $\hat{m}$-labels of the edges e^j, for $j = 1, \cdots, p$, is given by

$$(r_a(e^j))_i = \begin{cases} (\hat{m}(e^1)+\hat{\tau}(\alpha_j))_i & \text{if this is equal to } (\hat{m}(e^1)+\hat{\tau}(\alpha'_j))_i, \\ \omega & \text{otherwise.} \end{cases}$$

We also define $r_a(e^j)$ to consist of all ω for $j = p+1, \cdots, p'$.

Thus the count c_a is finite for exactly those edges of R that occur the same number of times in any admissible path α'' in R with $\Phi'(\alpha'') \geqq \Phi'(\alpha)$, and it then equals this number. An analogous statement holds for the restrictions $r_a(e)$ of edges e on the base path of R. For edges e' within SCC's of R, $r_a(e')$ is only defined for technical convenience and represents no actual restriction.

We next describe a procedure *refine*. Given a regular constraint graph R as above with labels $t(e^i)$, $\hat{m}(e^i)$, $r_a(e^i)$, and $c_a(e^i)$ attached to its edges $e^1, \cdots, e^{p'}$ (where the $\hat{m}$-labels are weakly consistent), *refine* constructs a refinement R' or R (i.e., there is a homomorphism from R' to R) that embodies the bounds given by r_a and c_a in a

sense made formal later. The program for *refine* is rather lengthy. It consists, however, of two very similar phases.

```
procedure refine (R, R');
begin
  co we use auxiliary node labellings m̃, z̃, and c̃; changes made to R are assumed to be local to refine oc
  co in a first phase, R is refined in order to account for c_a oc
  r := initial node of R; q := final node of R;
  let R' initially consist of one node r';
  z̃(r') := r; c̃(r') := (c_a(e^1), ···, c_a(e^p'));
  declare r' "unfinished";
  while there are "unfinished" nodes in R' do
    select at random some "unfinished" node k, declare it "finished"
    co this random selection does not influence the final R' oc;
    for each edge e = (z̃(k), z') in R with e's component c̃(k, e) > 0 do
      add a new edge e' = (k, k') to R', where k' is a new node;
      t(e') := t(e); m̂(e') := m̂(e); r_a(e') := r_a(e); z̃(k') := z'; c̃(k') := c̃(k) with e's coordinate reduced by 1;
      if there is some node k'' ≠ k' in R' such that c̃(k') = c̃(k'') and z̃(k') = z̃(k'') then
        identify k' with k'' else declare k' "unfinished" fi
    od
  od;
  if there is no node q' in R' with z̃(q') = q and a c̃(q') all of whose finite coordinates are zero then
    stop "R has no refinement as required" fi;
  nondeterministically select a simple path in R' from r' to some node q' with z̃(q') = q and c̃(q') such
  that all its finite coordinates are zero;
  attach to every node k on a copy of this simple path a copy of its SCC in R' such that it shares only the
    node k with the path; call the result R'';
  redefine R to be R'' with the edge labellings t, m̂, and r_a as inherited, and with no other labels;

  co in a second phase, the new R is now further refined taking into account the m̂- and r_a-labels; clearly,
    the m̂-labels are still weakly consistent oc
  r := initial node of R; q := final node of R; e^1 := edge leaving r in R;
  let R' initially consist of one node r';
  z̃(r') := r; m̃(r') := m̂(e^1) + t^-(e^1);
  declare r' "unfinished";
  while there are "unfinished" nodes in R' do
    select at random some "unfinished" node k, declare it "finished"
    co this random selection does not influence the final R' oc;
    for each edge e = (z̃(k), k') in R with t(e) firable at m̃(k) do
      if m̃(k) − t^-(e), m̂(e), and r_a(e) are pairwise compatible then
        add a new edge e' = (k, k') to R', where k' is a new node;
        t(e') := t(e); m̂(e') := min {m̃(k) − t^-(e), m̂(e), r_a(e)} co min componentwise oc;
        z̃(k') := z'; m̃(k') := the maximum (R, z')-cover of m̂(e') + t^+(e');
        if there is some node k'' ≠ k' in R' such that m̃(k') = m̃(k'') and z̃(k') = z̃(k'') then
          identify k' with k'' else declare k' "unfinished" fi
      fi
    od
  od;
  if there is no node q' in R' with z̃(q') = q then
    stop "R has no refinement as required" fi;
  nondeterministically select a simple path in R' from r' to some node q' with z̃(q') = q;
  attach to every node k on a copy of this simple path a copy of its SCC in R' such that it shares only the
    node k with the path; call the result R'';
  redefine R' to be R'' with the edge labellings t and m̂ as inherited, and with no other labels
end refine.
```

To see that the procedure *refine* terminates, first note that in both phases the mapping of an edge $e'=(k,k')$ in R' to the edge $e=(\tilde{z}(k),\tilde{z}(k'))$ in R with label $t(e')=t(e)$ provides a graph homomorphism from R' into R.

For the first phase, also note that for every SCC of R' (and hence for any SCC of the finally selected R''), this homomorphism is in fact one-to-one, because of the node-identification in the procedure. Moreover, while nodes in the same SCC of R' have equal $\tilde{c}$-values, those in different SCC's have different $\tilde{c}$-values, all of which are smaller than $\tilde{c}(r)$. Since there are only finitely many such values, the first phase of *refine* terminates.

For the second phase, we may restrict ourselves to showing that the preimage of any SCC of R under the above homomorphism is finite. Since the restrictions r_a within SCC's of R have only ω-components, the proof is similar to the demonstration of termination for Algorithm 1 in § 2.3. The r_a do not influence the construction.

We call the homomorphism from the output R' of *refine* into its input R the *refinement homomorphism*. We obtain it by a double application of the homomorphism given above. Note that under the refinement homomorphism, every edge e' is mapped to an edge e with $t(e')=t(e)$ and $\hat{m}(e') \leq \hat{m}(e)$. Extending a similar observation made above for the first phase of *refine*, we find

LEMMA 4. *Let R be a regular constraint graph with a weakly consistent $\hat{m}$-labelling. Suppose also that counts c_a and restrictions r_a for R are given, and that refine produces R' from R.*

(a) *Then R' is again a regular constraint graph with a weakly consistent $\hat{m}$-labelling.*

(b) *Let C' be any SCC of R', and let C be the SCC of C''s image under the refinement homomorphism. Also, let $W(C)$ (resp., $W(C')$) be the set of ω-coordinates of the $\hat{m}$-labels of edges within C (resp., C') (if C or C' is trivial we set $W(C)$ (resp., $W(C')$) equal to the empty set). Then*

(i) *$W(C')=W(C)$ and C' is isomorphic to a subgraph of C, or*

(ii) *$W(C')$ is a strict subset of $W(C)$.*

Proof. Part (a) of the lemma is obvious from the construction. As for part (b), we noted above that certainly $W(C')\subseteq W(C)$. If $W(C')=W(C)$ the node identification process in *refine* guarantees that C' is isomorphic to a subgraph of C. We note that this subgraph is a true subgraph of C if the count of some edge within C is finite. □

We are now ready to describe the final algorithm. Let as before $P=(S, T, K)$ be a Petri net, and let m and m' be markings for P. We may assume without loss of generality that $m \neq m'$, and that m' is not reachable from m by firing just one transition in T. These exceptional cases can be easily checked. We call an *initial regular constraint graph* for (P, m, m') any regular constraint graph R with a weakly consistent $\hat{m}$-labelling such that

(i) R has exactly three SCC's; and

(ii) the middle (and only nontrivial) SCC of R consists of a self-loop $e=(k^1, k^1)$ with $t(e)=t$ and $\hat{m}(e)=(\omega, \cdots, \omega)\in \bar{\mathbf{N}}^v$ for every $t\in T$.

Note that there are at most $|T|^2$ initial regular constraint graphs. The following algorithm basically refines an initial regular constraint graph in alternating directions until this refinement process stabilizes.

ALGORITHM 2.

nondeterministically select an initial regular constraint graph R;
repeat
  $LR := R$ **co** LR for *left-right* **oc**;
  determine $AP(LR)$;
  nondeterministically select a minimal $a \in AP(LR)$;
  determine c_a and r_a; attach them to LR's edges;
  refine(LR, R');

```
    RL := R'_rev co RL for right-left oc;
    determine AP(RL);
    nondeterministically select a minimal a ∈ AP(RL);
    determine c_a and r_a; attach them to RL's edges;
    refine(RL, R');
    R := R'_rev
until LR = R co only take into account the t- and m̂-labels oc;
output R
end Algorithm 2.
```

To prove termination of Algorithm 2 consider first one execution of the **repeat**-loop. Let LR be the regular constraint graph entering, and R the one leaving this execution of the loop. Choose any SCC C' of R, and let C be the SCC of LR into which C' gets mapped by a double application of the refinement homomorphism. Let nm (respectively, nm') be the number of ω-coordinates of the $\hat{m}$-labels of edges within C (respectively, C') and be zero if there are no such edges. Let ne (respectively, ne') be the number of edges within C and C', respectively. Applying Lemma 4 twice, we obtain that either (nm', ne') is lexicographically strictly smaller than (nm, ne), or that they are both the same and the preimage of C under the twofold application of the refinement homomorphism *is* C' and is isomorphic to C itself. Attach to every SCC of a regular constraint graph with a weakly consistent $\hat{m}$-labelling its corresponding pair (nm, ne). Then after one execution of the loop in Algorithm 2, each of these pairs either is unchanged (and then the corresponding SCC also does not change) or it is replaced by a finite multiset of pairs which are all strictly smaller in the lexicographic ordering than the one being replaced. Since the corresponding multiset-ordering is well-founded the algorithm must terminate [6].

LEMMA 5. *Let P, m, and m' be as above.*

(a) *If Algorithm 2 terminates normally, it outputs a regular constraint graph with a consistent $\hat{m}$-labelling.*

(b) *If m' is (nontrivially) reachable from m in P, then there is a computation of Algorithm 2 which terminates normally.*

Proof. Part (a) follows from the fact that the regular constraint graph finally output is stable under refinement in both directions. This fact provides for the existence of all the necessary forward and backward ω-paths. Stability under the refinement procedure in any one of the directions already gives the existence of justifying admissible paths.

Whenever there is a nontrivial firing sequence τ such that $m \to^{\tau} m'$, there is certainly an initial regular constraint graph for which τ is the label sequence of an admissible path. And in every refinement step, there is always a nondeterministic choice for which τ remains an admissible path in the refined regular constraint graph. This fact is clear from the specification of the nondeterministic choices in the algorithm and its subroutines, and from the definition of the count and restriction vectors. Hence there is a sequence of refinements which can be chosen by the algorithm and which always retains τ as the labelling sequence of an admissible path. As argued above, this sequence must terminate. □

The proof of Theorem 2 promised at the end of § 4 now immediately follows from Lemma 5.

6. Conclusion. The following is a list of some problems that have been shown to be effectively reducible ($\leqq$) or equivalent ($\equiv$) to the general reachability problem for Petri nets:

(a) ($\equiv$) the liveness problem for Petri nets [14];

(b) ($\equiv$) the zero marking reachability problem [14];

(c) (≡) the persistence problem for one transition [14];
(d) (≦) the containment and equivalence problem for sets of firing sequences [14];
(e) (≡) the word problem for commutative Semi-Thue-Systems [21];
(f) (≡) the word problem for PBLIND [10];
(g) (≡) the emptiness problem for the intersection of two Szilard languages [5];
(h) (≡) the recursiveness of subsets of finitely presented commutative semigroups compatible with relations with finitely many minimal elements [11].

As shown in [24], the reachability problem for Petri nets is exp-space hard. In [26], Petri nets with finite, nonprimitive recursive reachability sets are exhibited. It is not clear whether this implies that Algorithm 2 is also nonprimitive recursive in this case. Without the additional constraints derived from the count and restriction vectors for the first refinement, the first refined regular constraint graph would in effect enumerate the whole reachability set. However, we do not know so far how including these additional constraints affects the complexity of the algorithm. Also, it is still open whether Dickson's lemma [7] which is used implicitly several times, and which implies that every infinite sequence in $\mathbf{N}^v$ has an infinite nondecreasing subsequence, can be replaced by a different argument providing effective upper bounds for the case of marking sequences generated by Petri net transitions.

Other open problems concern the reachability sets of Petri nets, e.g.:
(a) Is there a live marking in $\mathcal{R}(P, m)$?
(b) Is there a marking m such that $\mathcal{R}(P, m)$ is live?
(c) Is $\mathcal{R}(P, m)$ semilinear?
(d) Is there a "small" bound $S_P(m, m')$ for $m' \in \mathcal{R}(P, m)$ such that m' is reachable via intermediate markings which are all bounded by $S_P(m, m')$?

It is hoped that the techniques shown in this paper can also be applied to other word problems (e.g., in monotone systems characterized by the property that transitions possible in some state are also possible in all "bigger" states) where no a priori upper bounds on the length of shortest derivations are known so far.

Acknowledgment. The author is very grateful for the many helpful suggestions by Alan Siegel and the anonymous referees (especially one of them) to improve the presentation.

REFERENCES

[1] T. ARAKI AND T. KASAMI, *Decidable problems on the strong connectivity of Petri net reachability sets*, Theoret. Comput. Sci., 4 (1977), pp. 99–119.
[2] H. G. BAKER, *Rabin's proof of the undecidability of the reachability set inclusion problem of vector addition systems*, MIT Project MAC, CSGM 79, Cambridge, MA, 1973.
[3] G. BERTHELOT AND G. ROUCAIROL, *Reduction of Petri nets*, in Mathematical Foundations of Computer Science 1976, Gdansk, A. Mazurkiewicz, ed., Springer, Berlin–Heidelberg–New York, 1976, pp. 202–209.
[4] F. COMMONER, *Deadlocks in Petri nets*, Applied Data Research, Wakefield MA, CA 7206-3211, 1972. Applied Data Research, Inc.
[5] S. CRESPI-REGHIZZI AND D. MANDRIOLI, *Petri nets and Szilard languages*, Information and Control, 33 (1977), pp. 177–192.
[6] N. DERSHOWITZ AND Z. MANNA, *Proving termination with multiset orderings*, Comm. ACM, 22 (1979), pp. 465–476.
[7] L. E. DICKSON, *Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors*, Amer. J. Math., 35 (1913), pp. 413–422.
[8] S. GINSBURG AND E. H. SPANIER, *Semigroups, Presburger formulas, and languages*, Pacific J. Math., 16 (1966), pp. 285–296.
[9] J. GRABOWSKI, *The decidability of persistence for vector addition systems*, IPL, 11 (1980), pp. 20–23.
[10] S. A. GREIBACH, *Remarks on blind and partially blind one-way multicounter machines*, Theoret. Comput. Sci., 7 (1978), pp. 311–324.

[11] M. HACK, *Petri nets and commutative semigroups*, MIT Project MAC, CSGN 18, Cambridge, MA, 1974.
[12] ———, *The recursive equivalence of the liveness problem and the reachability problem for Petri nets and vector addition systems*, Proc. of the 15th Annual Symposium on SWAT, New Orleans, LA, Oct. 1974, pp. 156–164.
[13] ———, *Decision problems for Petri nets and vector addition systems*, MIT Project MAC, MAC-TM 59, Cambridge, MA, 1975.
[14] ———, *Decidability questions for Petri nets*, MIT, LCS, TR 161, Cambridge, MA, 1976.
[15] O. HERZOG, *Liveness of extended control structure nets*, Univ. Utah, Salt Lake City, Dept. Computer Science, TR CSUU-77-107, August 1977.
[16] A. W. HOLT ET AL., *Final report of the information systems theory project*, Griffiss Air Force Base, Rome Air Development Center, NY, RADC-TR-68-305, 1968.
[17] A. W. HOLT AND F. COMMONER, *Events and conditions*, Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, Association for Computing Machinery, New York, 1970, pp. 3–52.
[18] J. HOPCROFT AND J. J. PANSIOT, *On the reachability problem for 5-dimensional vector addition systems*, Theoret. Comp. Sci., 8 (1979), pp. 135–159.
[19] N. D. JONES, L. H. LANDWEBER AND Y. E. LIEN, *Complexity of some problems in Petri nets*, Theoret. Comp. Sci., 4 (1977), pp. 277–299.
[20] R. M. KARP AND R. E. MILLER, *Parallel program schemata*, J. Comput. Systems Sci., 3 (May 1969), pp. 147–195.
[21] R. M. KELLER, *Vector replacement systems: a formalism for modelling asynchronous systems*, Princeton Univ., Princeton, NJ, CSL, TR 117, 1972.
[22] S. R. KOSARAJU, *Decidability of reachability in vector addition systems*, Proc. 14th Ann. ACM STOC, 1982, pp. 267–281.
[23] K. LAUTENBACH, *Exakte Bedingungen der Lebendigkeit für eine Klasse von Petri-Netzen*, GMD Bonn (St. Augustin), Bericht Nr. 82, 1973.
[24] R. LIPTON, *The reachability problem is exponential-space-hard*, Dept. Computer Science Rep. 62, Yale Univ., New Haven, CT, 1976.
[25] E. W. MAYR, *Einige Sätze über Umformungen und verklemmungsfreie Führbarkeit bei bewerteten Petrinetzen*, Technische Universität München, Institut für Informatik, Diplomarbeit, August 1975.
[26] E. W. MAYR AND A. R. MEYER, *The complexity of the finite containment problem for Petri nets*, J. Assoc. Comput. Mach., 28 (1981), pp. 561–576.
[27] E. W. MAYR, *Persistence of vector replacement systems is decidable*, Acta Informatica, 15 (1981), pp. 309–318.
[28] ———, *An algorithm for the general Petri net reachability problem*, Proc. 13th Ann. ACM STOC, 1981, Milwaukee, WI, pp. 238–246.
[29] R. E. MILLER, *A comparison of some theoretical models of parallel computation*, IEEE Trans. Comput., C-22 (1973), pp. 710–717.
[30] H. MÜLLER, *Decidability of reachability in persistent vector replacement systems*, Proc. 9th Symposium on MFCS, 1980, LNCS 88, Springer, New York, pp. 426–438.
[31] D. C. OPPEN, *A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic*, J. Comput. Systems Sci., 16 (1978), pp. 323–332.
[32] R. J. PARIKH, *On context-free languages*, J. Assoc. Comput. Mach., 13 (1966), pp. 570–581.
[33] J. L. PETERSON AND T. H. BREDT, *A comparison of models of parallel computation*, Proc. IFIP Congress 74, North-Holland, Amsterdam, 1974, pp. 466–470.
[34] C. A. PETRI, *Kommunikation mit Automaten*, Institut für Instrumentelle Mathematik (Bonn), Schriften des IMM Nr. 2, 1962.
[35] M. PRESBURGER, *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt*, Compte-Rendus du I. Congrès des Mathématiciens des pays Slavs, Warsaw (1930), pp. 92–101.
[36] G. SACERDOTE AND R. L. TENNEY, *The decidability of the reachability problem for vector addition systems*, Proc. 9th Ann. ACM STOC, 1977, pp. 61–76.
[37] J. VAN LEEUWEN, *A partial solution to the reachability problem for vector addition systems*, Proc. 6th Ann. ACM STOC, 1974, pp. 303–309.

SIAM J. COMPUT.
Vol. 13, No. 3, August 1984

© 1984 Society for Industrial and Applied Mathematics
002

QUANTITATIVE RELATIVIZATIONS OF COMPLEXITY CLASSES*

RONALD V. BOOK,† TIMOTHY J. LONG‡ AND ALAN L. SELMAN§

Abstract. Consider the following open problems:
(i) P = ? NP;
(ii) NP = ? co-NP;
(iii) P = ? PSPACE;
(iv) NP = ? PSPACE.
In this paper we study these four problems from a particular point of view. To illustrate our approach, consider the first problem. It is known that there exist recursive sets A and B such that $\mathrm{P}(A) = \mathrm{NP}(A)$ and $\mathrm{P}(B) \neq \mathrm{NP}(B)$. We study restrictions R on both the deterministic and also the nondeterministic polynomial time-bounded oracle machines such that the following holds: P = NP if and only if for every set A, $\mathrm{P}_R(A) = \mathrm{NP}_R(A)$. The restrictions are "quantitative" in the sense that the size of the set of strings queried by the oracle in computations of a machine on an input is bounded by a polynomial in the length of the input. We study several different ways of specifying such quantitative restrictions, each of which has the desired property.

Key words. complexity classes, restricted relativizations, bounding the size of the set of queries, P, NP, co-NP, PSPACE

1. Introduction. The fundamental issues of machine-based complexity theory such as "determinism vs. nondeterminism" and "time vs. space" have received renewed interest in the last decade due to the enormous effort that has been expended in investigating whether specific combinatorial problems are in P or are in NP or are NP-complete, etc. In the 1970's several attempts were made to solve the "P = ? NP" problem by applying techniques of recursive function theory. It seemed reasonable to assume that any diagonalization technique yielding P ≠ NP would be sufficiently general to yield, for every set A, $\mathrm{P}(A) \neq \mathrm{NP}(A)$, and that any simulation technique yielding P = NP would be sufficiently general to yield, for every set A, $\mathrm{P}(A) = \mathrm{NP}(A)$. However, Baker, Gill and Solovay [3] showed that there exist A and B such that $\mathrm{P}(A) = \mathrm{NP}(A)$ and $\mathrm{P}(B) \neq \mathrm{NP}(B)$, so that P = NP does not imply that for every A, $\mathrm{P}(A) = \mathrm{NP}(A)$, and P ≠ NP does not imply that for every A, $\mathrm{P}(A) \neq \mathrm{NP}(A)$.

The theme of the present paper is the study of the questions "P = ? NP," "NP = ? co-NP," "P = ? PSPACE" and "NP = ? PSPACE," by placing restrictions on deterministic and nondeterministic oracle machines operating in polynomial time or polynomial space and then considering the corresponding restricted relativized complexity classes. We develop "positive relativizations" of each of these questions; that is, we restrict the behavior of these oracle machines to obtain statements such as "P = NP if and only if for every set D, $\mathrm{P}_R(D) = \mathrm{NP}_R(D)$" where $\mathrm{P}_R(D)$ ($\mathrm{NP}_R(D)$) is the class of languages L such that $L \in \mathrm{P}(D)$ (resp., $L \in \mathrm{NP}(D)$) is witnessed by a deterministic (resp., nondeterministic) polynomial time-bounded oracle machine operating with restriction R.

* Received by the editors October 27, 1982, and in revised form July 6, 1983. This research was supported in part by the National Science Foundation under grants MCS80-11979, MCS81-20263 and MCS83-12472.

† Department of Mathematics, University of California at Santa Barbara, Santa Barbara, California 93106.

‡ Department of Computer Science, New Mexico State University, Las Cruces, New Mexico 88003.

§ Department of Computer Science, Iowa State University, Ames, Iowa 50010. The research of this author was performed while he visited the Department of Computer Science, Technion, Haifa, Israel, with funds provided by the United States–Israel Education Foundation (Fulbright Award).

To motivate our approach, consider the oracle property "$x \in L(A)$ if and only if A contains a string of length $|x|$" that is used in [3] in order to separate P(A) from NP (A) for some oracle set A. A nondeterministic oracle machine can accept $L(A)$ relative to A because it can nondeterministically search a set of size $2^{|x|}$. Intuition suggests that a deterministic oracle machine cannot perform this search in polynomial time, and the proof in [3] bears this out. We want the restrictions on our oracle machines to retain the combinatorial difficulties of the "P = ? NP" question. Such considerations indicate that we reach the borderline of these kinds of classes by permitting nondeterministic oracle machines to search sets of size at most a polynomial in the length of the input.

We illustrate our case with the following result. For a nondeterministic oracle machine M, let $Q(M, A, x)$ denote the set of all strings queried in the entire tree of computations of M on input x with oracle set A. Let $\mathrm{NP}_B(A)$ be the class of languages accepted relative to A by nondeterministic polynomial time-bounded oracle machines M such that there is a polynomial q (depending on M and A) such that for all input strings x, the number of distinct strings in $Q(M, A, x)$ is at most $q(|x|)$. In Theorem 4.5 we show that NP = co-NP if and only if for every set A, $\mathrm{NP}_B(A) = \text{co-NP}_B(A)$. In Theorem 5.3 we show that P = NP if and only if for every set A, $\mathrm{P}(A) = \mathrm{NP}_B(A)$. Observe that this constraint on the set of strings queried is weak: the computation tree of M on input x relative to A may still be exponential in size.

The type of controlled relativization exemplified by $\mathrm{NP}_B(\ \ )$ is "quantitative" in the sense that it is the *size* of the set of queries that is bounded. Other quantitative relativizations are studied here as well.

As a consequence of the results cited above, one approach to proving P $\neq$ NP is to construct a set A such that $\mathrm{P}(A) \neq \mathrm{NP}_B(A)$. If one attempts to construct such an A by diagonalization, then it is clear that one wants the class $\mathrm{NP}_B(A)$ to be as large as possible so as to be able to separate it from P (A). Among the restrictions considered here, $\mathrm{NP}_B(\ \ )$ is the weakest restriction for which the Baker–Gill–Solovay phenomenon does not occur.

Basic notation and definitions are established in § 2. In § 3 we define several complexity classes of functions computable deterministically or nondeterministically in polynomial time, and we develop relationships between these function classes and the language classes P and NP. These relationships are used extensively in the proofs of our major results. In § 4 a number of quantitative relativizations of NP are introduced in order to study the "NP = ? co-NP" question, while in § 5 these same relativizations are used to study the "P = ? NP" question. Section 5 contains the main result, P = NP if and only if for every set D, $\mathrm{P}(D) = \mathrm{NP}_B(D)$. The various quantitative relativizations considered in §§ 4 and 5 are proved to be distinct from one another in § 6.

Ladner, Lynch and Selman [15] showed that in the case of deterministic polynomial-time oracle machines, if one considered the restricted class where the entire set of queries could be computed before any questions were asked of the oracle, then this relativization characterized deterministic polynomial-time truth-table reducibility. In § 7 we investigate the quantitative relativizations of P and NP with the further restriction that the set of queries be computable before any questions are asked of the oracle and develop more positive relativizations of the "P = ? NP" question.

While there have been several studies of oracle machines with varying abilities to use the oracle [2], [11], [12], [14], [19], the first positive relativizations were developed in the study of the "NP = ? PSPACE" question. In this case the restriction on nondeterministic polynomial space-bounded oracle machines was that the number of times the oracle could be queried was bounded by a polynomial [5], [7]. Subsequently, it was

found that this same restriction yields a positive relativization of the "P = ? PSPACE" question [23]. (Other results using this type of restriction are reported in [6].) Section 8 is devoted to additional positive relativizations of the "P = ? PSPACE" and "NP = ? PSPACE" questions using the notions of quantitative relativizations. Finally, technical questions and directions for further research are discussed in § 10.

In this paper we bring together a number of concepts and proof techniques that extend and generalize methods used in previous papers [5]–[7], [17]. Clearly these methods can be used to study questions about other complexity classes, but the techniques will be basically the same as those developed here. However, one question for which these techniques appear to fail is that of "P = ? NP ∩ co-NP." We have succeeded in developing a positive relativization of this question by restricting the "pattern of queries" that a nondeterministic oracle machine may make in its computations. We refer to such restrictions as "qualitative." An exposition of the properties of qualitative relativizations is in preparation.

2. Preliminaries. It is assumed that the reader is familiar with the basic concepts from the theories of automata, computability and formal languages. Some of the concepts that are most important for this paper are reviewed here and notation is established.

For a string w, $|w|$ denotes the length of w. The empty string is denoted by e, $|e| = 0$.

For a set S, $\|S\|$ denotes the cardinality of S.

It is assumed that all sets of strings are taken over some fixed alphabet Σ that includes $\{0, 1\}$. If $A \subseteq \Sigma^*$, then $\bar{A} = \Sigma^* - A$.

Let $<$ denote any standard polynomial time computable total order defined on Σ^*. For a finite set $S \subset \Sigma^*$, say $S = \{y_1, \cdots, y_n\}$ where $i < j$ implies $y_i < y_j$, let $c(S) = \%y_1\% \cdots \%y_n\%$ where % is a symbol not in Σ. Let $c(\phi) = \%$. We consider c to be an encoding function. Notice that if $S \in \Sigma^*$ is a finite set and $y \in \Sigma^*$, then the predicate "y is in S" can be computed in polynomial time from the inputs y and $c(S)$.

For sets $A, B \subseteq \Sigma^*$, the *join* of A and B is defined as $A \oplus B = \{0x | x \in A\} \cup \{1y | y \in B\}$.

An *oracle machine* is a multitape Turing machine M with a distinguished work tape, the *query* tape, and three distinguished states QUERY, YES, and NO. At some step of a computation on an input string w, M may transfer into the state QUERY. In state QUERY, M transfers into the state YES if the string currently appearing on the query tape is in some *oracle set* A; otherwise, M transfers into the state NO; in either case the tape is instantly erased. The set of strings *accepted by M relative to the oracle set* A is $L(M, A) = \{w|$there is an accepting computation of M on input w when the oracle set is $A\}$. If M has no query tape, we write $L(M)$ instead of $L(M, \phi)$.

Oracle machines may be deterministic or nondeterministic. An oracle machine may operate within some time bound T, where T is a function of the length of the input string, and the notation of operation within a time bound for an oracle machine is just the same as that for an ordinary Turing machine. An oracle machine may operate within some space bound S, where S is a function of the length of the input string, and here we require that the query tape as well as the ordinary work tapes be bounded in length by S. For other notions of space-bounded oracle machines, see [2], [14], [19].

For any space bound S, where $S(n) \geqq n$, and any oracle set A, let NSPACE (S, A) (DSPACE (S, A)) be the class of languages accepted relative to A by nondeterministic (resp., deterministic) oracle machines that operate within space bound $S(n)$. Let NSPACE (S) = NSPACE (S, ϕ) and DSPACE (S) = DSPACE (S, ϕ). For any time bound T, where $T(n) \geqq n$, and any oracle set A, let NTIME (T, A) (DTIME (T, A))

be the class of languages accepted relative to A by nondeterministic (resp., deterministic) oracle machines that operate within time bound $T(n)$. Let NTIME $(T)=$ NTIME (T, ϕ) and DTIME $(T)=$ DTIME (T, ϕ).

If $\mathcal{S}$ is a set of space bounds and A is an oracle set, let DSPACE $(\mathcal{S}, A) = \bigcup\{\text{DSPACE}(S, A) \mid S \in \mathcal{S}\}$ and NSPACE $(\mathcal{S}, A) = \bigcup\{\text{NSPACE}(S, A) \mid S \in \mathcal{S}\}$, and let DSPACE $(\mathcal{S})=$ DSPACE $(\mathcal{S}, \phi)$ and NSPACE $(\mathcal{S})=$ NSPACE $(\mathcal{S}, \phi)$. If $\mathcal{T}$ is a set of time bounds and A is an oracle set, let DTIME $(\mathcal{T}, A) = \bigcup\{\text{DTIME}(T, A) \mid T - \mathcal{T}\}$ and NTIME $(\mathcal{T}, A) = \bigcup\{\text{NTIME}(T, A) \mid T \in \mathcal{T}\}$, and let DTIME $(\mathcal{T})=$ DTIME $(\mathcal{T}, \phi)$ and NTIME $(\mathcal{T})=$ NTIME $(\mathcal{T}, \phi)$.

Let PSPACE $(A) = \bigcup_{k \geqq 1}$ DSPACE (n^k, A). It is known [21], [24] that for every oracle set A and all $k \geqq 1$, NSPACE $(n^k, A) \subseteq$ DSPACE (n^{2k}, A) so that PSPACE $(A) = \bigcup_{k \geqq 1}$ NSPACE (n^k, A). Let PSPACE $=$ PSPACE (ϕ).

Let P $(A) = \bigcup_{k \geqq 1}$ DTIME (n^k, A) and NP $(A) = \bigcup_{k \geqq 1}$ NTIME (n^k, A). Let P $=$ P (ϕ) and NP $=$ NP (ϕ).

It will be useful to refer to an enumeration of classes such as NP (A), and to have universal sets for certain subclasses of NP (A). Hence, we assume the existence of an effective enumeration of clocked nondeterministic oracle machines that run in polynomial time, say $M(0), M(1), \cdots$. Thus, for any sets L and D, $L \in$ NP (D) if and only if there is some i such that $L = L(M(i), D)$. We lose no generality by assuming that for every i, machine $M(i)$ runs in time $p_i(n) = n^i + i$. Let $\langle \cdot, \cdot \rangle$ denote a fixed polynomial time computable pairing function with polynomial time computable inverses. We define the set K as follows: For machines $M(i)$, inputs x, finite sets T_Y, T_N of strings, and integers k, let $\langle M(i), x, c(T_Y), c(T_N), 0^k \rangle \in K$ if and only if some computation of machine $M(i)$ on input x accepts x in at most k steps, and if y is a string that is queried in this computation, then $y \in T_Y \cup T_N$ and the answer to the query is "yes" if and only if $y \in T_Y$. It is clear that $K \in$ NP. Also, it should be clear that this set K stands in strong analogy to the standard diagonal halting set that is complete for the class of recursively enumerable sets, and that the two sets are not the same. No confusion should arise since we consider only subrecursive classes.

The set K will be used in the following situation. Let $M(i)$ be the ith nondeterministic polynomial time oracle machine. Suppose that on input x the set of strings queried in $M(i)$'s computations on x relative to set A is $T_Y \cup T_N$ and $(T_Y \cup T_N) \cap A = T_Y$. Then $x \in L(M(i), A)$ if and only if $\langle M(i), x, c(T_Y), c(T_N), 0^{p_i(|x|)} \rangle$ is in K. In this sense K is a universal set.

3. Computing functions. In this section we develop some results about functions computed nondeterministically in polynomial time. These results will be used as tools in later sections.

A nondeterministic Turing transducer is a nondeterministic Turing machine with distinguished accepting states and a distinguished output tape. A transducer T computes a value y on an input string x if there is an accepting computation of T on x for which y is the final contents of T's output tape. In general, a nondeterministic transducer computes a partial, multivalued function.

Given a partial, multivalued function f, define set-f by set-$f(x) = \{y \mid y \text{ is a value of } f(x)\}$, for all x. If $\|\text{set-}f(x)\|$ is finite for each x, then the function $c(\text{set-}f)$ is defined by $c(\text{set-}f)(x) = c(\text{set-}f(x))$. For each x, $c(\text{set-}f)(x)$ is a string encoding of the set of all words y such that y is a value of $f(x)$. Also, note that $c(\text{set-}f)$ is a single-valued total function defined on Σ^*.

DEFINITION 3.1. (a) NPMV is the set of all partial, multivalued functions computed by nondeterministic polynomial time-bounded transducers.

(b) NPSV is the set of all $f \in$ NPMV that are single-valued.
(c) NPMVPB is the set of all $f \in$ NPMV such that for some polynomial p and all x, $\|\text{set-}f(x)\| \leqq p(|x|)$.
(d) PF is the set of all partial single-valued functions computed by deterministic polynomial time-bounded transducers.

Clearly, PF $\subseteq$ NPSV $\subseteq$ NPMVPB $\subseteq$ NPMV. Let "NPMVPB $\subseteq_c$ PF" denote the assertion that "for every $f \in$ NPMVPB, $c(\text{set-}f)$ is in PF," and let "NPMVPB $\subseteq_c$ NPSV" denote the assertion that "for every $f \in$ NPMVPB, $c(\text{set-}f) \in$ NPSV." (Of course, there is no expectation that either of these assertions is true.) It should be clear that we are simply making useful conventions and not set-theoretic inclusions.

PROPOSITION 3.2. *If* NP = co-NP, *then* NPMVPB $\subseteq_c$ NPSV.

Proof. Let T witness $f \in$ NPMVPB and let q be a polynomial such that for all x, $\|\text{set-}f(x)\| \leqq q(|x|)$. Define the set ACC to be the set of all accepting configurations of T. Define the set OKCON as follows: $\langle x, c(S), I\rangle$ is in OKCON, where x is an input word to T, I is a configuration of T, and S is a finite set, if and only if there is a computation of T starting from configuration I that outputs a string $y \in \text{set-}f(x)$ which is not in the finite set S. Note that ACC $\in$ P and OKCON $\in$ NP.

Without loss of generality, 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 computes $c(\text{set-}f)$. The basic idea is to implement a depth-first search of the computation tree of T on input x. When a configuration I is visited, OKCON is used to determine which of the successors, left (I) or right (I), is to be visited next.

ORACLE PROCEDURE 3.1.

```
    begin
    input x;
    S := φ;
    I_0 := initial configuration of T on x;
(1) while ⟨x, c(S), I_0⟩ ∈ OKCON do
      begin {S is a proper subset of set-f(x)}
      I := I_0;
(2)   while I ∉ ACC do
        {simulate computation of T}
(3)     if ⟨x, c(S), left (I)⟩ ∈ OKCON
          then I := left (I)
          else I := right (I);
      S := S ∪ [the string on the output tape of configuration I]
      end;
(4) halt in accepting state with c(S) on the output tape
    end.
```

Since OKCON $\in$ NP and NP = co-NP is assumed, there exist NP machines N_1 and N_2 that accept OKCON and $\neg$OKCON, respectively. The test on line 1 is implemented by simultaneous executions of N_1 and N_2 on input values $\langle x, c(S), I_0\rangle$. Line 1 becomes true if N_1 accepts and becomes false if N_2 accepts. If N_1 and N_2 both fail to accept a common input, the procedure terminates. Line 3 is implemented the same way. Thus, for every input word x, there is a computation that leads to line 4.

When execution of the outer while-loop terminates, $S = \text{set-}f(x)$; i.e., the procedure nondeterministically computes $c(\text{set-}f)$. To see this, note that line 2 is reached

only if there is a string $y \in$ set-$f(x)$ that has not yet been found and that the inner while-loop preserves this property.

Now let us observe that this nondeterministic procedure runs in polynomial time. We have already seen that the test on lines 1 and 3 take polynomial time and this is clearly true for the test on line 2. Furthermore, the outer while-loop is executed at most $q(|x|)$ times and, for each execution of the outer loop, the inner while-loop executes at most $p(|x|)$ times, where p is a polynomial bound on the running time of T. Thus, we conclude $c(\text{set-}f) \in \text{NPSV}$. □

Here is an example that indicates how the technical results of this section are to be used in the forthcoming sections. Let M be a nondeterministic oracle machine that operates in polynomial time and let $Q(M, x)$ denote the set of words queried in all possible computations of M on x, i.e., relative to all oracle sets. Suppose that for some polynomial q and for all x, $\|Q(M, x)\| \leqq q(|x|)$. Then, for any oracle set D one could nondeterministically recognize $L(M, D)$ by first constructing "tables" $T_Y = Q(M, x) \cap D$ and $T_N = Q(M, x) \cap \bar{D}$ upon input x and then simulating M without further use of D. Let us not be concerned with the latter simulation at this point, but rather consider how the tables can be constructed (as they are in Theorem 4.5A under the hypothesis NP = co-NP and again in Theorem 5.3A under the hypothesis P = NP). Simply let f be the multivalued function such that for all x, set-$f(x) = Q(M, x)$ and observe that according to our assumption $f \in$ NPMVPB. Then, some function $g \in$ NPSV computes $c(\text{set-}f)$ if NP = co-NP, by Proposition 3.2, and, if P = NP, then, using Proposition 3.3 below, some function $h \in$ PF computes $c(\text{set-}f)$. Thus, tables T_Y and T_N can be constructed in polynomial time relative to D, nondeterministically if NP = co-NP, and deterministically if P = NP.

PROPOSITION 3.3. *The following are equivalent*:

(a) P = NP.

(b) NPSV $\subseteq$ PF.

(c) NPMVPB $\subseteq_c$ PF.

Proof. Certainly, (c) implies (b). For each set L in NP, the function f defined by $f(x) = 1$ if $x \in L$, and $f(x)$ is undefined otherwise, is single-valued. Hence, (b) implies (a). Thus, it is sufficient to show that (a) implies (c). This follows from the proof of Proposition 3.2. Namely, if P = NP is assumed, then OKCON is in P. Therefore, each test in the procedure is executable deterministically in polynomial time. The conclusion is that $c(\text{set-}f) \in$ PF. □

We will also want to consider partial multivalued functions computed by nondeterministic oracle Turing transducers. With reference to Definition 3.1, for any set D, the classes NPMV (D), NPSV (D), NPMVPB (D) and PF (D) are defined in the obvious way. Clearly, PF $(D) \subseteq$ NPSV $(D) \subseteq$ NPMV $(D) \subseteq$ NPMVPB (D), for every set D.

PROPOSITION 3.4. *For any multivalued function f of one argument, define the single-valued function g as follows*:

$$g(x, 0^k) = \begin{cases} c(\text{set-}f(x) & \text{if } \|\text{set-}f(x)\| \leqq k, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Suppose that f is in NPMV. *Then, the following implications hold*:

(i) *If* NP = co-NP, *then* $g \in$ NPSV *and* domain $(g) \in$ NP $\cap$ co-NP.

(ii) *If* co-NP $\subseteq$ NP (D) *for some set D, then* $g \in$ NPSV (D) *and* domain $(g) \in$ NP $(D) \cap$ co-NP (D).

(iii) *If* P = NP, *then* $g \in$ PF *and* domain $(g) \in$ P.

(iv) *If* NP $\subseteq$ P (D) *for some set D, then* $g \in$ PF (D) *and* domain $(g) \in$ P (D).

Proof. A minor modification of the procedure used in the proof of Proposition 3.2 gives the results.

ORACLE PROCEDURE 3.2.

```
    begin
    input x and 0^k;
    S := φ;
    j := 0;
    I_0 := initial configuration of T on x;
(1) while ⟨x, c(S), I_0⟩ ∈ OKCON and j ≦ k do
      begin {j = ||S||}
      I := I_0;
      while I ∉ ACC do
(2)     if ⟨x, c(S), left (I)⟩ ∈ OKCON
          then I := left (I)
          else I := right (I);
      S := S ∪ [string on output tape of configuration I];
      j := j + 1
      end;
(3) if j ≦ k
      then halt in an accepting state with c(S) on the output tape,
      else halt in a nonaccepting state
    end.
```

If either NP = co-NP or P = NP is assumed, then the proofs of Propositions 3.2 and 3.4 show that for each input x there is a computation of the procedure that reaches line 3. Using the loop invariant "$j = \|S\|$," it follows that the procedure accepts x and outputs $c(\text{set-}f(x))$ if and only if $\|\text{set-}f(x)\| \leqq k$. Thus, the procedure correctly computes g. If NP = co-NP, then $g \in$ NPSV, and if P = NP, then $g \in$ PF.

Recall that OKCON $\in$ NP. Therefore, if co-NP $\subseteq$ NP (D) for some D, then $\neg$OKCON $\in$ NP (D). Hence, the tests on lines 1 and 2 can be implemented by simultaneous executions of nondeterministic oracle machines relative to the set D that accept OKCON and $\neg$OKCON. Therefore, co-NP $\subseteq$ NP (D) implies $g \in$ NPSV (D).

Similarly, if NP $\subseteq$ P (D) for some set D, then OKCON $\in$ P(D), and $g \in$ PF (D) follows.

The remaining assertions are trivial. If $g \in$ NPSV, then domain $(g) \in$ NP, and since in clause (i) NP = co-NP is assumed, domain $(g) \in$ NP $\cap$ co-NP. Also, $g \in$ NPSV (D) (PF, PF (D) resp.) implies domain $(g) \in$ NP $(D) \cap$ co-NP (D) (P, P (D), resp.). □

Observe that in the proof of (iv), nondeterminism occurs only in the tests that direct the search. For this reason the procedure yields something stronger than domain $(g) \in$ NP (D), namely domain $(g) \in$ P $(K \oplus D)$. Therefore, under the hypothesis NP $\subseteq$ P (D), it follows that $K \in$ P (D) and so domain $(g) \in$ P (D). (Recall that NP $\subseteq$ P (D) means that D is $\leqq_T^{\text{P}}$-hard for NP.) This point of view will be useful in later sections.

With respect to parts (ii) and (iv) of Proposition 3.4, recall that co-NP $\subseteq$ NP (D) means that D is $\leqq_T^{\text{NP}}$-hard for NP. In fact, as the following argument shows, D is $\leqq_T^{\text{SN}}$-hard for NP:

co-NP $\subseteq$ NP (D) if and only if $\Sigma_2^{\text{P}} \subseteq$ NP (D), if and only if NP (SAT) $\subseteq$ NP (D), if and only if SAT $\leqq_T^{\text{SN}} D$, and if and only if D is $\leqq_T^{\text{SN}}$-hard for NP.

4. NP = ? co-NP. In this section we introduce new, restricted relativizations of NP. These restrictions are quantitative in the sense that the set of strings potentially queried by the oracle machine is explicitly bounded.

DEFINITION 4.1. Let M be an oracle machine. (a) For any set D and any input string x of M, let $Q(M, D, x)$ (resp., $QA(M, D, x)$) be the set of strings y such that in some computation (resp., accepting computation) of M relative to D on input x, the oracle is queried about y.

(b) For any input string x, let $Q(M, x) = \bigcup_D Q(M, D, x)$ and $QA(M, x) = \bigcup_D QA(M, D, x)$.

In some of our constructions, it will be convenient to further subdivide the set $Q(M, D, x)$.

DEFINITION 4.2. Let M be an oracle machine. For any set D, any input string x and any integer $k > 0$, let $Q(M, D, x, k)$ be the subset of $Q(M, D, x)$ such that $y \in Q(M, D, x, k)$ if and only if there is a computation of M relative to D on input x that queries the oracle at least k times, and at the kth 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 D on input x, $Q(M, D, x, 1)$ is the set of strings queried the first time M reaches a query configuration, $Q(M, D, x, 2)$ is the set of strings queried the second time M reaches a query configuration, etc.

Let us consider for a moment the standard machine model NP $(\cdot)$ together with the well-known construction of Baker, Gill and Solovay [3] of a set D such that P $(D) \neq$ NP (D). They apply the oracle property "$x \in L(D)$ if and only if D contains a string of length $|x|$." For any set D, $L(D)$ can be recognized relative to D by a machine M that on reading x nondeterministically guesses a string y such that $|y| = |x|$ and then queries the oracle for D about y. In this case $Q(M, D, x)$ has size $2^{|x|}$; by "guessing," M can nondeterministically search a set of size $2^{|x|}$. Intuition says that a deterministic oracle machine cannot perform this search in polynomial time and the proof in [3] bears this out. In order to eliminate this difference, we consider here restrictions of the size of the sets $Q(M, D, x)$ and $QA(M, D, x)$.

DEFINITION 4.3. Let D be a set. Define the following classes:

(a) NP.ALL (D) is the class of languages L such that $L \in$ NP (D) is witnessed by a machine M such that, for some polynomial q and all x, $\|Q(M, x)\| \leqq q(|x|)$;

(b) NP.ACC (D) is the class of languages L such that $L \in$ NP (D) is witnessed by a machine M such that, for some polynomial q and all x, $\|QA(M, x)\| \leqq q(|x|)$;

(c) NP.ALL.DEP (D) is the class of languages L such that $L \in$ NP (D) is witnessed by a machine M such that, for some polynomial q and all x, $\|Q(M, D, x)\| \leqq q(|x|)$;

(d) NP.ACC.DEP (D) is the class of languages L such that $L \in$ NP (D) is witnessed by a machine M such that, for some polynomial q and all x, $\|QA(M, D, x)\| \leqq q(|x|)$.

For some choices of M and D, the size of $Q(M, D, x)$ may be polynomially bounded while the size of $Q(M, x)$ is not.

For any set D four classes of languages are specified by Definition 4.3. Each of the classes arises from a restriction of the class of nondeterministic polynomial time-bounded oracle machines. In each case the restriction is based on the machine's computation trees. The notation NP.ALL $(\cdot)$ refers to the set $Q(M, x)$ of *all* strings queried in the computation tree containing all computations of M on x independent of the choice of oracle. The notation NP.ACC ( ) refers to the set $QA(M, x)$ of strings queried in the *accepting* computations independent of the choice of oracle. In contrast the classes NP.ALL.DEP (D) and NP.ACC.DEP (D) refer to the sets $Q(M, D, x)$ and $QA(M, D, x)$, respectively, that *depend* on the set D.

The classes NP.ALL.DEP (D) are the classes referred to in the Introduction as $\mathrm{NP}_B(D)$. Not all of the classes defined here are distinct.

LEMMA 4.4. *For every set D,* NP.ALL $(D) =$ NP.ACC (D).

Proof. For every M and x, $QA(M, x) \subseteq Q(M, x)$, so it is clear that for every set D, NP.ALL $(D) \subseteq$ NP.ACC (D). For the converse, let M_1 witness $L \in$ NP.ACC (D).

Define the set OKCON as follows: $\langle I, S\rangle \in$ OKCON if and only if I is a configuration of M_1, S is a finite set of words of the form $\langle w, b\rangle$, where $w \in \Sigma^*$, $b \in \{0, 1\}$, and $\langle w, 1\rangle \in S$ implies that $\langle w, 0\rangle \notin S$, and there is an accepting computation of M_1 that begins with I so that for every w, if w is queried during this computation and if for some b, $\langle w, b\rangle \in S$, then the return state is YES if and only if $b = 1$. Clearly, OKCON is in NP; let M_0 witness OKCON $\in$ NP.

Now we describe a nondeterministic oracle machine M_2 such that $L(M_2, D) = L(M_1, D)$ and $Q(M_2, x) = QA(M_1, x)$. One of M_2's work tapes is used to store (an appropriate encoding of) S and this tape is initially empty. On input x, M_2 begins a simulation of some computation of M_1 on input x. Whenever this simulated computation is to enter a query configuration, say I, M_2 first simulates some computation of M_0 on I and S, where S is determined by the current value of M_2's designated work tape. If this computuation accepts so that $\langle I, S\rangle \in$ OKCON, then M_2 continues its simulation of M_1 from configuration I in the following manner. Let w be the word to be queried in configuration I. If for some b, $\langle w, b\rangle$ belongs to S already, then simulation of M_1 continues in state YES if $b = 1$ and in state NO if $b = 0$. If $\langle w, b\rangle$ does not belong to S for any b, then M_2 nondeterministically chooses a transfer state from configuration I and writes the pair $\langle w, b\rangle$ on its designated tape, where $b = 1$ if the transfer state is YES and $b = 0$ otherwise.

Clearly, $L(M_2, D) = L(M_1, D)$ and M_2 operates nondeterministically in polynomial time. In a computation on input x, M_2 queries the oracle only if the current query configuration could potentially lead to an accepting computation and S is maintained to insure consistency of oracle responses along this computation. Thus, $Q(M_2, x) = QA(M_1, x)$ so that the size of $Q(M_1, x)$ is polynomial-bounded since the size of $QA(M_1, x)$ is so bounded. This means that M_2 witnesses $L \in$ NP.ALL (D). □

Thus, for each set D there are at most three different restricted classes. Clearly, NP.ALL $(D) \subseteq$ NP.ALL.DEP $(D) \subseteq$ NP.ACC.DEP $(D) \subseteq$ NP (D). We will see in Theorems 6.1 and 6.4 that there are recursive sets E and F such that NP.ALL $(E) \neq$ NP.ALL.DEP (E) and NP.ALL.DEP $(F) \neq$ NP.ACC.DEP (F). Note that NP.All (ϕ) = NP.ALL.DEP (ϕ) = NP.ACC.DEP (ϕ) = NP (ϕ) = NP.

Now we have our first result, two positive relativizations of the "NP = ? co-NP" problem.

THEOREM 4.5. A. NP = co-NP *if and only if for every set* D, NP.ALL (D) = co-NP.ALL (D).

B. NP = co-NP *if and only if for every set* D, NP.ALL.DEP (D) = co-NP.ALL.DEP (D).

Before proving Theorem 4.5, we repeat an observation made in § 3. If a nondeterministic polynomial time-bounded oracle machine M has the property that, for some polynomial q and all x, $\|Q(M, x)\| \leq q(|x|)$, then for any set D one can nondeterministically recognize $L(M, D)$ by first constructing the tables $T_Y = Q(M, x) \cap D$ and $T_N = Q(M, x) \cap \bar{D}$ upon input x and then simulating M. Notice that in this case $Q(M, x)$ is constructed from M and x and then the oracle for D is queried. On the other hand, if M witnesses $L \in$ NP.ALL.DEP (D) for some set D, then T_Y and T_N must be constructed by using the oracle for D throughout; in the proof of Theorem 4.5B, T_Y and T_N are constructed iteratively.

Proof of A. If for every set D, NP.ALL (D) = co-NP.ALL (D), then NP.ALL (ϕ) = co-NP.ALL (ϕ) so that NP = co-NP. Conversely, suppose that NP = co-NP. It suffices to show that for every D and every $L \in$ NP.ALL (D), the complement $\bar{L}$ of L is in NP.ALL (D).

Let M witness $L \in$ NP.ALL (D) and let q be a polynomial such that for all x, $\|Q(M, x)\| \leqq q(|x|)$. Let f be the multivalued function such that for all x, set-$f(x) = Q(M, x)$. Since for all x, $\|Q(M, x)\| \leqq q(|x|)$, f is in NPMVPB. By Proposition 3.2, the function c(set-f) is in NPSV.

Recall from § 2 the set $K = \{\langle M(i), x, c(T_Y), c(T_N), 0^k \rangle |$ some computation of $M(i)$ on input x accepts x in at most k steps, and if y is a string that is queried in this computation, then $y \in T_Y \cup T_N$ and the answer to the query is "yes" if and only if $y \in T_Y\}$. Recall that K is in NP, so that under the hypothesis NP = cp-NP, $\bar{K}$ is in NP. Let p be a polynomial bounding M's running time, and let us consider the following:

ORACLE PROCEDURE 4.1.
begin
$T_Y := T_N := \phi$;
(1) **for** each y in set-$f(x)$ **do**
  if $y \in$ oracle set
    then $T_Y := T_Y \cup [y]$
    else $T_N := T_N \cup [y]$;
(2) **if** $\langle M, x, c(T_Y), c(T_N), 0^{p(|x|)} \rangle \cap \in \bar{K}$
  then accept x
end.

We have already seen that c(set-f) is in NPSV; thus, control of the for-loop at line 1 takes nondeterministic polynomial time. Since c(set-f) $= Q(M, x)$ and $\|Q(M, x)\| \leqq q(|x|)$, the body of the for-loop executes at most a polynomial number of times. When D is used as oracle set, $T_Y = Q(M, x) \cap D$ and $T_N = Q(M, x) \cap \bar{D}$ upon execution of line 2. Thus, this procedure witnesses $\bar{L} \in$ NP.ALL (D). □

Proof of B. As in the proof of part A, it suffices to show that for every set D and every $L \in$ NP.ALL.DEP (D), the complement $\bar{L}$ of L is in NP.ALL.DEP (D). If M_1 witnesses $L \in$ NP.ALL.DEP (D), then there is a polynomial q such that for all x, $\|Q(M_1, D, x)\| \leqq q(|x|)$. Since it is not necessarily the case that $\|Q(M_1, x)\|$ is bounded by a polynomial in $|x|$, the method used to obtain for each x, $T_Y = Q(M_1, D, x) \cap D$ and $T_N = Q(M_1, D, x) - T_Y$ is different from that used in the proof of part A.

Define the function f as follows: For each input string x of M_1, each pair T_Y and T_N of finite sets of strings, and each integer $k > 0$, y is a value of $f(x, c(T_Y), c(T_N), 0^k)$ if and only if there is a computation C of M_1 on x such that

(i) the kth time that C enters the QUERY state, y is the string on the query tape;

and

(ii) if w is any string queried during the first $k-1$ times that C enters the QUERY state, then $w \in T_Y \cup T_N$ and the answer used by C to the query about w is "yes" if and only if $w \in T_Y$.

It is clear that $f \in$ NPMV (but, in general, f is not in NPMVPB). As long as $T_Y \subseteq D$ and $T_N \subseteq \bar{D}$, then $f(x, c(T_Y), c(T_N), 0^k) \subseteq Q(M_1, D, x)$ and so $\|$set-$f(x, c(T_Y), c(T_N), 0^k)\| \leqq q(|x|)$. Therefore, let g be the function in NPSV obtained from f by Proposition 3.4(i). Then, $g(\langle x, c(T_Y), c(T_N), 0^k \rangle, 0^{q(|x|)}) \subseteq Q(M_1, D, x)$ when $T_Y \subseteq D$ and $T_N \subseteq \bar{D}$.

In the following oracle procedure S is a program variable (of type string, but used to encode a finite set):

ORACLE PROCEDURE 4.2.
begin
input x;
$T_Y := T_N := \phi$;

(1) **for** $k := 1$ **to** $p(|x|)$ **do**
(2) **if** $g(\langle x, c(T_Y), c(T_N), 0^k\rangle, 0^{q(|x|)})$ is defined
then begin
(3) $S := g(\langle x, c(T_Y), c(T_N), 0^k\rangle, 0^{q(|x|)})$;
(4) **for** each y in the set encoded by S **do**
if $y \in$ oracle set
then $T_Y := T_Y \cup [y]$
else $T_N := T_N \cup [y]$
end;
(5) **if** $\langle M, x, c(T_Y), c(T_N), 0^{p(|x|)}\rangle \in \bar{K}$
then accept x
end.

Consider the implementation and running time of this procedure. Under the hypothesis NP = co-NP, $g \in$ NPSV, domain $(g) \in$ NP, and $\bar{K} \in$ NP. Thus, the tests at lines 2 and 5 and the function evaluation at line 3 can be carried out nondeterministically in polynomial time. For each execution of the outer for-loop at line 1, the size of the set computed at line 3 is at most $q(|x|)$. Thus, each execution of the inner for-loop takes at most polynomial time (relative to the oracle set). Since the outer for-loop iterates $p(|x|)$ times, the entire procedure can be implemented to run nondeterministically in polynomial time relative to the oracle set.

Now consider the correctness of this procedure for $\bar{L}$ when using oracle set D. Since $\|Q(M_1, D, x)\| \leqq q(|x|)$ and T_Y and T_N are both initialized to ϕ, it is clear that at the end of k iterations of the for-loop beginning at line (1), $T_Y = \bigcup_{j \leqq k} Q(M_1, D, x, j) \cap D$ and $T_N = \bigcup_{j \leqq k} Q(M_1, D, x, j) \cap \bar{D}$. Since the running time of M_1 is bounded by p, $Q(M_1, D, x) \cap D = \bigcup_{j \leqq p(|x|)} (Q(M_1, D, x, j) \cap D)$ and $Q(M_1, D, x) \cap \bar{D} = \bigcup_{j \leqq p(|x|)} (Q(M_1, D, c, j)) \cap D)$. Thus, when execution reaches line (5), $T_Y = Q(M_1, D, x) \cap D$ and $T_N = Q(M_1, D, x) \cap \bar{D}$. This means that the text at line (5) correctly determines membership in $\bar{L}$. Thus, there is a nondeterministic oracle machine M_2 that implements this procedure in polynomial time, and $Q(M_2, D, x) = Q(M_1, D, x)$ so that $\|Q(M_2, D, x)\| \leqq q(|x|)$. Hence, M_2 witnesses $\bar{L} \in$ NP.ALL.DEP (D). □

We do not know whether one can obtain a positive relativization of "NP = ? co-NP" in terms of the classes NP.ACC.DEP (?). Baker, Gill and Solovay [3] described a set E such that NP$(E) \neq$ co-NP (E), and their proof shows that NP.ACC.DEP $(E) \neq$ co-NP.ACC.DEP (E). Thus, of the relativizations considered here, NP.ALL.DEP (?) is the most general positive relativization of NP relative to the "NP = ? co-NP" problem.

For what sets D is it the case that NP.ALL (D) = co-NP.ALL (D)? To study the question, consider the notion of being "hard."

DEFINITION 4.6. If $R(\cdot)$ is a relativization, then write $A \leqq^R B$ whenever $A \in R(B)$. Set D is *$\leqq^R$-hard for class $\mathscr{C}$* if for every $C \in \mathscr{C}$, $C \leqq^R D$.

Simply, D is $\leqq^R$-hard for $\mathscr{C}$ if and only if $\mathscr{C} \subseteq R(D)$.

COROLLARY 4.7. For every set D:

A. NP.ALL (D) = co-NP.ALL (D) *if and only if D is $\leqq^{\text{NP.ALL}}$-hard for* co-NP;

B. NP.ALL.DEP (D) = co-NP.ALL.DEP (D) *if and only if D is $\leqq^{\text{NP.ALL.DEP}}$-hard for* co-NP.

Proof. We consider statement B only, the proof for statement A being similar. First, assume that NP.ALL.DEP (D) = co-NP.ALL.DEP (D) and let us show that co-NP $\subseteq$ NP.ALL.DEP (D). Let $L \in$ co-NP. Then, $\bar{L} \in$ NP $\subseteq$ NP.ALL.DEP (D), so $L \in$ co-NP.ALL.DEP (D), which by assumption yields $L \in$ NP.ALL.DEP (D). Thus, co-NP $\subseteq$ NP.ALL.DEP (D).

Now assume that co-NP $\subseteq$ NP.ALL.DEP (D). Let $L \in$ NP.ALL.DEP (D) be witnessed by oracle machine M. We will see that $\bar{L} \in$ NP.ALL.DEP (D). First let us note that because co-NP $\subseteq$ NP.ALL.DEP (D), given any $f \in$ NPMV, Proposition 3.4(ii) applies, so that $g \in$ NPSV (D) and domain $(g) \in$ NP (D). Moreover, the proof of Proposition 3.4 yields more information than just stated. Namely, the set OKCON $\in$ NP and $\neg$OKCON $\in$ co-NP $\subseteq$ NP.ALL.DEP (D) yields $g \in$ NPSV (D) witnessed by an oracle transducer, call it T, such that $Q(T, D, x)$ is bounded by a fixed polynomial for all x. As a consequence, domain $(g) \in$ NP.ALL.DEP (D) holds also.

Now consider Oracle procedure 4.1. By the comments of the previous paragraph, lines 2 and 3 can be implemented by procedures that are in NP.ALL.DEP (D). Also, $\bar{K} \in$ NP.ALL.DEP (D), since $\bar{K} \in$ co-NP, so line 5 is in NP.ALL.DEP (D) as well. The entire procedure, therefore, witnesses the fact that $\bar{L} \in$ NP.ALL.DEP (D). $\square$

It is also interesting to consider the results of this section from a somewhat different point of view. It is known that $\leqq_T^{\mathrm{NP}}$ is not closed under complementation [3]; that is, here exist A and B such that $A \leqq_T^{\mathrm{NP}} B$ and $\bar{A} \not\leqq_T^{\mathrm{NP}} B$. We have introduced three restrictions of $\leqq_T^{\mathrm{NP}}$: namely, $\leqq^{\mathrm{NP.ALL}}$, $\leqq^{\mathrm{NP.ALL.DEP}}$, and $\leqq^{\mathrm{NP.ACC.DEP}}$. It is natural to ask if these restrictions are closed under complementation. As noted above, $\leqq^{\mathrm{NP.ACC.DEP}}$ is not. For $\leqq^{\mathrm{NP.ALL}}$ and $\leqq^{\mathrm{NP.ALL.DEP}}$, we have the surprising result that they are closed under complementation if and only if NP = co-NP.

As previously stated, the controlled relativizations of NP considered in this section are quantitative in the sense that the "space of queries" is bounded in size by a polynomial. This is a weak constraint on oracle machines since the computation relative to oracle set D of a machine M on input x may still be exponential in size. In § 5 we introduce the deterministic versions of these reducibilities. We will find that the question of whether the deterministic reducibility and the corresponding nondeterministic reducibility differ is, in some instances, equivalent to the "P = ?NP" question. This results in positive relativizations of the "P = ?NP" question.

5. P = ?NP. Now we introduce controlled relativizations of P corresponding to the controlled relativizations of NP defined in § 4. We use these notions to study the "P = ?NP" question. We begin with the analogue of Definition 4.1.

DEFINITION 5.1. Let D be a set. Define the following classes:

(a) P.ALL (D) is the class of languages L such that $L \in P(D)$ is witnessed by a machine M such that, for some polynomial q and all x, $\|Q(M, D, x)\| \leqq q(|x|)$;

(b) P.ACC (D) is the class of languages L such that $L \in P(D)$ is witnessed by a machine M such that, for some polynomial q and all x, $\|QA(M, x)\| \leqq q(|x|)$;

(c) P.ALL.DEP (D) is the class of languages L such that $L \in P(D)$ is witnessed by a machine M such that, for some polynomial q and all x, $\|Q(M, D, x)\| \leqq q(|x|)$;

(d) P.ACC.DEP (D) is the class of languages L such that $L \in P(D)$ is witnessed by a machine M such that, for some polynomial q and all x, $\|QA(M, D, x)\| \leqq q(|x|)$.

For any oracle set four classes are defined in Definition 5.1. Clearly, for every set D, P.ALL $(D) \subseteq$ P.ACC (D) and P.ALL $(D) \subseteq$ P.ALL.DEP $(D) \subseteq$ P.ACC.DEP (D). If M witnesses $L \in$ P (D), then there is a polynomial q such that M runs in time $q(n)$. Since M is deterministic, this means that for all x, $\|Q(M, D, x)\| \leqq q(|x|)$ and $\|QA(M, D, x)\| \leqq q(|x|)$, so that M witnesses $L \in$ P.ALL.DEP (D) and $L \in$ P.ACC.DEP (D). This is stated formally as follows:

LEMMA 5.2. *For every set* D, P.ACC.DEP (D) = P.ALL.DEP (D) = P (D).

Thus, for each set D there are at most three different classes. It will be shown in Corollary 6.3 that there exist sets E and L such that $L \in$ P (E) − NP.ALL (E). For every set D, P.ALL $(D) \subseteq$ P.ACC $(D) \subseteq$ NP.ACC (E) = NP.ALL (D), so that $L \in$

P (E) − P.ACC (E), thus witnessing P.ACC $(E) \neq$ P (E). We do not know whether there is a set F such that P.ALL $(F) \neq$ P.ACC (F). We will see in Corollary 5.5 below that if P.ALL $(D) \neq$ P.ACC (D) for some D, then P ≠ NP.

Now consider positive relativizations of the "P = ? NP" problem.

THEOREM 5.3. A. P = NP *if and only if for every set D*, P.ALL (D) = NP.ALL (D).

B. P = NP *if and only if for every set D*, P.ACC (D) = NP.ALL (D).

C. P = NP *if and only if for every set D*, P (D) = NP.ALL.DEP (D).

Proof of A. This proof is very similar to that of part A of Theorem 4.5. Under the hypothesis P = NP, the function f is such that c(set-f) is in PF by Proposition 3.3. Thus the tables T_Y and T_N can be constructed deterministically in polynomial time using an oracle for D, and then M can be simulated deterministically in polynomial time under the hypothesis P = NP. The details are left to the reader. □

Proof of B. It is clear that for every set D, P.ACC $(D) \subseteq$ NP.ACC (D). But by Lemma 4.4, for every set D, NP.ACC (D) = NP.ALL (D), and so P.ACC $(D) \subseteq$ NP.ALL (D) If for every set D, P.ACC (D) = NP.ALL (D), then P = P.ACC (ϕ) = NP.ALL (ϕ) = NP. If P = NP, then by part A for every set D, P.ALL (D) = NP.ALL (D), and since P.ALL $(D) \subseteq$ P.ACC $(D) \subseteq$ NP.ALL (D), this means that P.ACC (D) = NP.ALL (D). □

Proof of C. This is very similar to the proof of part B of Theorem 4.5. Under the hypothesis P = NP, the functions f and g are such that c(*set*-g) is in PF by Proposition 3.4(iii). Thus, the tables T_Y and T_N can be constructed deterministically in polynomial time using an oracle for D, and M can be simulated deterministically in polynomial time under the hypothesis P = NP. The details are left to the reader. □

We will see in Corollary 6.5 that there is a set F such that P $(F) \neq$ NP.ALL.ACC (F). Thus, of the relativizations considered here, NP.ALL.DEP ( ) is the most general positive relativization of NP relative to the P = ? NP problem.

For what sets D is it the case that P.ALL (D) = NP.ALL (D) or P (D) = NP.ALL.DEP (D)? From Theorem 5.3 and Proposition 3.4 we have the following fact:

COROLLARY 5.4. *For every set D*,

A. P.ALL (D) = NP.ALL (D) *if and only if D is* $\leqq^{\text{P.ALL}}$*-hard for* NP;

B. P (D) = NP.ALL.DEP (D) *if and only if D is* $\leqq^{\text{P}}$*-hard for* NP.

Thus, the questions "do the reducibilities $\leqq^{\text{NP.ALL}}$ and $\leqq^{\text{P.ALL}}$ coincide?" and "do the reducibilities $\leqq^{\text{NP.ALL}}$ and $\leqq^{\text{P.ACC}}$ coincide?" and "do the reducibilities $\leqq^{\text{NP.ALL.DEP}}$ and $\leqq^{\text{P}}_T$ coincide?" are each equivalent to the question "P = ? NP."

From Theorem 5.3 we see that if P = NP, then for every set D, P.ALL (D) = NP.ALL (D) = P.ACC (D). We rephrase this as a corollary which shows that distinguishing between certain deterministic reducibilities would imply that P ≠ NP:

COROLLARY 5.5. *If there exists a set D such that* P.ALL $(D) \neq$ P.ACC (D), *then* P ≠ NP.

We do not know whether the converse of Corollary 5.5 is true.

Let us consider Theorem 5.3 in a different light. For every set D, P.ALL (D) is a restriction of P (D) and NP.ALL (D) is a restriction of NP (D); for some choices of D, these are proper restrictions. To prove that P ≠ NP, one wishes to find a set E such that P.ALL (E) is as small as possible and NP.ALL (E) is as large as possible so that some argument, possibly a diagonalization, can be used to show that P.ALL $(E) \neq$ NP.ALL (E).

Recall that there is a set E such that P $(E) \neq$ NP (E). The proof of this fact in [3] does not speak to the difference between deterministic and nondeterministic computation in general, but rather illustrates the power of nondeterminism in steps that write on the query tape, allowing $Q(M, E, x)$ to be exponential in size; in the

deterministic case, the size of $Q(M, E, x)$ is always bounded by a polynomial (the running time). On the other hand, by forcing the size of $Q(M, D, x)$ and $Q(M, x)$ to be bounded in the same way in both the deterministic and the nondeterministic cases, Theorem 5.3 allows us to face the issue of the potential difference between deterministic and nondeterministic computation within the framework of relativized computation.

As noted at the end of § 3, nondeterminism arises in the procedure used in the proof of Proposition 3.4 only in the tests that direct the search. This is also true in the proof of Theorems 4.5 and 5.3. If one attempts to prove Theorem 5.3B directly from Proposition 3.4, then one can show the following fact:

COROLLARY 5.6. *For every set D*, NP.ALL.DEP $(D) \subseteq$ P $(K \oplus D)$.

Then one obtains Theorem 5.3B by noting that P = NP implies $K \in \mathrm{P} \subseteq P(D)$ so that P $(K \oplus D)$ = P (D) for all D. Thus, one obtains P (D) = NP.ALL.DEP (D) from Corollary 5.6. Similarly, Corollary 5.4 follows from this observation.

Figure 1 shows the relationships between the classes studied in this section.

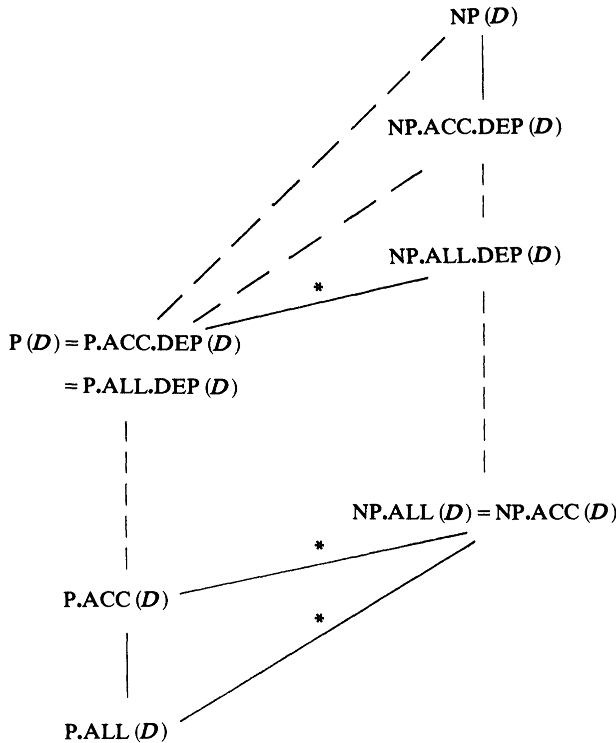


FIG. 1. $\mathscr{L}_1(D)/\mathscr{L}_2(D)$ *indicates that for all D*, $\mathscr{L}_1(D) \subseteq \mathscr{L}_2(D)$. $\mathscr{L}_1(D)$.·´$\mathscr{L}_2(D)$ *indicates that for all D*, $\mathscr{L}_1(D) \subseteq \mathscr{L}_2(D)$ *and for some E*, $\mathscr{L}_1(E) \neq \mathscr{L}_2(E)$. $\mathscr{L}_1(D)$ */$\mathscr{L}_2(D)$ *indicates that* P = NP *if and only if for all D*, $\mathscr{L}_1(D) = \mathscr{L}_2(D)$.

6. Some separation theorems. It is natural to ask if $\leqq_T^{\mathrm{NP}}$, $\leqq^{\mathrm{NP.ACC.DEP}}$, $\leqq^{\mathrm{NP.ALL.DEP}}$ and $\leqq^{\mathrm{NP.ALL}}$ differ. Here we show that both $\leqq^{\mathrm{NP.ALL}}$ and also $\leqq^{\mathrm{NP.ALL.DEP}}$ differ from all of the others. We have not shown that $\leqq^{\mathrm{NP.ACC.DEP}}$ and $\leqq_T^{\mathrm{NP}}$ differ, but we conjecture that this is true.

THEOREM 6.1. *There exists a recursive set E such that* NP.ALL $(E) \neq$ NP.ALL.DEP (E).

Proof. The proof is based on a construction used by Ladner, Lynch and Selman [15]. Assume that the alphabet Σ contains 0, 1 and %. (If $\|\Sigma\| = 2$, we assume a simple coding of 0, 1 and % over Σ.) For each $x \in \Sigma^*$, let $S(x) = \{x\%y \mid |y| \leqq |x| \text{ and } y \in \{0, 1\}^*\}$.

Associated with the set E (which is described below) will be a language L that will be shown to be in NP.ALL.DEP (E) − NP.ALL (E). Membership of x in L will depend on a single string $x\%y \in S(x)$ with $|x|=|y|$; more precisely, x will be in L if and only if $x\%y$ is in E. The set E is constructed so that the prefixes $x\%$, $x\%a_1$, $x\%a_1a_2, \cdots, x\%a_1a_2\cdots a_{|x|-1}$ of $x\%y$ can be used to determine $x\%y = x\%a_1\cdots a_{|x|}$ where, for $i=1,\cdots,|x|$, a_i is in $\{0,1\}$. This notion is implemented by considering any proper prefix $x\%a_1\cdots a_j$ of $x\%y$ for $0 \leqq j < |x|$ and letting the next prefix be $x\%a_1\cdots a_j0$ if $x\%a_1\cdots a_j \notin E$ and $x\%a_1\cdots a_j1$ if $x\%a_1\cdots a_j \in E$. When E is constructed so as to have this property, then the following oracle procedure witnesses $L \in$ NP.ALL.DEP (E).

ORACLE PROCEDURE 6.1.
begin
input x;
$w := x\%$;
for $i := 1$ to $|x|$ **do**
  if $w \in$ oracle set
    then $w := w1$
    else $w := w0$;
if $w \in$ oracle set
  then accept x
  else reject x
end.

Notice that this procedure witnesses $L \in \mathrm{P}(E) \subseteq$ NP.ALL.DEP (E).

The set E will be constructed in stages. For each natural number m, at the end of stage m, $E(m+1)$ and $\bar{E}(m+1)$ will denote the set of strings put into E and $\bar{E}$, respectively, in stages 0 through m. The length of the longest string in $E(m+1) \cup \bar{E}(m+1)$ will be denoted by n_{m+1}. It will be the case that if $|x| \leqq n_{m+1}$, then $x \in E(m+1) \cup \bar{E}(m+1)$, so that $E(m+1)$ and $\bar{E}(m+1)$ completely determine all strings of length at most n_{m+1}. When the construction begins, $E(0)=\{e\}$, $\bar{E}(0)=\phi$ and $n_0=0$.

Recall that $M(0), M(1), \cdots$ is an effective enumeration of the clocked nondeterministic oracle machines that run in polynomial time, and that for every i, machine $M(i)$ runs in time $p_i(n)=n^i+i$. For any oracle set D, we say that "$L \in$ NP.ALL (D) via $M(i)$ and p_j" if $L = L(M(i), D)$ and, for all x, $\|Q(M(i),x)\| \leqq p_j(|x|)$. We consider ordered pairs of natural numbers. At stage $m=\langle i,j\rangle$ the construction guarantees that $L \notin$ NP.ALL (E) via $M(i)$ and p_j. This will yield $L \notin$ NP.ALL (E).

Stage $m=\langle i,j\rangle$. Let n be the least integer such that $n > n_m$ and $p_j(n) < 2^n$. For every string x such that

$$\left\lfloor \frac{n_m}{2} \right\rfloor \leqq |x| \leqq \left\lfloor \frac{p_i(n)-1}{2} \right\rfloor,$$

add each string $x\%1^k$, $0 \leqq k \leqq |x|$, to $E(m)$; add all other strings $y \notin S(0^n)$ such that $n_m < |y| \leqq p_i(n)$ to $\bar{E}(m)$. Notice that the effect of this is to put into L each string x such that

$$\left\lfloor \frac{n_m}{2} \right\rfloor \leqq |x| \leqq \left\lfloor \frac{p_i(n)-1}{2} \right\rfloor$$

except for the string 0^n. Also, all strings y such that $n_m < |y| \leqq p_i(n)$ have been placed into $E(m) \cup \bar{E}(m)$ except for $y \in S(0^n)$. Membership of 0^n in L or in $\bar{L}$ will be determined so that 0^n witnesses $L \notin$ NP.ALL (E) via $M(i)$ and p_j.

Case (1). $\|Q(M(i), 0^n)\| > p_j(n)$. In this case $L \notin$ NP.ALL (E) via $M(i)$ and p_j because $Q(M(i), 0^n)$ is too large. Place each of the strings $0^n\%1^k$, $0 \leqq k \leqq n$, into $E(m)$; place all other strings in $S(0^n)$ into $\bar{E}(m)$. Set $E(m+1)$ to $E(m)$, $\bar{E}(m+1)$ to $\bar{E}(m)$, n_{m+1} to $p_i(n)$, and go to stage $m+1$.

Case (2). $\|Q(M(i), 0^n)\| \leqq p_j(n)$. By choice of n, $p_j(n) < 2^n$ so there is some string $0^n\%y = 0^n\%a_1 \cdots a_n$ in $S(0^n)$ which is not in $Q(M(i), 0^n)$. Place each of the strings $0^n\%a_1 \cdots a_i$, $0 \leqq i \leqq n-1$, into $E(m)$ and place all other strings $w \in S(0^n) - \{0^n\%y\}$ into $\bar{E}(m)$. At this point the only string of length at most $p_i(n)$ which is not yet decided for E or $\bar{E}$ is $0^n\%y$. Now consider computations of $M(i)$ relative to $E(m)$. (Note that all queries can be answered consistently with E and $\bar{E}$ since the running time of $M(i)$ is bounded by p_i and $0^n\%y \notin Q(M(i), 0^n)$.) If $0^n\%y \in L(M(i), E(m))$, then place 0^n % y into $\bar{E}(m)$; otherwise, place 0^n % y into $E(m)$. This yields $L \notin$ NP.ALL (E) via $M(i)$ and p_j because $M(i)$ on input 0^n, with an oracle for E, gives the wrong answer. Set $E(m+1)$ to $E(m)$, $\bar{E}(m+1)$ to $\bar{E}(m)$, n_{m+1} to $p_i(n)$, and go to stage $m+1$. *End of stage m.*

From the construction of $E = \bigcup_m E(m)$ and $\bar{E} = \bigcup_m \bar{E}(m)$, it is clear that $L \notin$ NP.ALL (E). Since each stage is effective, E is a recursive set. □

Recall that for all sets D, NP.ALL $(D) \subseteq$ NP.ALL.DEP $(D) \subseteq$ NP.ACC.DEP $(D) \subseteq$ NP(D). From Theorem 6.1 we have the following fact:

COROLLARY 6.2. *There is a recursive set E such that* NP.ALL $(E) \neq$ NP.ACC.DEP (E) *and* NP.ALL $(E) \neq$ NP (E).

In fact, the proof of Theorem 6.1 yields more fruit:

COROLLARY 6.3. *There is a recursive set E such that* NP.ALL $(E) \neq$ P(E).

Now we show that $\leqq^{\text{NP.ALL.DEP}}$ and $\leqq^{\text{NP.ACC.DEP}}$ differ.

THEOREM 6.4. *There exists a recursive set F such that* NP.ALL.DEP $(F) \neq$ NP.ACC.DEP (F).

Proof. For any set D, let $L(D) = \{x \mid \text{there exists } y \text{ such that } |y| = |x| \text{ and } y \in D\}$. We will construct a set F such that for every n, either F contains no strings of length n or F contains exactly one string of length n; this will show that $L(F) \in$ NP.ACC.DEP (F). The set F will be constructed so that $L(F)$ is not in NP.ALL.DEP (F).

The set F will be constructed in stages. The notation $F(m+1)$, $\bar{F}(m+1)$, and n_{m+1} is defined as in the proof of Theorem 6.1. Initially, $F(0) = \{e\}$, $\bar{F}(0) = \phi$, and $n_0 = 0$. (The construction of $\bar{F}$ in finite stages is not necessary from a technical standpoint, but may be useful for the reader.)

For any set D, we say that "$L \in$ NP.ALL.DEP (D) via $M(i)$ and p_j" if $L = L(M(i), D)$ and $\|Q(M(i), D, x)\| \leqq p_j(|x|)$ for all x, where again we assume an effective enumeration of the clocked nondeterministic oracle machines that run in polynomial time.

Again we consider pairs of natural numbers. At stage $m = \langle i, j\rangle$ the construction guarantees that $L(F) \notin$ NP.ALL.DEP (F) via $M(i)$ and p_j. This will yield $L(F) \notin$ NP.ALL.DEP (F).

Stage $m = \langle i, j\rangle$. Let n be the smallest integer such that $n > n_m$ and $p_j(n) < 2^n$. Add to $\bar{F}(m)$ all strings y such that $n_m < |y| \leqq p(n)$.

Case (1). $\|Q(M(i), F(m), 0^n)\| > p_j(n)$. In this case, $L(F) \notin$ NP.ALL.DEP (F) via $M(i)$ and p_j because $Q(M(i), F(m), 0^n)$ is too big. Set $F(m+1)$ to $F(m)$, $\bar{F}(m+1)$ to $\bar{F}(m)$, n_{m+1} to $p_i(n)$, and go to stage $m+1$.

Case (2). $\|Q(M(i), F(m), 0^n)\| \leqq p_j(n)$. There are two possibilities:

(a) If $M(i)$ accepts 0^n relative to $F(m)$, since $n > n_m$, $0^n \notin L(F)$ so that 0^n witnesses $L(F) \notin$ NP.ALL.DEP (D) via $M(i)$ and p_j. Set $F(m+1)$ to $F(m)$, $\bar{F}(m+1)$ to $\bar{F}(m)$, n_{m+1} to $p_i(n)$, and go to stage $m+1$.

(b) If $M(i)$ does not accept 0^n relative to $F(m)$, then notice that some string of length n, say y, is not in $Q(M(i), F(m), 0^n)$ since $\|Q(M(i), F(m), 0^n)\| \leqq p_j(n) < 2^n$. Set $F(m+1)$ to $F(m) \cup \{y\}$, $\bar{F}(m+1)$ to $\bar{F}(m) - \{y\}$, n_{m+1} to $p_j(n)$, and go to stage $m+1$. Notice that $|0^n| = |y|$ and $y \in F$ so that $0^n \in L(F)$. Thus, 0^n witnesses $L(F) \notin$ NP.ALL.DEP (F) via $M(i)$ and p_j. *End of stage m.*

From the construction of $F = \bigcup_m F(m)$ and $\bar{F} = \bigcup_m \bar{F}(m)$, it is clear that F is a recursive set and $L(F) \notin$ NP.ALL.DEP (F). □

Recall that for every set D, NP.ALL.DEP $(D) \subseteq$ NP.ACC.DEP $(D) \subseteq$ NP (D). Also, notice that for every set D, P $(D) \subseteq$ NP.ALL.DEP (D). Thus, from Theorem 6.4 we have the following fact:

COROLLARY 6.5. *There is a recursive set F such that* NP.ALL.DEP $(F) \neq$ NP (F) *and* P $(F) \neq$ NP.ALL.ACC (F).

7. Uniform relativizations. In §§ 4 and 5 we have considered relativizations where the size of the query tree is polynomially bounded. Ladner, Lynch and Selman [15] considered classes where the entire set of queries could be computed uniformly before any questions were asked of the oracle. In the case of deterministic polynomial time, Ladner, Lynch and Selman showed that this restriction of $\leqq_T^{\mathrm{P}}$ characterizes deterministic polynomial time truth-table reducibility. On the other hand, the nondeterministic polynomial time version of truth-table reducibility developed in [15] is equivalent to $\leqq_T^{\mathrm{NP}}$. In the present study we are interested in the question of whether knowing the existence of a polynomial bound on the set of queries allows one to uniformly compute this set in advance. We find all answers to these questions except those that directly entail solutions of imporant separation problems.

DEFINITION 7.1. Let D be a set.

(a) Let P.UNIF.ALL (D) (P.UNIF.ACC (D)) be the class of languages L such that $L \in$ P (D) is witnessed by a deterministic polynomial time oracle machine M such that the function $f(x) = c(Q(M, x))$ (resp., $f(x) = c(QA(M, x))$) is in PF.

(b) Let NP.UNIF.ALL (D) (NP.UNIF.ACC (D)) be the class of languages L such that $L \in$ NP (D) is witnessed by a nondeterministic polynomial time oracle machine M with the property that the function $f(x) = c(Q(M, x))$ (resp., $f(x) = c(QA(M, x))$) is in NPSV.

It follows from characterizations of polynomial time truth-table reducibility $\leqq_{tt}^{\mathrm{P}}$, given in [15], that $\leqq^{\mathrm{P.UNIF.ALL}} = \leqq_{tt}^{\mathrm{P}}$. This is not so for the corresponding nondeterministic reducibilities. For every set D, NP.UNIF.ALL $(D) \subseteq$ NP.ALL (D) and, by Corollary 6.2, there is a set E such that NP.ALL $(E) \neq$ NP (E). Therefore, recalling once more that $\leqq_T^{\mathrm{NP}} = \leqq_{tt}^{\mathrm{NP}}$, we see that $\leqq^{\mathrm{NP.UNIF.ALL}} \neq \leqq_{tt}^{\mathrm{NP}}$. Similarly, it is easy to see that $\leqq^{\mathrm{NP.UNIF.ALL}} \neq \leqq_{tt}^{\mathrm{NP}}$.

For any set D, if M_1 witnesses $L \in$ P.UNIF.ACC (D), let $f \in$ PF be the function such that or all x, $f(x) = c(QA(M_1, x))$. Let M_2 be a machine that on input x, first computes $f(x)$ and then simulates M_1 on x, accepting relative to D if and only if M_1 accepts x relative to D and rejecting if in the simulated computation M_1 reaches a query configuration where the string on the query tape is not in $QA(M_1, x)$. Thus, $Q(M_2, x) = QA(M_1, x)$ for all x, and so M_2 witnesses $L \in$ P.UNIF.ALL (D). This means that for all sets D, P.UNIF.ACC $(D) \subseteq$ P.UNIF.ALL $(D) \subseteq$ P.ALL $(D) \subseteq$ P.ACC (D) and, similarly, NP.UNIF.ACC $(D) \subseteq$ NP.UNIF.ALL $(D) \subseteq$ NP.ALL (D).

We consider relations between these relativizations under the hypothesis NP = co-NP.

LEMMA 7.2. *If* NP = co-NP, *then for every set D*, NP.UNIF.ACC (D) = NP.UNIF.ALL (D) = NP.ALL (D) = NP.ACC (D) *and each of these classes is closed under complementation.*

Proof. Recall from Lemma 4.4 that NP.ALL (D) = NP.ACC (D). Under the hypothesis NP = co-NP, Proposition 3.2 yields NPMVPB $\subseteq_c$ NPSV. If M witnesses $L \in$ NP.ALL (D) for some set D, then the multivalued function f defined by $y \in \text{set-}f(x)$ if and only if $y \in Q(M, x)$ is in NPMVPB, so the function $c(Q(M, x))$ is in NPSV. Thus, a machine that on input x first computes $c(Q(M, x))$ and then simulates M on x will witness $L \in$ NP.UNIF.ALL (D). Thus, NP.UNIF.ALL (D) = NP.ALL (D).

Let M witness $L \in$ NP.UNIF.ALL (D). As just stated, the function $f(x) = c(Q(M, x))$ is in NPSV when NP = co-NP. For each $y \in Q(M, x)$, determining whether $y \in QA(M, x)$ can be done nondeterministically in polynomial time. Assuming NP = co-NP, it follows that the function $g(x) = c(QA(M, x)) \in$ NPSV and that $L \in$ NP.UNIF.ACC (D). Thus, NP.UNIF.ACC (D) = NP.UNIF.ALL (D). Finally, closure of these classes under complementation follows easily from the assumption NP = co-NP. □

Construction of a set which separates any of the classes related in Lemma 7.2 would yield a proof that NP ≠ co-NP. On the other hand, the hypothesis NP = co-NP allows us to conclude that if we know the existence of a polynomial bound on the set of queries, then we can compute this set in advance. Further separation of either the deterministic or nondeterministic relativizations will be hard to prove because of the corollary to the next result.

LEMMA 7.3. *If* P = NP, *then*

A. *for every set* D, P.UNIF.ACC (D) = P.UNIF.ALL (D) = P.ALL (D) = P.ACC (D); *and*

B. *for every set* D, NP.UNIF.ACC (D) = NP.UNIF.ALL (D) = NP.ALL (D) = NP.ACC (D).

Proof. Part B is an immediate consequence of Lemma 7.2. Now consider Part A. By Corollary 5.5, P = NP implies that P.ALL (D) = P.ACC (D) for all sets D.

The proofs that P.UNIF.ALL (D) = P.ALL (D) and that P.UNIF.ACC (D) = P.UNIF.ALL (D) are almost identical to the proofs used in Lemma 7.2 to show that NP.UNIF.ALL (D) = NP.ALL (D) and that NP.UNIF.ACC (D) = NP.UNIF.ALL (D), respectively. □

THEOREM 7.4. A. P = NP *if and only if for every set* D, P.UNIF.ACC (D) = NP.UNIF.ACC (D).

B. P = NP *if and only if for every set* D, P.UNIF.ALL (D) = NP.UNIF.ALL (D).

Proof. To prove both A and B from right to left, let $D = \phi$. The proof of A (resp., B) from left to right follows immediately from Lemma 7.3 and Theorem 5.3B (resp., Theorem 5.3A). □

COROLLARY 7.5. *If* P = NP, *then for every set* D, P.UNIF.ACC (D) = P.UNIF.ALL (D) = P.ALL (D) = P.ACC (D) = NP.ACC (D) = NP.ALL (D) = NP.UNIF.ALL (D) = NP.UNIF.ACC (D).

Thus, if P = NP, then all of the reducibilities studied here that involve bounding the number of query strings are simply equivalent formulations of deterministic polynomial time truth-table reducibility. This means that if one can separate any two of the restrictions of P $(\cdot)$ or NP $(\cdot)$ obtained by uniformly bounding the set of queries, then P ≠ NP. Notice that nondeterministic polynomial time truth-table reducibility is equal to nondeterministic polynomial time Turing reducibility [15], and P $(\cdot)$ is not equal to NP $(\cdot)$ whether or not P equals NP.

Figure 2 shows the relationships between the classes studied in this section.

8. P = ? PSPACE and NP = ? PSPACE. Now we turn to the "P = ? PSPACE" and "NP = ? PSPACE" questions. Baker, Gill and Solovay [3], Baker and Selman [4] and

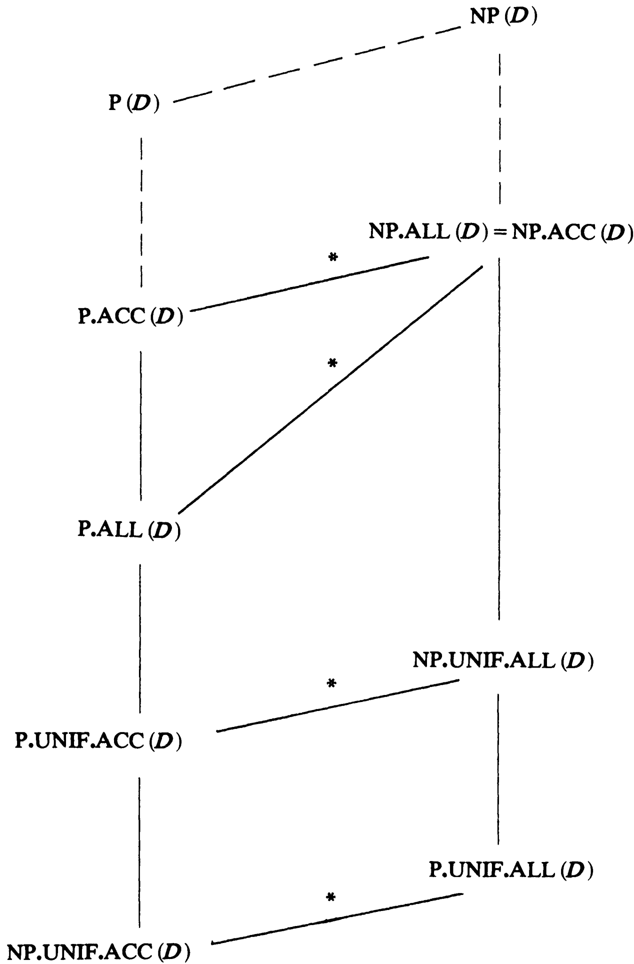


FIG 2. *$\mathscr{L}_1(D)/\mathscr{L}_2(D)$ indicates that for all D, $\mathscr{L}_1(D) \subseteq \mathscr{L}_2(D)$. $\mathscr{L}_1(D)$ ⋰ $\mathscr{L}_2(D)$ indicates that for all D, $\mathscr{L}_1(D) \subseteq \mathscr{L}_2(D)$ and for some E, $\mathscr{L}_1(D) \neq \mathscr{L}_2(E)$. $\mathscr{L}_1(D) \overset{*}{/} \mathscr{L}_2(D)$ indicates that* P = NP *if and only if for all D, $\mathscr{L}_1(\mathrm{D}) = \mathscr{L}_2(D)$.*

Simon and Gill [25] have shown the existence of various sets D such that NP $(D) \neq$ PSPACE (D). However, if E is $\leq_T^{\mathrm{P}}$-complete for PSPACE, then P(E) = NP (E) = PSPACE (E) = PSPACE. Book and Wrathall [5], [7] developed a positive relativization of the "NP = ? PSPACE" problem by introducing a new relativization of PSPACE ( ), called NPQUERY ( ). The corresponding positive relativization of the "P = ? PSPACE" problem was conjectured in [5] and subsequently affirmed in [23]. Here we consider analogues of the quantitative relativizations of P and NP and obtain further positive relativizations of the "P = ? PSPACE" and "NP = ? PSPACE" problems.

DEFINITION 8.1. Let D be a set. Let NPQUERY $(D) = \{L(M, D) | M$ is a nondeterministic oracle machine that uses at most polynomial work space and that is restricted so that in every accepting computation only a polynomial number of oracle calls are made$\}$. Let PQUERY $(D) = \{L(M,D) | M$ is a deterministic oracle machine that uses polynomial work space and that is restricted so that in every accepting computation only a polynomial number of oracle calls are made$\}$.

The classes NPQUERY (D) and PQUERY (D) have a certain "robustness" since they are invariant under the following changes of definition:

(i) "in every accepting computation" is replaced by "there exists an accepting computation";

(ii) "in every accepting computation" is replaced by "in every computation."

Recall that PSPACE is closed under complementation. However, there exist sets D and E such that NPQUERY $(D) \neq$ co-NPQUERY (D) and PQUERY $(E) \neq$ NPQUERY (E). The interest in these relativizations in the present study stems from the following facts:

PROPOSITION 8.2 (a) [5]. NP = PSPACE *if and only if for every set D*, NP $(D) =$ NPQUERY (D).

(b) [23]. P = PSPACE *if and only if for every set D*, P $(D) =$ PQUERY (D).

Let M be an oracle machine. If M is time-bounded or if M is space-bounded and the number of oracle queries allowed is bounded by the space bound, then M can be simulated by a machine of the same type that queries the oracle about any given string at most once. Thus, if M is a polynomial space-bounded oracle machine with the property that for every set D there is a polynomial q such that, for all x, $\|Q(M, D, x)\| \leqq q(|x|)$ or $\|QA(M, D, x)\| \leqq q(|x|)$, then M witnesses $L(M, D) \in$ NPQUERY (D) if M is nondeterministic and M witnesses $L(M, D) \in$ PQUERY (D) if M is deterministic. Thus, quantitative relativizations of PSPACE similar in form to those of P and NP considered in §§ 4–7 are, in fact, modifications of PQUERY ( ) and NPQUERY ( ).

Consider again multivalued deterministic or nondeterministic transducers with accepting states. A string y is an output of a transducer T on input x if some computation of T on input x halts in an accepting state with y on its distinguished output tape. We consider only those transducers T which operate in polynomial space and for some polynomial p, if y is an output of T on input x then $|y| \leqq p(|x|)$.

DEFINITION 8.3. (a) NPSPACEMV is the set of all partial, multivalued functions computed by nondeterministic polynomial space-bounded transducers.

(b) NPSPACESV is the set of all $f \in$ NPSPACEMV that are single-valued.

(c) NPSACEMVPB is the set of all $f \in$ NPSPACEMV such that for some polynomial q and all x, $\|\text{set-}f(x)\| \leqq q(|x|)$.

(d) PSPACEF is the set of all partial single-valued functions computed by deterministic polynomial space-bounded transducers.

Recall that PSPACE $= \bigcup_{k \geqq 1}$ DSPACE $(n^k) = \bigcup_{k \geqq 1}$ NSPACE (n^k) and PSPACE = co-PSPACE. Also, analogous to what was done in § 2, let us introduce the conventions "NPSPACEMVPB $\subseteq_c$ NPSPACESV" and "NPSPACEMVPB $\subseteq_c$ PSPACEF" to mean that "for every $f \in$ NPSPACEMVPB, $c(\text{set-}f) \in$ NPSPACESV" or "$c(\text{set-}f) \in$ PSPACEF," respectively. The arguments used to prove Propositions 3.2 and 3.3 can be extended to show the following fact:

LEMMA 8.4. (a) NPSPACESV = PSPACEF, NPSPACEMVPB $\subseteq_c$ NPSPACESV, *and* NPSPACEMVPB $\subseteq_c$ PSPACEF.

(b) *If* P = PSPACE, *then* PF = PSPACEF.

(c) *If* NP = PSPACE, *then* NPSV = PSPACEF.

Let D be a set. Let $L \in$ NPQUERY (D) be witnessed by a machine M_1 such that for some polynomial q and all x, $\|Q(M_1, x)\| \leqq q(|x|)$. Let f be the function given by $y \in \text{set-}f(x)$ if and only if $y \in Q(M_1, x)$, so that $f \in$ NPSPACEMVPB. Thus, the function $g(x) = c(Q(M_1, x))$ is in NPSPACESV = PSPACEF. Hence, there is a deterministic polynomial space-bounded oracle machine M_2 that recognizes L relative to D by first computing $g(x)$ on input x, then querying the oracle for D about each string in

$Q(M_1, x)$, and finally deterministically simulating M_1's computations on x relative to D.

Suppose that one defines the analogues of the classes NP.ALL (D), P.ALL (D), etc., specified by polynomial space-bounded oracle machines, where in the uniform cases the function $g(x) = c(Q(M, x))$ is computed by a polynomial space-bounded transducer. Consider only those classes where the bound on the set of queries does not depend on the oracle set. The argument given in the last paragraph applies to all of these classes, so that Lemma 8.4 yields the fact that for any set D, there is exactly one class resulting from a polynomial bound on the size of $Q(M, x)$. Thus, the fact that PSPACE $= \bigcup_{k \geqq 1}$ DSPACE $(n^k) = \bigcup_{k \geqq 1}$ NSPACE (n^k) allows one to conclude that, instead of eight potentially different classes for an oracle set D, there is just one; this is the analogue of Corollary 7.4. We will refer to this class as PQUERY.ALL (D). Details are left to the reader.

Now consider the situation where the size of $Q(M, D, x)$ depends on D as well as M.

DEFINITION 8.5. Let D be a set.

(a) Let NPQUERY.ALL.DEP (D) (NPQUERY.ACC.DEP (D)) be the class of languages L such that $L \in$ NPQUERY (D) is witnessed by a nondeterministic polynomial space-bounded oracle machine M such that, for some polynomial q and all x, $\|Q(M, D, x)\| \leqq q(|x|)$ ($\|QA(M, D, x)\| \leqq q(|x|)$).

(b) Let PQUERY.ALL.DEP (D) (PQUERY.ACC.DEP (D)) be the deterministic counterpart of the class defined in (a).

The proof of Lemma 5.2 can be extended to yield the following fact:

LEMMA 8.6. *For every set D,* PQUERY.ALL.DEP (D) = PQUERY.ACC.DEP (D) = PQUERY (D).

The fact that $\bigcup_k$ DSPACE $(n^k) = \bigcup_k$ NSPACE (n^k) = PSPACE = co-PSPACE allows one to use the argument in the proof of Theorem 5.3, part C, to obtain the following fact:

LEMMA 8.7. *For every set D,* PQUERY (D) = NPQUERY.ALL.DEP (D).

It is a straightforward exercise to modify the proof of Theorem 6.1 to show the existence of a set E such that PQUERY.ALL (E) $\neq$ NPQUERY.ALL.DEP (E) so that PQUERY.ALL (E) $\neq$ PQUERY (E). Also, one can modify the proof of Theorem 6.4 to show the existence of a set F such that NPQUERY.ALL.DEP (F) $\neq$ NPQUERY.ACC.DEP (F).

Thus, for every set D, there are at most four distinct classes obtained by starting with NPQUERY (D) and making restrictions on the access to sets of oracle queries.

Now we have positive relativizations of the "P =? PSPACE" and "NP =? PSPACE" problems.

THEOREM 8.8. A. P = PSPACE *if and only if for every set D,* P.UNIF.ALL (D) = PQUERY.ALL (D).

B. NP = PSPACE *if and only if for every set D,* NP.UNIF.ALL (D) = PQUERY.ALL (D).

C. NP = PSPACE *if and only if for every set D,* NP.ALL.DEP (D) = PQUERY (D).

D. NP = PSPACE *if and only if for every set D,* NP.ACC.DEP (D) = NPQUERY.ACC.DEP (D).

Proof of A. If for every set D, P.UNIF.ALL (D) = PQUERY.ALL (D), then P = P.UNIF.ALL (ϕ) = PQUERY.ALL (ϕ) = PSPACE. Since for every set D, P.UNIF.ALL (D) $\subseteq$ PQUERY.ALL (D), it suffices to show that P = PSPACE implies that for every set D, PQUERY.ALL (D) $\subseteq$ P.UNIF.ALL (D).

Let $L \in$ PQUERY.ALL (D) be witnessed by M so that the function $c(Q(M, x))$ is in PSPACEF. Under the hypothesis P = PSPACE, PF = NPSV = PSPACEF by Lemma 8.4 so that $c(Q(M, x)) \in$ PF. Now for each x, $T_Y = Q(M, x) \cap D$ and $T_N = Q(M, x) \cap \bar{D}$ can be computed deterministically in polynomial time using an oracle for D. Under the hypothesis P = PSPACE, there is a deterministic polynomial time procedure that on input $(x, c(T_Y), c(T_N))$ will determine whether x is accepted by M relative to D. Thus, L is in P.UNIF.ALL (D).

Proof. of B, C, *and* D. The proofs of parts B, C and D are essentially the same so that only that of part D will be given. As in the proof of part A, it is sufficient to show that for every set D, NPQUERY.ACC.DEP (D) $\subseteq$ NP.ACC.DEP (D).

Let M witness $L \in$ NPQUERY.ACC.DEP (D) for some D. Consider the following predicates:

(i) EVAC is true for a potential configuration J of M if and only if there is an accepting computation of M that begins with J and does not pass through a query configuration;

(ii) NEXT is true for a pair (I, J) of potential configurations of M if and only if I is not a query configuration, J is a query configuration, and there is a computation of M that begins with I and reaches J without passing through a query configuration.

Since M is a nondeterministic polynomial space-bounded oracle machine, EVAC and NEXT are predicates in PSPACE. Under the hypothesis NP = PSPACE, both EVAC and NEXT are total predicates in NP.

Since M witnesses $L \in$ NPQUERY.ACC.DEP (D), there is a polynomial q such that for every input x to M, every computation of M on x makes at most $q(|x|)$ oracle queries.

Consider the following:

ORACLE PROCEDURE 8.1.
begin
input x;
$I :=$ initial configuration of M on x;
$k := 0$;
while $k < q(|x|)$ and $\neg$EVAC(I) **do**
  begin
  nondeterministically guess a query configuration J;
  if NEXT(I, J)
    then begin
      query the oracle "$y \in$ oracle set?" for the string y
        on the query tape in configuration J;
      $I :=$ successor of J;
      $k := k+1$
      end
    else halt in a nonaccepting state
  end;
if EVAC(I)
  then accept
  else halt in a nonaccepting state
end.

Recall that EVAC and NEXT are total predicates in NP. The loop is executed at most $q(|x|)$ times. Thus the procedure is nondeterministic and runs in polynomial time and clearly accepts x relative to D if and only if $x \in L$. It is clear that the oracle

is queried regarding $y \in D$ in an accepting computation if and only if y is in QA (M, D, x). hence, this procedure witnesses $L \in$ NP.ACC.DEP (D). □

From Theorem 8.8 the following fact is immediate:

COROLLARY 8.9. A. *If* P = PSPACE, *then for every set D*, P.UNIF.ACC (D) = P.UNIF.ALL (D) = P.ALL (D) = P.ACC (D) = PQUERY.ALL (D).

B. *If* NP = PSPACE, *then for every set D*, NP.UNIF.ACC (D) = NP.UNIF.ALL (D) = NP.ALL (D) = PQUERY.ALL (D).

The proofs of the positive relativizations of the problems considered in §§ 4–7 are more complex than the proof of Theorem 8.8. The positive relativization of the "P = ? PSPACE" problem compares two classes specified by deterministic machines; thus, it is sufficient to guarantee that the sets of queries be generated in the same way. The same thinking applies to the positive relativizations of the "NP = ? PSPACE" problem, but is more striking since Theorem 8.8.D is the only result here that compares "ACC.DEP ( )" classes.

It is clear that for every set D, NP.ALL.DEP $(D) \subseteq$ PQUERY (D) (since by Lemma 8.7, PQUERY (D) = NPQUERY.ALL.DEP (D)). Thus, Theorem 8.8 shows that if NP = PSPACE, then for every set D, NP.ALL.DEP (D) = PQUERY (D). The results of [5] show that limiting the number of oracle calls does not restrict the power of nondeterminism in computations, since for every set D, NPQUERY (D) is closed under nonerasing homomorphism. On the other hand, there exists a set E such that PQUERY $(E) \neq$ NPQUERY (E), PQUERY (E) = NPQUERY.ALL.DEP (E), and PQUERY (E) is not closed under polynomial-erasing (even -nonerasing) homomorphism. Thus, the relativization NPQUERY.ALL.DEP $(\cdot)$ does not have the full power of nondeterminism. Also, if NP = PSPACE, then NP.ALL.DEP (E) = PQUERY (E), so that the relativization NP.ALL.DEP $(\cdot)$ does not have the full power of nondeterminism. We state this formally in the following way:

COROLLARY 8.10. *If for every set D*, NP.ALL.DEP (D) *is closed under polynomial-erasing (-nonerasing) homomorphism, then* NP $\neq$ PSPACE.

9. The polynomial-time hierarchy. Now we turn to the polynomial-time hierarchy.

Let D be a set. Define $\Sigma_1^P(D) = \mathrm{NP}(D)$, $\Pi_1^P(D) = \text{co-}\Sigma_1^P(D)$, and $\Delta_1^P(D) = \mathrm{P}(D)$. For each $i > 0$, define $\Sigma_{i+1}^P(D) = \bigcup\{\mathrm{NP}(E) \mid E \in \Sigma_i^P(D)\}$, $\Pi_{i+1}^P(D) = \text{co-}\Sigma_{i+1}^P(D)$, and $\Delta_{i+1}^P(D) = \bigcup\{\mathrm{P}(E) \mid E \in \Sigma_i^P(D)\}$. Define $\mathrm{PH}(D) = \bigcup_{i>0} \Sigma_i^P(D)$ and $\mathrm{PH} = \mathrm{PH}(\phi)$.

The structure $\Sigma_1^P(D), \Sigma_2^P(D), \cdots$ is known as the *polynomial-time hierarchy relative to D*. See [27], [30].

It is known [4] that there exists a set D such that $\Sigma_1^P(D) \subsetneq \Sigma_2^P(D) \subsetneq \Sigma_3^P(D)$, but it is not known if there is a set E such that $\Sigma_i^P(E) \neq \Sigma_{i+1}^P(E)$ for all i; that is, whether there exists a properly infinite polynomial-time hierarchy. Of course, PH = NP if and only if NP = co-NP.

It is the case that for every set D, PH $(D) \subseteq$ PSPACE (D), and if PH (D) = PSPACE (D), then the polynomial-time hierarchy relative to D is finite.

A positive relativization of the "PH = ? PSPACE" problem is known [7]. We review this result here.

Let D be a set. Define $\Sigma_1^{PQ}(D) = \mathrm{NPQUERY}(D)$ and for each $i > 0$, $\Sigma_{i+1}^{PQ}(D) = \bigcup\{\mathrm{NPQUERY}(E) \mid E \in \Sigma_i^{PQ}(D)\}$. Define $\mathrm{PQH}(D) = \bigcup_{i>0}\Sigma_i^{PQ}(D)$.

PROPOSITION 9.1. [7]. PH = PSPACE *if and only if for every set D*, PH (D) = PQH (D).

Now we consider connections between the polynomial-time hierarchy and the operator NP.ALL.DEP ( ) and show that the hierarchy obtained by application of

NP.ALL.DEP ( ) determines exactly the class Δ_2^P of the polynomial-time hierarchy. In contrast, it is not known how many proper levels exist in the polynomial-time hierarchy.

THEOREM 9.2. *For each* $k \geqq 2$, NP.ALL.DEP $(\Delta_k^P) = \Delta_k^P$.

Proof. Clearly, $\Delta_k^P \subseteq$ NP.ALL.DEP (Δ_k^P) for all k. For some $k \geqq 2$ and some $A \in \Delta_k^P$, let L be in NP.ALL.DEP (A). Corollary 5.6 shows that NP.ALL.DEP $(A) \subseteq$ P $(A \oplus K)$ so that $L \in$ P $(A \oplus K)$. But for $k \geqq 2$, $K \in \Delta_k^P$ so that $A \oplus K \in \Delta_k^P$ when $A \in \Delta_k^P$. This implies that $L \in \Delta_k^P$ since P $(\Delta_k^P) = \Delta_k^P$. Since $k \geqq 2$ was chosen arbitrarily, we have NP.ALL.DEP $(\Delta_k^P) \subseteq \Delta_k^P$ for all $k \geqq 2$. □

Theorem 9.2 provides the basis for a characterization of the classes Δ_k^P, $k \geqq 2$, in the polynomial-time hierarchy in terms of the operator NP.ALL.DEP ( ) as opposed to the operator P ( ).

COROLLARY 9.3. *For each* $k \geqq 1$, NP.ALL.DEP $(\Sigma_k^P) = \Delta_{k+1}^P$.

Proof. It is clear that for all k, $\Delta_{k+1}^P \subseteq$ NP.ALL.DEP. (Σ_k^P). For each k, $\Sigma_k^P \subseteq \Delta_{k+1}^P$ so that NP.ALL.DEP $(\Sigma_k^P) \subseteq$ NP.ALL.DEP (Δ_{k+1}^P). But for $k \geqq 1$, NP.ALL.DEP $(\Delta_{k+1}^P) = \Delta_{k+1}^P$ by Theorem 9.2 so that NP.ALL.DEP $(\Sigma_k^P) \subseteq \Delta_{k+1}^P$. □

For each set D, let NP.ALL.DEP^1 (D) = NP.ALL.DEP (D) and for $k > 1$, let NP.ALL.DEP^{k+1} (D) = NP.ALL.DEP (NP.ALL.DEP^k (D)).

COROLLARY 9.4. *For each* $k \geqq 2$, NP.ALL.DEP^k $(P) = \Delta_2^P$.

Proof. Clearly, NP.ALL.DEP^1 (P) = NP = Σ_1^P. Thus, NP.ALL.DEP^2 (P) = NP.ALL.DEP (Σ_1^P). By Corollary 9.3, NP.ALL.DEP $(\Sigma_1^P) = \Delta_2^P$ so that NP.ALL.DEP^2 $(P) = \Delta_2^P$. For each $k > 2$, applying Theorem 9.2 yields NP.ALL.DEP^k (P) = NP.ALL.DEP^2 $(\Delta_2^P) = \Delta_2^P$ by induction on k. □

Let us consider Corollary 9.4 from a different viewpoint. Consider the smallest class $\mathscr{L}$ of languages such that $\mathscr{L}$ contains the empty set and $\mathscr{L}$ = NP.ALL.DEP $(\mathscr{L})$. The proof of Corollary 9.4 shows that $\mathscr{L}$ is exactly Δ_2^P. In contrast, the least fixed point for the operator NP ( ) is PH, and the least fixed point for the operator P ( ) is P.

Theorem 9.2 also yields interesting information about why nondeterminism is necessary for recognition in polynomial time of sets in the Σ_k^P levels of the polynomial-time hierarchy for $k \geqq 2$, assuming of course that the hierarchy is proper through level k.

COROLLARY 9.5. *For each* $k \geqq 2$, *if* $B \in \Sigma_k^P - \Delta_k^P$, *then for every nondeterministic polynomial time-bounded oracle machine* M *and every set* A *in* Σ_{k-1}^P *such that* $L(M, A) = B$, *it is the case that for every polynomial* p, $\|Q(M, A, x)\| > p(|x|)$ *for infinitely many* x.

Proof. The statement is that if $B \in \Sigma_k^P - \Delta_k^P$ *for some* $k \geqq 2$, then $B \notin$ NP.ALL.DEP (Σ_{k-1}^P). But this follows immediately from Corollary 9.3 since NP.ALL.DEP $(\Sigma_{k-1}^P) = \Delta_k^P$ when $k \geqq 2$. □

Corollary 9.5 may be interpreted as follows: If $B \in \Sigma_k^P - \Delta_k^P$ for some $k \geqq 2$, then all nondeterministic polynomial time-bounded oracle machines which accept B relative to some set in Σ_{k-1}^P must search through nonpolynomial-size portions of that oracle set on infinitely many of the accept inputs. Thus, if the polynomial time hierarchy extends to some level $k \geqq 2$ with $\Delta_k^P \subsetneq \Sigma_k^P$, then the hierarchy stands to this level because the power of nondeterminism is necessary for searching through oracle sets when accepting sets in $\Sigma_i^P - \Delta_i^P$ for $2 \leqq i \leqq k$.

10. Possible extensions. Here we consider some possible topics for further study.

10.1. Other resource bounds. The techniques developed in §§ 4–9 apply equally well to classes specified by resource bounds other than just the set of polynomials and to classes specified by simultaneous bounds on time and space. Positive relativizations

of some of the questions comparing classes specified by time and space have been developed previously [6], [23]. Here let us consider other resource bounds.

For every set D, let DEXT (D) (NEXT (D)) be the class of languages accepted relative to D by deterministic (resp., nondeterministic) oracle machines that operate in time 2^{cn} for some $c>0$.

It is not known whether DEXT = NEXT or whether NEXT = co-NEXT. There exist sets A and B such that DEXT (A) = NEXT (A) and DEXT (B) $\neq$ NEXT (B). Jones and Selman [10] have shown that a language L is in NEXT if and only if L is the binary encoding of a spectrum, i.e., is the set of cardinalities of the finite models of a first-order formula (see also [8]). Thus, the question of whether the complement of each spectrum is itself a spectrum is equivalent to the question of whether NEXT is closed under complementation. The technique used to study "NP = ? co-NP" in § 4 is applicable here.

For every set D, let NL (D) be the class of languages accepted relative to D by nondeterministic oracle machines that operate in linear time. Just as the polynomial-time hierarchy is defined from NP ( ) in § 9, one can define the linear-time hierarchy from NL ( ). The class RUD = $\bigcup_{i>0} \mathrm{NL}^i(\phi)$ has been shown by Wrathall [29], [31] to be an encoding of the rudimentary predicates. The question of whether RUD is equal to DSPACE (lin), i.e., the class of languages accepted by deterministic machines that use linear space, is open. For every set D, RUD (D) $\subseteq$ DSPACE (lin, D). Using techniques similar to those in §§ 8 and 9 and in [7], one can show that RUD = DSPACE (lin) if and only if for every set D, $\bigcup_{i>0} \mathrm{NL}^i(D) = \bigcup_{i>0} \{\mathrm{NL}^i(D \oplus A) | A \in \mathrm{DSPACE(lin)}\}$. Again, this question is of interest both because of the foundational role of the rudimentary predicates and also because DSPACE (lin) is the class of sets with characteristic functions in the Grzegorczyk class $\mathscr{E}_2$.

10.2. Sparse sets. A set D is *sparse* if there is a polynomial p such that for all $n>0$, $\|\{x \in D \mid |x| \leqq b\}\| \leqq p(n)$. Mahaney [20] has shown that P = NP if and only if there is a sparse set that is NP-complete. Further, Long and Selman [18] have shown that for every $k \geqq 2$ $\Sigma_k^{\mathrm{P}} = \Pi_k^{\mathrm{P}}$ if and only if for every sparse D, $\Sigma_k^{\mathrm{P}}(D) = \Pi_k^{\mathrm{P}}(D)$.

Consider the result of Mahaney mentioned above along with the result of Theorem 5.3C: P = NP if and only if there is a sparse set that is NP-complete if and only if for every set D, P (D) = NP.ALL.DEP (D). On the one hand, every sparse set is "small," and on the other hand, if M witnesses $L \in$ NP.ALL.DEP (D), then for all x, $Q(M, D, x)$ is "small." We suggest that there may be formal connections between the restricted relativizations considered here and arbitrary relativizations relative to sparse oracles.

10.3. Structural properties of complexity classes. In addition to inclusion relationships between complexity classes, structural properties of complexity classes have recently been studied via relativizations. Sipser [26] has constructed an oracle A for which NP (A) $\cap$ co-NP (A) has no complete set. There are several constructions [9], [22] of oracles B such that NP (B) has P (B)-immune sets, i.e., infinite sets in NP (B) which have no infinite subsets in P (B). Of course, for any oracle C such that P (C) = NP (C), NP (C) contains no P (C)-immune sets. The original construction of Baker, Gill and Solovay [3] described a set D such that NP (D) $-$ P (D) contains a sparse set. On the other hand, Kurtz [13] has shown the existence of an oracle E such that NP (E) $-$ P (E) contains no sparse sets.

Such results naturally suggest the development of positive relativizations of structural properties of complexity classes. We conjecture that NP contains P-immune sets if and only if for every set A, NP.ALL.DEP (A) contains P (A)-immune sets. Perhaps it is the case that NP $-$ P contains a sparse set if and only if NP.ALL.DEP (A) $-$ P (A) contains a sparse set for some set A.

10.4. Effectively enumerable reducibilities. There seems to be yet another difference between the standard polynomial time-bounded oracle machine model and the relativizations studied here that is worth drawing out. As we have noted earlier, every relativization $R(\ )$ determines a reducibility $\leqq^R$, and, conversely, every reducibility $\leqq^R$ determines the relativization given by forming the $\leqq^R$-reduction classes, e.g., $R(D)=\{A|A\leqq^R D\}$. If $\leqq^R$ is a deterministic polynomial time-bounded reducibility and $\mathcal{M}$ is a class of deterministic oracle machines, we will say that *$\mathcal{M}$ defines* $\leqq^R$ if $A\leqq^R D$ if and only if $A\leqq^R D$ by means of a machine in $\mathcal{M}$. Similarly, if $\leqq^R$ is a nondeterministic polynomial time-bounded reducibility and $\mathcal{M}$ is a class of nondeterministic oracle machines, *$\mathcal{M}$ defines* $\leqq^R$ if $A\leqq^R D$ if and only if $A\leqq_T^{\mathrm{NP}} D$ by means of a machine in $\mathcal{M}$. A set $\mathcal{M}$ of oracle machines is *effectively enumerable* if $\mathcal{M}$ is the range of some total recursive function. A reducibility $\leqq^R$ is an *effectively enumerable reducibility* if there is an effectively enumerable set $\mathcal{M}$ of machines that defines $\leqq^R$.

To illustrate, the set $\mathcal{T}$ of all deterministic oracle Turing machines that operate in polynomial time defined $\leqq_T^{\mathrm{P}}$, but $\mathcal{T}$ is not effectively enumerable. Nevertheless, $\leqq_T^{\mathrm{P}}$ is an effectively enumerable reducibility because the class of all deterministic oracle machines that have polynomial time clocks is effectively enumerable and defines $\leqq_T^{\mathrm{P}}$.

In this paper we have defined a number of relativizations by placing semantic conditions on the computation trees of their defining machines. Although they are open questions, we suspect that

$$\leqq^{\mathrm{NP.ALL}},\quad \leqq^{\mathrm{NP.ACC}},\quad \leqq^{\mathrm{NP.ALL.DEP}},\quad \leqq^{\mathrm{NP.ACC.DEP}},\quad \leqq^{\mathrm{P.ALL}},\quad \leqq^{\mathrm{P.ACC}},$$
$$\leqq^{\mathrm{NP.UNIF.ALL}} \text{ and } \leqq^{\mathrm{NP.UNIF.ACC}}$$

are not effectively enumerable. It is easy to see that

$$\leqq^{\mathrm{P.UNIF.ALL}} \text{ and } \leqq^{\mathrm{P.UNIF.ACC}}$$

are effectively enumerable.

Acknowledgments. It is a pleasure to thank two anonymous referees for their extremely constructive criticism of the first version of this paper. In addition, the first author wishes to thank José Balcázar and Uwe Schöning for some very stimulating conversations regarding this work.

REFERENCES

[1] D. ANGLUIN, *On counting problems and the polynomial-time hierarchy*, Theoret. Comput. Sci., 12 (1980), pp. 161–173.

[2] ———, *On relativizing auxiliary pushdown machines*, Math. Systems Theory, 13 (1980), pp. 283–299.

[3] T. BAKER, J. GILL AND R. SOLOVAY, *Relativizations of the* P=?NP *question*, this Journal, 4 (1975), pp. 431–442.

[4] T. BAKER AND A. SELMAN, *A second step towards the polynomial hierarchy*, Theoret. Comput. Sci., 8 (1979), pp. 177–187.

[5] R. BOOK, *Bounded query machines: On* NP *and* PSPACE, Theoret. Comput. Sci., 15 (1981), pp. 27–39.

[6] R. BOOK, C WILSON AND XU MEI-RUI, *Relativizing time, space, and time-space*, this Journal, 11 (1982), pp. 571–581.

[7] R. BOOK AND C. WRATHALL, *Bounded query machines: On* NP ( ) *and* NPQUERY ( ), Theoret. Comput. Sci., 15 (1981), pp. 41–50.

[8] R. FAGIN, *Generalized first-order spectra and polynomial-time recognizable sets*, in Complexity of Computation, R. Karp, ed., SIAM–AMS Proceedings, American Mathematical Society, Providence, RI, 1973, pp. 43–73.

[9] S. HOMER AND W. MAASS, *Oracle dependent properties of the lattice of* NP *sets*, Theoret. Comput. Sci., 24 (1983) pp. 279–289.

[10] N. JONES AND A. SELMAN, *Turing machines and the spectra of first-order formulas*, J. Symbolic Logic, 39 (1974), pp. 139–150.

[11] D. KOZEN, *Indexings of subrecursive classes*, Theoret. Comput. Sci., 11 (1980), pp. 277–301.

[12] D. KOZEN AND M. MACHTEY, *On relaive diagonals*, unpublished manuscript, 1981.

[13] S. KURTZ, *On sparse sets in* NP−P: *Relativizations*, manuscript, 1983.

[14] R. LADNER AND N. LYNCH, *Relativizations of quesions about log space computability*, Math. Systems Theory, 10 (1976), pp. 19–32.

[15] R. LADNER, N. LYNCH AND A. SELMAN, *A comparison of polynomial-time reducibilities*, Theoret. Comput. Sci., 1 (1975), pp. 103–123.

[16] T. LONG, *Strong nondeterministic polynomial-time reducibilities*, Theoret. Comput. Sci., 21 (1982), pp. 1–25.

[17] T. LONG AND A. SELMAN, *Relativizing complexity classes with sparse oracles*, abstract 1983.

[18] ———, *Some remarks on sparse sets, tally sets, and the polynomial-time hierarchy*, in preparation.

[19] N. LYNCH, *Log space machines with multiple oracle tapes*, Theoret. Comput. Sci., 6 (1978), pp. 25–39.

[20] S. MAHANEY, *Sparse complete sets for* NP: *Solution to a conjecture of Berman and Hartmanis*, J. Comput. Systems Sci., 25 (1982), pp. 130–143.

[21] W. SAVITCH, *Relationships between nondeterministic and deterministic space complexities*, J. Comput. Systems Sci., 4 (1970), pp. 177–192.

[22] U. SCHÖNING AND R. BOOK, *Immunity*, in Automata, Languages, and Programming, Lecture Notes in Computer Science, 154, Springer-Verlag, New York, 1983, pp. 653–661

[23] A. SELMAN, XU MEI-RUI AND R. BOOK, *Positive relativizations of complexity classes*, this Journal, 12 (1983), pp 565–579.

[24] I. SIMON, *On some subrecursive reducibilities*, Ph.D. dissertation, Stanford Univ., Stanford, CA, 1977.

[25] I. SIMON AND J. GILL, *Polynomial reducibilities and upward diagonalizations*, in Proc. 9th ACM Symp. Theory of Computing, Association for Computing Machinery, New York, 1977, pp. 186–194.

[26] M. SIPSER, *On relativization and the existence of complete sets*, in Automata, Languages, and Programming, Lecture Notes in Computer Science, 140, Springer-Verlag, New York, 1982, pp. 523–531.

[27] L. STOCKMEYER, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1976), pp. 1–22.

[28] C. WILSON, *Relativization, reducibilities, and the exponentional hierarchy*, M.S. thesis, Univ. of Toronto, Toronto, Ontario, Canada, 1980.

[29] C. WRATHALL, *Subrecursive predicates and automata*, Ph.D. dissertation, Harvard Univ., Cambridge, MA, 1975.

[30] ———, *Complete sets and the polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1976), pp. 23–33.

[31] ———, *Rudimentary predicates and relative computation*, this Journal, 7 (1976), pp. 194–209.

SIAM J. COMPUT.
Vol. 13, No. 3, August 1984
© 1984 Society for Industrial and Applied Mathematics
003

CONVEX PARTITIONS OF POLYHEDRA: A LOWER BOUND AND WORST-CASE OPTIMAL ALGORITHM*

BERNARD CHAZELLE†

Abstract. The problem of partitioning a polyhedron into a minimum number of convex pieces is known to be NP-hard. We establish here a quadratic lower bound on the complexity of this problem, and we describe an algorithm that produces a number of convex parts within a constant factor of optimal in the worst case. The algorithm is linear in the size of the polyhedron and cubic in the number of reflex angles. Since in most applications areas, the former quantity greatly exceeds the latter, the algorithm is viable in practice.

Key words. Computational geometry, convex decompositions, data structures, lower bounds, polyhedra

1. Introduction. The general problem of decomposing complex structures into simpler components has received a great deal of attention recently [1], [4], [5], [8]. The reason for this concern comes partly from the impossibility of applying many of people's favorite geometric algorithms to nonconvex structures. Often, decomposing the structures into convex parts and applying the algorithms to each part is one way to overcome this difficulty. For example, intersection [2] and searching problems [9] can be solved efficiently by means of convex decompositions. One of the forefathers of decomposition algorithms is Garey et al.'s algorithm [4] for partitioning an n-gon into triangles in $O(n \log n)$ time. Minimality considerations were addressed later on in [1], where an $O(n+N^3)$ time algorithm was given for decomposing an n-gon with N reflex angles into a *minimum* number of nonoverlapping convex pieces. Several variants of this problem were shown to be NP-hard [8]; in particular, the generalization of the problem to polygons with *holes* [5]. This result was to be used as a stepping stone to prove that the following problem was NP-hard.

Given a three-dimensional polyhedron P, what is the smallest set of pairwise disjoint convex polyhedra, whose convex union is exactly P?

This paper is devoted to this problem, and is organized along the following lines: in § 2, we present the basic concepts and outline an effective method for decomposing an arbitrary polyhedron into convex pieces. Let n and N designate respectively the size of the input and the number of reflex angles into the polyhedron. We prove that the algorithm never produces more than approximately $N^2/2$ convex pieces. We show in § 3 that this figure is optimal in the worst case up to within a constant factor. To do so, we exhibit a polyhedron P with an arbitrary number of reflex angles N and $n = O(N)$ vertices, and we prove that P necessarily has $\Omega(N^2)$ convex parts. Of course, by a trivial output size argument, this result also establishes a quadratic lower bound on the time complexity of the decomposition problem. Finally in § 4, we give the details of the algorithm outlined at the beginning.

Before proceeding, we shall set our notation. We define a three-dimensional polyhedron as a finite, connected set of simple plane polygons, such that every edge of each polygon belongs to exactly one other polygon. To exclude degenerate cases (e.g., two cubes connected by a single vertex), we also require that the polygons surrounding each vertex form a simple circuit [3, p. 4]. Note that this definition does

* Received by the editors May 17, 1982, and in revised form July 21, 1983. This work was partly supported by a Yale fellowship and by the Defense Advanced Research Projects Agency under contract F33615-78-C-1551.

† Department of Computer Science, Brown University, Providence, Rhode Island 02912.

not prevent faces from having holes (Fig. 1a). A face with k holes is said to be of *genus k*. Similarly, polyhedra may have holes (i.e., *handles*), and we define the genus of a polyhedron as the genus of the surface formed by its boundary [6]. It follows from the definition that a polyhedron may not have interior boundaries.

Let P be a polyhedron with n vertices $v_1, \cdots, v_n$, p edges $e_1, \cdots, e_p$, and q faces $f_1, \cdots, f_q$. A necessary condition for vertices, edges, and faces to be adjacent is to have at least one point in common. For simplicity, however, we will say that a face and an edge or two faces are adjacent if and only if they share an entire line segment. If T and U are two adjacent faces intersecting in a segment L, we define the angle (T, U) as the angle between two segments lying respectively on T and U and perpendicular to L. Recall that there is no natural orientation of angles in Euclidean space. Thus, to avoid ambiguity, the angle (T, U) will always be measured between 0 and 360 degrees with respect to a given side of the pair T, U. Noticing that each face of P has an outer and an inner side, we define a *notch* of P as an edge with its adjacent faces forming a reflex angle (i.e. >180 degrees) with respect to their inner side (Fig. 1b). Let $g_1, \cdots, g_N$ denote the notches of P.

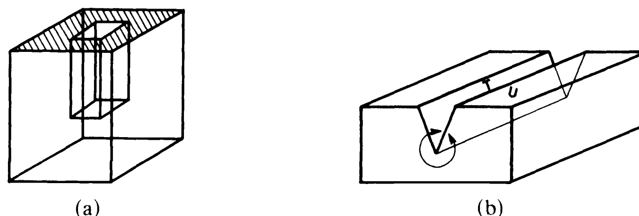


FIG. 1. a) *A face of a polyhedron with a hole in the middle.* b) *A notch of a polyhedron.*

2. The basic method. It is easy to see that the presence of notches in a polyhedron is characteristic of its nonconvexity [3, p. 4]. Thus we can view a convex decomposition of P either as a partition of P into convex polyhedra or as a set of *cuts* performed through P in order to resolve the reflex angles at its notches. This suggests a naive decomposition algorithm, which we proceed to describe next.

2.1. The naive decomposition. Informally, a notch can be removed by cutting along a plane adjacent to it so as to resolve the reflex angle between its adjacent faces. More precisely, let g be a notch of the polyhedron P with f_i and f_j its adjacent faces, and let T be a plane which contains g and resolves its reflex angle, i.e., such that both angles (f_i, T) and (T, f_j), as measured from the inner side of f_i and f_j, are not reflex. The intersection of T and P is in general a set of polygons. These polygons may have holes and the holes may themselves contain other polygons (Fig.11a). Let S be the unique polygon containing g. We call S a *cut* of the naive decomposition. It is clear that cutting along S will remove the notch g. Note that, in general, this operation will break P into two pieces. If P has a nonzero genus, however, removing a notch may simply cut a handle of P and preserve its connectivity. In this case, the polyhedron obtained has two distinct faces with the same geometric location (Fig. 2a). Other intriguing effects may be observed and it is worthwhile to mention some of them.

If the polygon S has holes, removing g may create a handle in either of the two parts produced (Fig. 2b). Therefore the added genus of all the pieces produced thus far will increase by one. We also observe that the operation may produce one piece, while removing a handle and creating another handle (Fig. 2c). We will thus treat the more general case where the polyhedron P may have arbitrary genus, since the naive

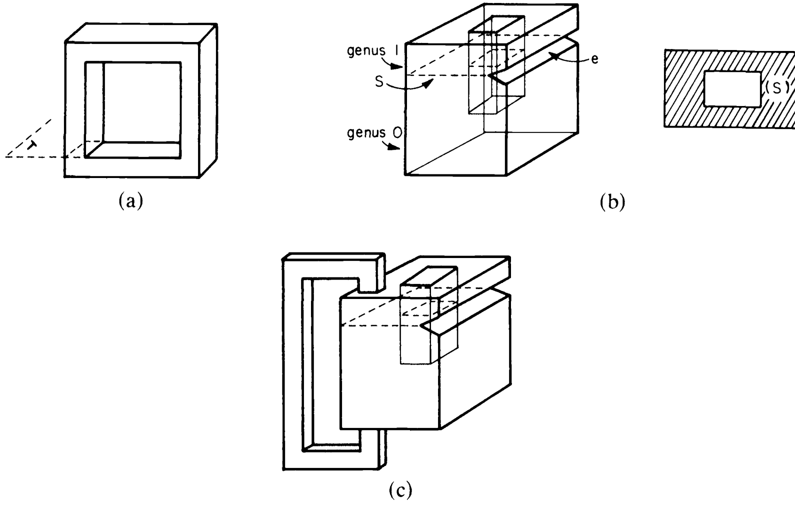


FIG. 2. *Removing a notch.*

decomposition may produce intermediate objects of higher genuses. In spite of these intricacies, we can easily show that repeating the cutting process on each remaining nonconvex part will eventually produce a convex decomposition in a finite number of steps. To find out how many convex parts such a decomposition may generate, we first observe that, at any time, any notch of a part is either a notch of P or the subsegment of a notch of P, called a *subnotch.* This follows from the fact that a cut may intersect other notches, thus *duplicating* them (Fig. 3). Note, however, that no new notch is ever created. At worst, each cut may intersect all of the other notches or subnotches present in the polyhedron considered. If $f(N)$ is the maximum number of cuts which a complete decomposition may necessitate, we have $f(0)=0$, and

$$f(N) \leqq 2f(N-1)+1.$$

Therefore, at most 2^N-1 cuts are needed, which shows that the procedure will always converge and produce at most 2^N convex parts. Unfortunately, as shown in [1], this scheme may indeed produce an exponential number of pieces, so an alternate method is in order.

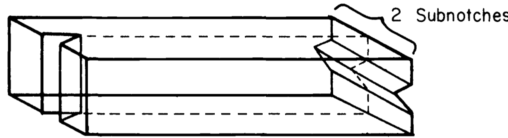


FIG. 3. *The duplication of notches.*

LEMMA 1. *There exist two constants a, b and a class of polyhedra $P(n)$ with $O(n)$ vertices, such that for any $n>a$, the naive decomposition applied to P produces at least 2^{bn} convex parts.*

Proof. See [1]. □

2.2. The naive decomposition revisited. To avoid an exponential blow-up in the number of pieces, we will remove all the subnotches of each notch with coplanar cuts. This will ensure that all the cuts used in the removal of a notch duplicate a total of at most $N-1$ other notches, leading to an $O(N^2)$ upper bound on the number of convex parts. More precisely, let us define for each notch g_i a plane T_i that resolves its reflex

angle. We proceed as before, with the additional requirement that the cuts of each subnotch of g_i should be coplanar with T_i.

THEOREM 2. *The revised naive decomposition algorithm applied to P yields at most* $N^2/2+N/2+1$ *convex parts.*

Proof. We can assume that all the subnotches of a notch are removed consecutively. Since the cuts corresponding to the subnotches of g_i are coplanar, their union intersects every other notch in at most one point. It follows that, at the ith step, each remaining notch will have been broken up into at most $i+1$ subnotches, and step $i+1$ will introduce at most $i+1$ polyhedra into the decomposition. □

In the last section of this paper, we will describe an effective method for carrying out the naive decomposition. But first we will establish a lower bound on the size of any convex decomposition.

3. A quadratic lower bound on the number of convex parts.

3.1. Introduction. The algorithm described above produces $O(N^2)$ convex parts, thus saving us from an exponential blow-up. We may yet wonder whether $O(N)$ parts is not always achievable, as is the case in two dimensions [1]. We next tackle this problem and prove that this $O(N^2)$ upper bound is indeed tight. To achieve our goal, we must exhibit a class of polyhedra which cannot be decomposed into fewer than cN^2 parts. The technique used to derive this lower bound is based on volume considerations. We define a portion Σ of the polyhedron P and, observing that a decomposition of P also realizes a partition of Σ, we study the contribution of each convex part to this partitioning. The crux is to show that a convex part can only have a small piece lying in Σ, and therefore lots of convex parts are needed to fill up Σ. To realize this condition, we must carefully design Σ, giving it a warped shape so that its intersection with any convex object can never occupy too much space. The fact that Σ must be defined by means of straight lines suggests giving it the shape of a hyperbolic paraboloid. Recall that this surface can be generated by two sets of orthogonal lines [11, p. 649].

The main idea can be summarized as follows: Σ has thickness ε so that its volume is approximately εN^2. The warpness of a hyperbolic paraboloid will then ensure that since Σ is bounded by notches, the "chunk" of Σ removed by any convex piece can only be very small, i.e. have volume ε. As a result at least $\Omega(N^2)$ convex parts will be necessary to decompose Σ.

3.2. Description of the polyhedron *P*. P is essentially a rectangular parallelepiped with a series of $N+1$ notches cut through the lower face and $N+1$ similar notches cut through the upper face (Fig. 4a, b). The two faces adjacent to any notch form a very small angle and, for our purposes, can be regarded as a single vertical quadrilateral. Thus, we have $N+1$ such quadrilaterals emanating from the lower face, all of which are vertical, parallel to the plane Oxz, and equidistant. The upper edges of these quadrilaterals are called the *bottom notches* of the polyhedron P, and are designated $BOT0, \cdots, BOTN$ in ascending Y-value. To achieve the desired warping, all the bottom notches lie on the hyperbolic paraboloid $z=xy$. The $N+1$ quadrilaterals emanating from the upper face of P are parallel to the plane Oyz and satisfy the same specifications. Similarly, their lower edges are called the *top notches* of P and are designated $TOP0, \cdots, TOPN$ in increasing X-order. All these notches lie on the hyperbolic paraboloid $z=xy+\varepsilon$. We now give a more precise definition of P by characterizing its significant vertices with the system of axes indicated in Fig. 4b. Note that the origin O is the intersection of $BOT0$ with the vertical plane passing through $TOP0$. The upper face of the parallelepiped lies on the plane $z=2N^2$ and its lower face, on the plane $z=-2N$. This ensures that all bottom and top notches fit strictly

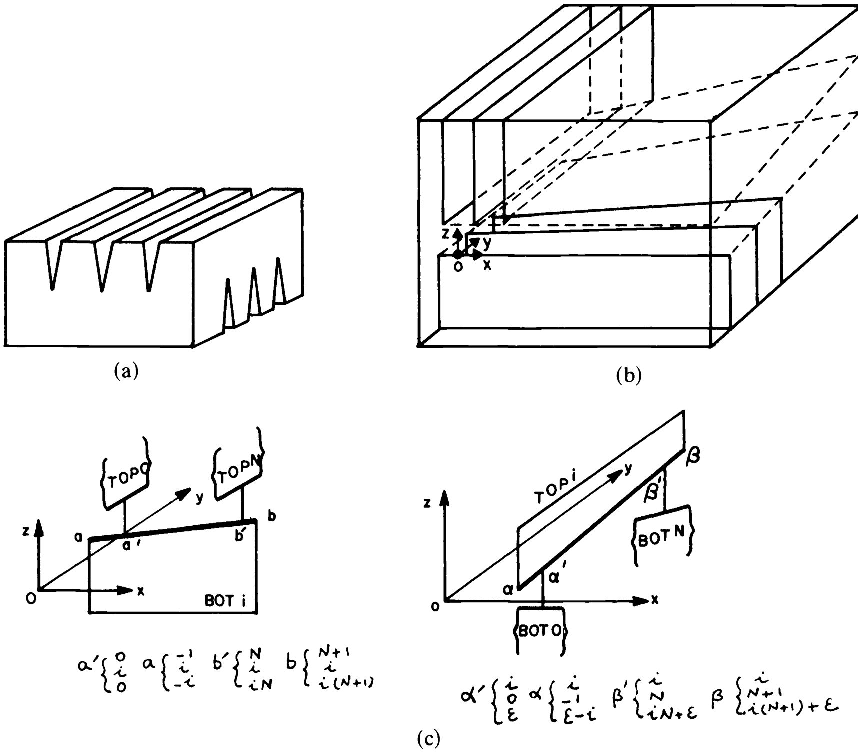


FIG. 4. *The polyhedron P.*

between these two faces. Also the parallelepiped has a depth and width of $N+2$. Fig. 4c gives all the coordinates of the top and bottom notches.

3.3. Decomposing *P* into convex parts. We define Σ as the portion of P comprised between the two hyperbolic paraboloids $z=xy$ and $z=xy+\varepsilon$ and the four planes $x=0$, $x=N$, $y=0$, $y=N$. Σ is a cylinder parallel to the z-axis, of height ε, whose base is the region of the hyperbolic paraboloid $z=xy$ with $0\leqq x, y\leqq N$ (Fig. 5). Let $Q_1, \cdots, Q_m$ be any convex decomposition of P and let Q_i^* denote the intersection of Q_i and Σ. Since Σ lies inside P, the set of Q_i^* forms a partition of Σ. Note that Q_i^* may consist of 0, 1, or several blocks, most of which are likely not to be polyhedra. Our goal is to prove that $m\geqq cN^2$ for some constant c, by showing that the volume of Q_i^* cannot be too large. By volume of Q_i^*, we mean the sum of all the volumes of the blocks composing Q_i^*. We first characterize the shape and the orientation of the large Q_i^*'s, which permits us to derive an upper bound on their maximum volume.

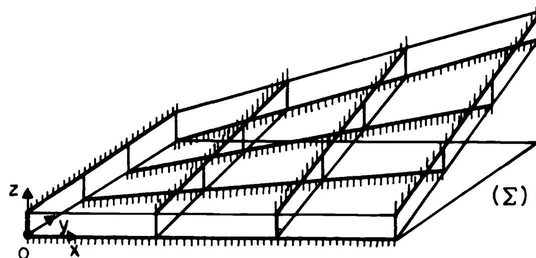


FIG. 5. *The warped region Σ.*

For all i between 0 and N, let $BOTi^*$ (resp. $TOPi^*$) denote the vertical projection of $BOTi$ (resp. $TOPi$) on the plane Oxy. The set of all $BOTi^*$ and $TOPi^*$ forms a regular square grid of N^2 cells, each cell being itself a one-by-one square. Consider the two points A: (x_A, y_A, z_A) and B: (x_B, y_B, z_B) lying in Q_j^*. We will investigate their possible positions when their vertical projections on the grid lie on two parallel lines which are at a distance 2 of each other. Wlog, we will assume that $x_A \leqq x_B$. We have the following result.

LEMMA 3. *Let A and B be two points of Q_j^*.*

1. *If x_A is an integer i with $0 \leqq i \leqq N-2$ and $x_B = x_A + 2$, then $y_B - y_A \leqq 2\varepsilon$.*
2. *If y_A is an integer i with $2 \leqq i \leqq N$ and $y_B = y_A - 2$, then $x_B - x_A \leqq 2\varepsilon$.*

Proof. Recall that the lines supporting $BOTi$ and $TOPi$ are defined respectively by $(y = i, z = ix)$ and $(x = i, z = iy + \varepsilon)$.

1. Let the coordinates of A and B be respectively $(x_A = i, y_A, z_A)$ and $(x_B = i + 2, y_B, z_B)$ with $0 \leqq i \leqq N-2$. Let T be the middle point of the segment AB, $(x_T = i + 1, y_T = (y_A + y_B)/2, z_T = (z_A + z_B)/2$, and consider the point C on $TOPi+1$ with coordinates $(x_C = x_T, y_C = y_T, z_C = x_C y_C + \varepsilon)$. Since Q_i is convex, the whole segment AB lies in Q_i and T lies inside P, therefore $z_T \leqq z_C$. Also, since A and B lie in Σ, $x_A y_A \leqq z_A$ and $x_B y_B \leqq z_B$, therefore $(x_A y_A + x_B y_B)/2 \leqq z_T$. Combining these results yields $(x_A y_A + x_B y_B)/2 \leqq z_C$, therefore

$$iy_A + (i+2) y_B \leqq 2(\varepsilon + (i+1)(y_A + y_B)/2),$$

hence

$$y_B - y_A \leqq 2\varepsilon.$$

2. The proof is very similar. The coordinates of A and B are respectively (x_A, i, z_A) and $(x_B, i-2, z_B)$ with $2 \leqq i \leqq N$. The middle point of AB is now defined by T: $(x_T = (x_A + x_B)/2, y_T = i - 1, z_T = (z_A + z_B)/2)$ and lies right above the point of $BOTi-1$, C: $(x_C = x_T, y_C = y_T, z_C = x_C y_C)$, therefore $z_C \leqq z_T$. Since both A and B belong to Σ, $z_A \leqq x_A y_A + \varepsilon$ and $z_B \leqq x_B y_B + \varepsilon$, therefore

$$2(i-1)(x_A + x_B)/2 \leqq 2\varepsilon + ix_A + (i-2)x_B$$

and

$$x_B - x_A \leqq 2\varepsilon$$

which completes the proof. □

When A is now any point in Σ with $0 \leqq x_A \leqq N-2$ and $2 \leqq y_A \leqq N$, we can still use the previous result to delimit the region where B cannot lie. The shaded area in Fig. 6 represents the forbidden area. Assume that $x_B - \lceil x_A \rceil > 2$ and let A' and B' be the two points on the segment AB with $x_{A'} = \lceil x_A \rceil$ and $x_{B'} = x_{A'} + 2$. Since A$'$ and B' lie in Q_j^*, we can apply the result of Lemma 3 on these two points. It follows that $y_{B'} - y_{A'} \leqq 2\varepsilon$, therefore

$$\frac{y_B - y_A}{x_B - x_A} = \frac{y_{B'} - y_{A'}}{x_{B'} - x_{A'}} \leqq \varepsilon.$$

This shows that B must lie under the line $y = y_A + \varepsilon(x - x_A)$ as indicated in Fig. 6. Similarly, we can show that if $\lfloor y_A \rfloor - y_B > 2$, B must lie on the left-hand side of the line $x = x_A + \varepsilon(y_A - y)$.

We can now attack our main problem, that is, evaluating the maximum volume of Q_j^*. Recall that Q_j^* may be empty or consist of several blocks. Let A be the point of Q_j^* with minimum X-coordinate. We will assume that A does not lie too close to

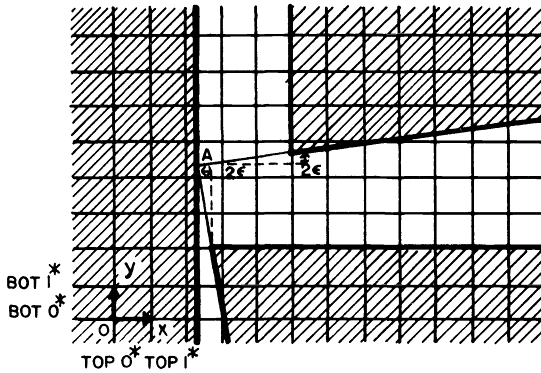


FIG. 6. *The forbidden area.*

*BOT*0 or *TOPN* in order to have the points B and C of Fig. 7 well defined. More precisely, we require that

$$0 < x_A < N-2, \qquad 2 < y_A < N-3\varepsilon.$$

Fig. 7 is only a reproduction of Fig. 6, specifying the regions of interest with respect to A. Note that VA, VB, and VC really denote the intersection of Σ with the vertical cylinders whose bases are represented by the shaded areas in Fig. 7. We know that Q_j^* lies entirely in the union of VA, VB, and VC. So we can partition Q_j^* into 3 parts, $VA1$, $VB1$, and $VC1$, defined respectively as the intersection of Q_j^* with VA, VB, and VC.

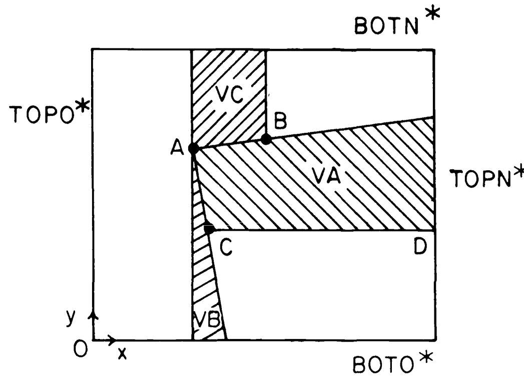


FIG. 7. *Restricting the domain where Q_j^* has to be computed.*

I) *Evaluating the volume of* $VA1$. When there is no ambiguity, we will refer to a three-dimensional object and to its volume by using the same symbol (in this case $VA1$). To derive an upper bound on the volume $VA1$, we integrate a vertical section of $VA1$ along a direction "almost" parallel to Y-axis. This permits us to exploit the warping of Σ in order to bound the area of the section, while having a very short interval of integration. More precisely, let P_w be the vertical plane (P_w: $y = x \tan\theta + w$), and $S(\theta, w)$ the area of the cross section formed by the intersection of P_w and $VA1$. The volume of $VA1$ can be computed by integrating $S(\theta, w)$ along a line normal to the planes P_w.

$$VA1 = \int S(\theta, w) \cos\theta \, dw.$$

If we choose θ larger than (Ox, AB) (Fig. 7), all values of $S(\theta, w)$ will be null outside of A and D, that is, for:

$$w > w_A = y_A - x_A \tan \theta$$

and

$$w < w_D = y_D - x_D \tan \theta.$$

Letting $S(\theta)$ be the maximum value of $S(\theta, w)$ for all w, we have

$$VA1 \leqq (w_A - w_D) S(\theta) \cos \theta$$

and from $y_A - 3 \leqq y_D$ and $x_D = N$, we derive

$$VA1 \leqq (3 + N \tan \theta) S(\theta) \cos \theta. \tag{1}$$

The condition on θ is easily expressed as

$$\varepsilon < \tan \theta. \tag{2}$$

We are now reduced to establishing an upper bound on $S(\theta, w)$. We will find it more convenient to change the system of coordinates so that the point $(0, w, 0)$ becomes the new origin and the line $(z = 0, y = x \tan \theta + w)$ becomes the new X-axis. We express the old coordinates (x, y, z) of any point in terms of the new coordinates (X, Y, Z) as follows:

$$x = X \cos \theta - Y \sin \theta,$$

$$y = w + X \sin \theta + Y \cos \theta,$$

$$z = Z.$$

The hyperbolic paraboloid $z = xy$ is now described by the equation:

$$Z = (X \cos \theta - Y \sin \theta)(w + X \sin \theta + Y \cos \theta)$$

and the intersection of P_w with Σ is a strip in the plane $(Y = 0)$ comprised between the two parabolas:

$$(f): Z = X^2 \sin \theta \cos \theta + Xw \cos \theta,$$

$$(g): Z = X^2 \sin \theta \cos \theta + Xw \cos \theta + \varepsilon.$$

Before proceeding further, we will prove a technical result about areas covered by parabolas. Suppose that we have two parabolas of the previous type, described by $f(x) = ax^2 + bx$ with $a > 0$, and $g(x) = f(x) + \varepsilon$. Let $T(x)$ be the area comprised between the parabola f and the tangent to g at x (Fig. 8). We can show the following

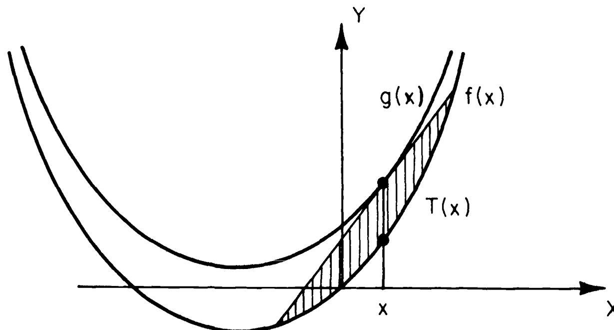


FIG. 8. *The function* $T(x)$.

LEMMA 4. *$T(x)$ is a constant function equal to* $4\varepsilon\sqrt{\varepsilon/a}/3$.

Proof. The tangent to g at x has the equation:

$$Y=(2ax+b)(X-x)+ax^2+bx+\varepsilon$$

and intersects the parabola f at the points with X-coordinates x_1 and x_2, solutions of

$$(2ax+b)(X-x)+ax^2+bx+\varepsilon=aX^2+bX$$

that is,

$$aX^2-2axX+ax^2-\varepsilon=0$$

yielding $x_1=x-\sqrt{\varepsilon/a}$ and $x_2=x+\sqrt{\varepsilon/a}$. It is now straightforward to evaluate $T(x)$.

$$T(x)=\int[(2ax+b)(t-x)+ax^2+bx+\varepsilon-at^2-bt]\,dt$$

that is,

$$T(x)=(x_2-x_1)(\varepsilon-ax^2+ax(x_1+x_2)-a(x_1^2+x_1x_2+x_2^2)/3)$$

therefore

$$T(x)=4\varepsilon\sqrt{\varepsilon/a}/3,$$

which establishes the proof. □

We will now take a closer look at the structure of the parabolic strip formed by the intersection of Σ and P_w which, we know, contains $S(\theta, w)$. Here again, $S(\theta, w)$ designates both the surface and its area. Recall that $S(\theta, w)$ may consist of several disconnected pieces. The intersection of P_w and Σ is a connected strip enclosed between two vertical lines $X=a, X=b$ (the exact values of a and b are irrelevant for our purposes). Also, as illustrated in Fig. 9a, the upper parabola of this strip, g, intersects the top notches, *TOPk*, at regular intervals of length $1/\cos\theta$. Let F denote the convex polygon formed by the intersection of Q_j and P_w. Assuming that F is not empty, we distinguish two cases:

1) No point of F lies above the parabola g (Fig. 9b).

Since F is convex, there exists a line L separating g and F. Since L', the tangent to g parallel to L, also separates g and F, the X-coordinate, u, of the tangent point satisfies $S(\theta)\leqq T(u)$.

2) There exists a point M in F lying above g (Fig. 9c).

Using the notation of Fig. 9c, it is clear that $S(\theta, w)$ lies totally in $L\cup C\cup R$. Since the areas of L and R are dominated by $T(X_k)=T(X_{k+1})$, and the area of C is exactly $\varepsilon/\cos\theta$, we have

$$S(\theta, w)\leqq 2T(X_k)+\varepsilon/\cos\theta.$$

From Lemma 4, it follows that

$$S(\theta, w)\leqq\varepsilon/\cos\theta+\tfrac{8}{3}\varepsilon\sqrt{\varepsilon/\sin\theta\cos\theta}.$$

And from (1), we derive

$$VA1\leqq\varepsilon(3+N\tan\theta)(1+\tfrac{8}{3}\sqrt{\varepsilon/\tan\theta}).$$

II) *Evaluating the volume of* $VC1$. Since the hyperbolic paraboloids are symmetric about x and y, the same computation will give an upper bound on $VC1$. Note that now, no condition like (2) must be set on the angle giving the direction of

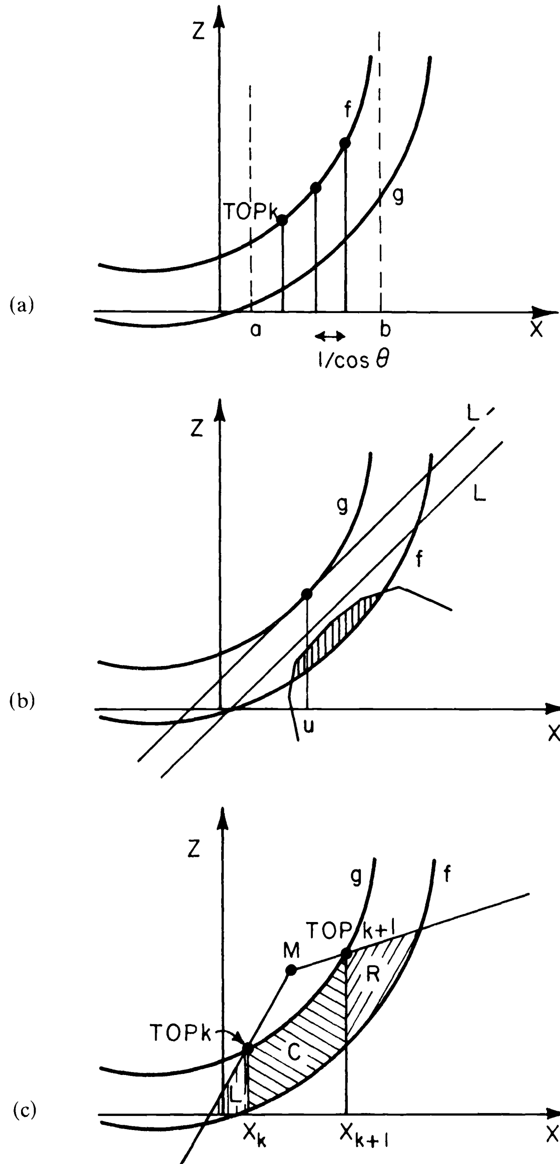


FIG. 9. *Evaluating* $S(\theta, w)$.

integration. For convenience, we will take it equal to θ, however. Thus, we have

$$VC1 \leqq \varepsilon(3+N\tan\theta)(1+\tfrac{8}{3}\sqrt{\varepsilon/\tan\theta}).$$

III) *Evaluating the volume of* $VB1$. The shaded area of Fig. 7 corresponding to VB has a maximum area of $\varepsilon N^2/2$, therefore the volume of VB is dominated by $\varepsilon^2 N^2/2$. This yields an upper bound on $VB1$

$$VB1 \leqq \varepsilon^2 N^2/2.$$

3.4. The lower bound on the number of convex parts. We can now prove our main result.

THEOREM 5. *There exist a constant c and a class of polyhedra involving an arbitrarily large number of vertices such that each polyhedron cannot be decomposed into fewer than cn^2 convex parts, where n is the number of vertices.*

Proof. Recall that the volumes computed in the previous section are only relevant for the points A satisfying

$$0 < z_A < N-2 \quad \text{and} \quad 2 < y_A < N-3\varepsilon.$$

Let V be the corresponding portion of Σ. We have

$$V = (N-2)(N-3\varepsilon-2)\varepsilon.$$

Since no Q_j can contribute more than $VA1 + VB1 + VC1$ to the volume V, we can derive the following lower bound on the number m of convex parts Q_j.

$$m \geqq \frac{V}{VA1 + VB1 + VC1}.$$

Assume that N is large enough and that $\varepsilon < \sin\theta < \tan\theta < 1/N^2$. Relation (2) is then satisfied, and we have

$$VA1,\ VC1 < (1+\tfrac{8}{3})(3+1/N)\varepsilon < 16\varepsilon.$$

Also, since

$$V > \varepsilon N^2/2$$

it follows that

$$m > \varepsilon N^2/2(32\varepsilon + \varepsilon^2 N^2/2),$$

hence

$$m > N^2/66$$

which completes the proof. □

4. The decomposition algorithm. We give a precise description of the decomposition algorithm outlined in § 2. We will show that it is possible to decompose P into $O(N^2)$ pieces in $O(nN^2(N + \log n))$ time, using $O(nN^2)$ storage. We will also indicate that at the price of added complication, we can reduce the running time to $O(nN^3)$.

The first issue to investigate is the mode of representation used for describing a polyhedron. Since many practical problems involve dealing with faces rather than edges or vertices, we may assume that the edges enclosing a given face are readily available. More precisely, we require the data structure chosen to provide three types of lists:

1. *Edge-to-face lists*: contain the names of the two faces adjacent to each edge.
2. *Face-to-edge lists*: give the sequence of edges enclosing each face in clockwise order.
3. *Adjacency lists*: provide a set of the vertices adjacent to each vertex.

Note that the faces of a nonconvex polyhedron may be polygons with holes. In that case, each face-to-edge list should provide clockwise descriptions of the outer as well as of the inner boundaries. We call a *graph representation* of a polyhedron any representation providing the above lists. We may notice that these representations are redundant, but they are chosen to be so for the sake of simplicity. These lists reflect the *size* of the polyhedron accurately, however, since they clearly require $O(p)$ storage. Recall that p is the number of edges in P.

Because decomposing P consists essentially of dividing it up with successive cuts, we first consider the problem of computing graph representations for the two polyhedra P_1 and P_2 into which a cut S breaks up P. For the time being, we will assume P to be of genus 0. In the following, we will successively show I) how to compute the intersection of T and P, II) how to obtain S from it, and III) how to compute the two polyhedra P_1 and P_2. But before proceeding we need to take a closer look at the problem and prove a preliminary result.

Let e be the edge through which the cut is performed. We first compute W, the intersection of P with the plane supporting S. W may consist of a set of polygons with holes, which may themselves contain polygons of the same nature. We identify S as the unique polygon which contains the edge e (Figs. 2, 11). Whereas it is immediate to compute a description of the outer boundary of S, obtaining the inner boundaries (if any) requires more work. Viewing W as a set of nonintersecting boundaries, we first determine all the boundaries in W which lie inside the outer boundary of S, thus forming a set W^*. Next, we keep all the *maxima* of W^*. A boundary is said to be a *maximum* if it is not contained in any other boundary. We can show that the two problems are very closely related, and that an algorithm for solving one can easily be modified to handle the other.

LEMMA 6. *All the maxima of a set W of boundaries can be found in $O(n \log n)$ time, if n is the total number of vertices in W.*

Proof. To begin with, we should note that the nonintersection of the boundaries of W implies that W always has at least one maximum. The method which we will describe is inspired from Shamos and Hoey's algorithm for intersecting pairs of segments [10]. The crucial observation to make is that the intersection of a vertical line L with the maxima of W forms a set (possibly empty) of disjoint segments. The endpoints of each segment lie on some edges of W, and the vertical line L induces a total ordering R on the set E of these edges. E consists exactly of all the edges of maxima which intersect L (Fig. 10a). We say that two edges of E, consecutive with respect to R, are *linked* if the vertical segment joining them lies in a maximum of W. Note that consecutive pairs of edges in R are alternately linked and not linked. For any point v of L, we define $h(v)$ (resp. $l(v)$) as the first edge in E above v (resp. below v). If no such edge exists, $h(v)$ or $l(v)$ is 0 (Fig. 10a). The notion of *above* and *below* is, of course, defined with respect to the vertical line L. Similarly, the order of two edges of W is defined with respect to a common intersecting vertical line. Actually, this order is the same for any vertical line since the edges of W can intersect only at their endpoints. If v is the leftmost vertex of a polygon P of W, P is a maximum if and only if $h(v)$ and $l(v)$ are not linked. This condition is clearly necessary since, if $h(v)$ and $l(v)$ are linked, they belong to the same polygon, which cannot be P since v is its leftmost vertex. To see that it is sufficient, assume that P is not a maximum; then there is a unique maximum Q in W which contains P, and Q must intersect the vertical line passing through v, therefore the intersection is a segment containing v and the pair $h(v)$, $l(v)$ must be linked.

The algorithm proceeds as follows: we sweep a vertical line from left to right, passing through each vertex v in W. The vertices are maintained in sorted order (by X-values) in a set Q. We first check if v is the leftmost vertex of a polygon P of W. If it is, we can decide immediately if P is a maximum by finding whether $h(v)$ and $l(v)$ are linked. If they are, P is not a maximum and all its vertices are deleted from Q. Otherwise, P is a maximum. Actually, since nonmaxima are removed as soon as their leftmost vertex is encountered, the polygon containing v is a maximum in all the other cases (i.e., when v is not a leftmost vertex). Then we can simply update the ordering R with the functions *insert* and *delete*, as well as the linked pairs with the

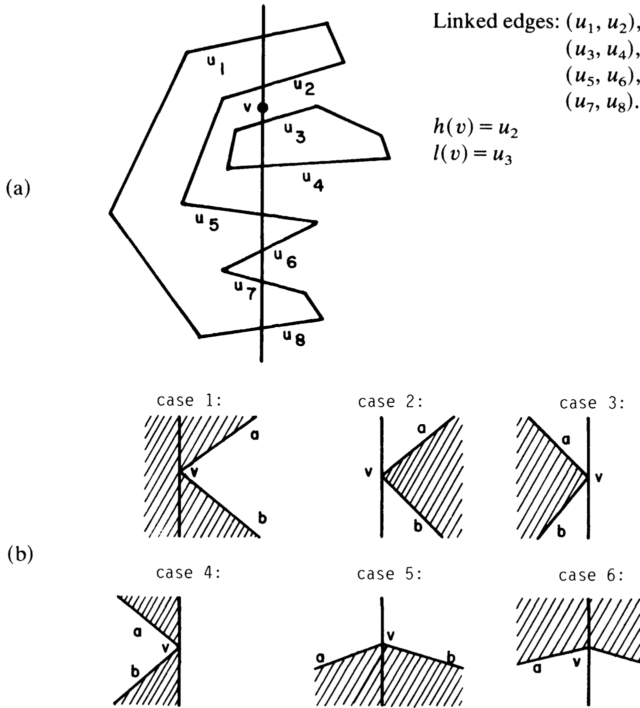


FIG. 10. a) *The ordering R.* b) *The algorithm for computing maxima.*

functions *link* and *unlink*. This is fairly straightforward and the algorithm we next present is self-explanatory.

```
MAXIMUM(W)
Q = Set of vertices in W stored in order
by x-values.
R = ∅.
for all v in Q (in ascending x-order)
   begin
   Let P be the polygon to which v belongs.
   if v is the leftmost vertex of P
     and h(v), l(v) are linked
     then "P is not a maximum"
           delete all vertices of P from Q
     else "P is a maximum"
           UPDATE(R, v)
   end
UPDATE(R, v)
Let a, b be the two edges adjacent to v.
Switch to the case corresponding to Fig. 10b.
case 1:
  insert (a), insert (b)
  unlink (h(v), l(v))
  link (h(v), a)
  link (b, l(v))
  break
```

```
case 2:
  insert (a), insert (b)
  link (a, b)
  break
case 3:
  delete (a), delete (b)
  unlink (a, b)
  break
case 4:
  delete (a), delete (b)
  unlink (h(v), a)
  unlink (b, l(v))
  link (h(v), l(v))
  break
case 5:
  delete (a), insert (b)
  unlink (a, l(v))
  link (b, l(v))
  break
case 6:
  delete (a), insert (b)
  unlink (h(v), a)
  link (h(v), b)
  break
```

Note that when the algorithm terminates, only the vertices of maxima will remain in Q, thus the maxima can be obtained from Q in $O(n)$ time. To implement the algorithm efficiently, we can store Q as a doubly-linked list with random-access to the nodes, thus allowing constant time deletions. R can be maintained as a balanced tree, so that the functions h, L, *insert*, and *delete* perform in logarithmic time. $Link(u, v)$ will simply set two pointers, one from u to v, and the other from v to u, while $unlink(u, v)$ will remove these pointers. With this implementation, the algorithm requires $O(n \log n)$ time. Note that all the preprocessing needed involves sorting the vertices by X-values and computing the leftmost vertices, all of which also takes $O(n \log n)$ time. □

We can now turn back to the problem of dividing up a polyhedron P. Recall that the intersection of P with the plane T supporting the cut S is in general a set of polygons. These polygons may have holes which may themselves contain other polygons of the same kind. We first compute S, from which we derive P_1 and P_2.

I) *Computing the intersection of P and T.* Consider each face F of P in turn and report all the edges of F which intersect the plane T, yet do not lie in T. This includes all the edges of the inner and outer boundaries. Let $a_1, \cdots, a_k$ denote the intersections of T with these edges, as they appear in sorted order on the line supporting the intersection of F and T. Call u_i the edge of F intersecting T at a_i. Observing that the intersection of T and F is made up of the segments $a_1a_2, \cdots, a_{k-1}a_k$ (Fig. 11b), we set two pointers for each pair (u_{2i-1}, u_{2i}); one from u_{2i-1} to u_{2i} and the other from u_{2i} to u_{2i-1}. Iterating on this process for all faces of P will eventually provide doubly-linked lists for all the boundaries of the polygons of the intersection of P and T. Let U denote this set of boundaries. Since each edge is considered at most twice, all these operations take $O(p)$ time, except for the sorts, each of which requires $O(p_i' \log p_i')$ time, where p_i' is the number of edges intersecting T involved in the face considered. Since each edge appears on two faces, the sum of all the p_i' is less than or

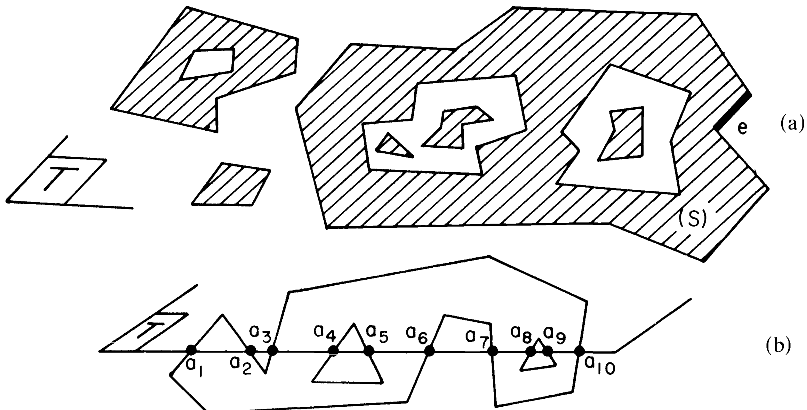


FIG. 11. a) *A cut S.* b) *The edges of S.*

equal to $2p'$, which leads to an $O(p' \log p')$ running time (similarly, p' is the number of edges of P intersecting T). Note that the conversion of the doubly-linked lists of u_i into lists of a_i is straightforward in general. Some special cases may yet be encountered, when a_i is the endpoint of u_i and several edges are adjacent to a_i. It is easy to see, however, that those cases can be handled separately without altering the total running time of the algorithm, which is $O(p+p' \log p')$.

II) *Computing S.* To begin with, we determine the outer boundary of S, denoted S^*, by identifying the boundary in U which contains the edge e. To find the inner boundaries is somewhat more involved. We first form the subset W of U consisting of all the boundaries which lie inside S^*. To do so, we can use a variant of the algorithm MAXIMUM used in the proof of Lemma 6.

Q is still the set of all vertices in U, ordered by X-values. The ordering R, however, will now involve the edges of S^* only. As before, the main loop sweeps a vertical line left-to-right passing through each vertex in Q. If v belongs to S^*, we simply maintain the ordering R with the function UPDATE defined earlier. Otherwise, we observe that the boundary in U which contains v lies inside S^* if and only if $h(v)$ and $l(v)$ are distinct from 0 and are linked. Thus, we know whether a boundary belongs to W or not as soon as we examine its leftmost vertex. To make the algorithm more efficient, we can thus delete all the vertices of the boundary from Q, after examining its first vertex. Like its look-alike, MAXIMUM, this algorithm requires $O(k \log k)$ time, where k is the total number of vertices in Q. Since each of these vertices corresponds to a distinct edge of P, the running time is $O(p' \log p')$.

```
Q = Set of vertices in U sorted by x-values.
R = W = Empty set.
for all v in Q (in ascending x-order)
  begin
  if v belongs to S*
    then UPDATE (R, v)
    else Let B be the boundary in U
         containing v.
         delete all vertices of B from Q.
         if h(v) and l(v) are not 0
           and are linked
           then "v lies inside S*"
                W = W ∪ {B}
  end
```

We are now ready to apply the result of Lemma 6 to the set W. This will give us exactly all the inner boundaries of S, with a total running time of $O(p' \log p')$.

III) *Computing* P_1 *and* P_2. The last step is to compute a graph representation of P_1 and P_2. This is a trivial graph transformation, and we only sketch out the procedure. Let Adj (w) be the adjacency list of the vertex w in the graph representation of P. Also, call E the set of edges of P passing through the vertices of S. We can assume E to be readily available, since the edges in E must be determined in order to compute S. Let w be an endpoint of some edge in E. Defining P_1 as the polyhedron cut by S that contains w, we next show how to compute P_1 in $O(p)$ time.

1) *Adjacency lists of* P_1. For each edge ab of E which does not lie on T, let v be the unique vertex of S lying on ab. We can always assume that a lies on the same side of T as w, that is, is a vertex of P_1 whereas b is a vertex of P_2. If v is distinct from a, we replace b by v in the list Adj (a) and delete the list Adj (b). If $v = a$, we simply delete b from Adj (a) as well as the list Adj (b). Repeating these operations for all the edges of E which do not lie on T has the effect of disconnecting P_1 from P_2. Then, a depth-first search in the resulting graph of P, starting at w, will provide all the vertices of P_1. All the adjacency lists of the vertices common to P and P_1 have already been updated. Finally, since we have a doubly-linked-list description of the boundaries of S, we can set up the adjacency lists of the new vertices, that is, the vertices of P_1 lying on S. All these operations require $O(p)$ time.

2) *Face-to-edge lists of* P_1. Since the previous lists provide the set of vertices of P_1, we first remove all the faces of P made up entirely of vertices not in P_1. Then, since all the faces of P intersecting S have been previously determined, it is easy to compute a description of the parts of those faces which lie in P_1. Let F be such a face, with $a_1, \cdots, a_k$ being the vertices of S lying on F. Recall that $a_1, \cdots, a_k$ have been computed in sorted order (Step I). We may assume that the boundaries of F are represented by doubly-linked lists with the nodes representing the vertices. Letting u_i be the edge of F passing through a_i and b_i be the endpoint which lies on the same side of T as w, we first delete from the lists all the vertices lying strictly on the other side of T, then we enter the vertices a_i into the lists by linking both ways b_i and a_i as well as a_{2i-1} and a_{2i} (Fig. 12a). Note that we can always assume that u_i does not lie on T, which ensures that b_i is always well-defined. The result of these operations may produce several disconnected lists, since F may be broken up into several faces of P_1. Finally, if F has some edges lying on T, the algorithm may produce lists consisting of two vertices, and these degenerate cases should be removed in a postprocessing stage (Fig. 12b). Finally, the face-to-edge lists of S (which have already been computed) must join the set of face-to-edge lists of P_1. Once again, all these operations will take $O(p)$ time.

3) *Edge-to-face lists of* P_1. These lists can be obtained in $O(p)$ time by scanning through the face-to-edge lists once and recording the faces next to each of their boundary edges.

The computation of P_1 and P_2 is now complete. We conclude:

LEMMA 7. *A polyhedron P of genus 0 can be partitioned with a cut in time $O(p + p' \log p')$, using $O(p)$ storage, with p' being the number of edges in P intersecting the plane supporting the cut.*

We have seen that in the course of its action, the naive decomposition may produce polyhedra containing holes. For that reason, we wish to generalize the previous result to polyhedra of arbitrary genus. Now, instead of breaking P into two pieces, a cut may simply decrease its genus by one or have some of the effects described at the beginning of § 2.1 (e.g., removing a handle and creating another). To handle these cases, we may first *cut* each edge of P which intersects S, by updating the adjacency

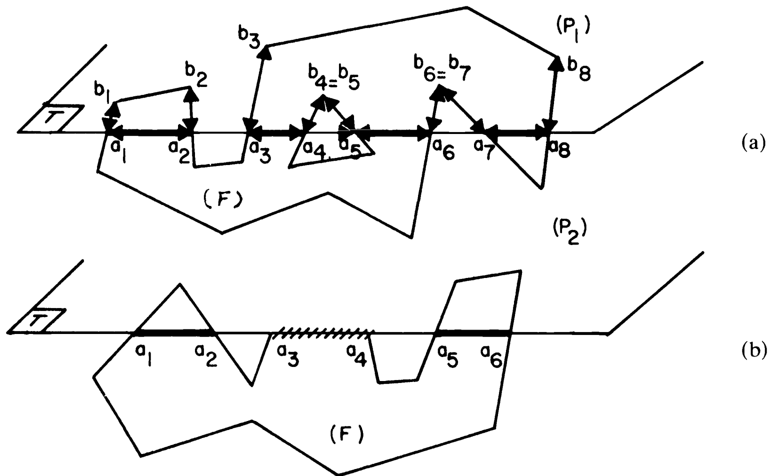


FIG. 12. *Computing the faces of* P_1.

lists accordingly. Next, we test the connectivity of the graph by doing a depth-first search with the adjacency lists. If it is no longer connected, the cut breaks P into two separate pieces P_1 and P_2 which can be computed as indicated above. Otherwise, we update the lists of the representation in a similar way; the only major difference being the introduction of two faces corresponding to the cut. We may omit the details of these operations which are very elementary.

In our analysis, we were careful to use the number of edges p and not the number of vertices as the measure of the input size. Indeed, Euler's formula, which relates the number of vertices, edges, and faces of a polyhedron has to be altered for higher genuses [6]. Consequently, the well-known inequality $p \leqq 3n-6$, which holds for 0-genus polyhedra, is no longer valid when it comes to polyhedra with holes, as is the case in our problem. It is, however, easy to verify that the number of edges always gives the size of the description of P, up to within a constant factor. The revised algorithm for the naive decomposition is merely a repeated application of the procedure described above. This leads to the following result.

THEOREM 8. *The naive decomposition of a polyhedron P of genus* 0 *can be done in $O(nN^2(N+\log n))$ time, using $O(nN^2)$ storage.*

Proof. The algorithm proceeds by removing each notch in turn. In an $O(p)$ preprocessing stage, we can assign to each notch a plane resolving its reflex angle. Then, for each notch in turn, we remove each of its subnotches with cuts lying in the plane associated with the notch. This will produce $O(N^2)$ convex parts in the end, as has been shown in Theorem 2. Each cut can be implemented with the procedure of Lemma 7 and the generalization for higher genuses which we just mentioned. Consider the partial decomposition before the notch g is removed. Let $P_1, \cdots, P_k$ be the (nonconvex) polyhedra in the current decomposition which contain a segment of g as a subnotch (we have seen that $k \leqq N$). Let p_i be the number of edges in P_i and p_i' the number of edges intersecting the plane supporting the cut used to remove g. From Lemma 7, we know that we can remove the subnotch of g in P_i in time $O(p_i + p_i' \log p_i')$. We next evaluate the maximum number C of edges present at any time in the decomposition. We distinguish two kinds of edges: first the edges which are pieces of edges of P. Since each edge of P can be divided into at most $N+1$ segments, the number C_1 of such edges cannot be greater than $p \times (N+1)$.

The other edges are intersections of cuts with faces (or parts of faces) of P or intersections between cuts. Since each cut lies on any of N possible planes, and all faces of P lie on q possible planes, the C_2 edges we are now considering lie on at most qN possible lines. Next we show that each of these lines supports at most $3N$ edges (we do not believe that this upper bound is tight). Let L be such a line and $u_1, \cdots, u_t$ be the edges of the decomposition that lie on L. The edges $u_1, \cdots, u_t$ form m disconnected segments $r_1, \cdots, r_m$ on L, each segment consisting of contiguous edges

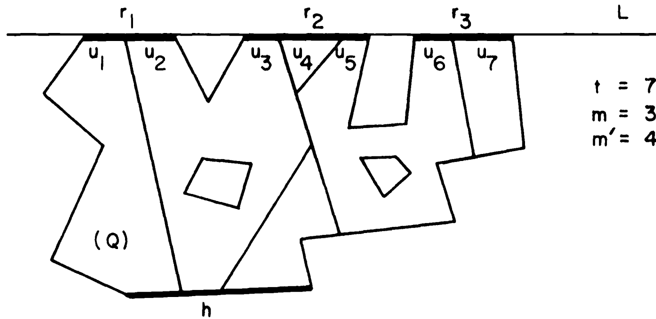


FIG. 13. *Counting the number of edges in the decomposition.*

u_i $(1 \leqq m \leqq t)$ (Fig. 13). Let m' be the number of endpoints common to two consecutive u_{ij}; we have

(1) $$m + m' = t.$$

L is the line passing through the intersection of a cut S with a face of P or the intersection of two cuts S and S'. In either case, let h be the notch passing through the cut S. The union of all the cuts used to remove h forms a polygon Q, which may possibly have holes. Moreover all the segments r_i are edges of Q and each notch of Q corresponds to a distinct notch of P. At this point, we must anticipate a little and use a result which we will prove at the end of this section (Lemma 10). This result states that the line L cannot intersect Q in more than $2N$ segments. Therefore we have

(2) $$m \leqq 2N.$$

Since the *interior* endpoints are all intersections of cuts with L, we also have

(3) $$m' \leqq N.$$

Combining (1)–(3) shows that $t \leqq 3N$, which proves our claim and implies that

$$C_2 \leqq 3qN^2.$$

Since each edge of P is adjacent to at most 2 faces of P while a face has at least 3 enclosing edges, we have

$$3q \leqq 2p$$

showing that

$$C_2 = O(nN^2)$$

since $p = O(n)$ (P is of genus 0). Our counting argument considered each u_i as the intersection of a cut or a face with a cut. Therefore each edge u_i will be counted exactly twice in $p_1 + \cdots + p_k$, hence

$$p_1 + \cdots + p_k \leqq C_1 + 2C_2.$$

Finally, since $C_1 \leqq p(N+1)$ and $p = O(n)$, we have

$$p_1 + \cdots + p_k = O(nN^2).$$

Also, since at most 2 edges intersecting a given plane in a single point can be collinear, the maximum number of edges which can intersect a given plane is bounded by the maximum number of lines L, therefore

$$p'_1 + \cdots + p'_k = O(nN).$$

It follows that all the subnotches of g can be removed in time $O(nN(N + \log n))$, using $O(nN^2)$ storage. Since N notches must be removed, the proof is now complete. □

It is possible to improve the running time of the algorithm to $O(nN^3)$, using the same amount of storage. The algorithm is too long and too complex to be presented here, given the relatively minor gain it represents. We, therefore, refer the reader to [1] for a detailed description of the method.

THEOREM 9. *The naive decomposition of P can be carried out in $O(nN^3)$ time and $O(nN^2)$ space.*

Proof. See [1]. □

We will now prove the claim made earlier that L intersects Q in at most $2N$ segments.

LEMMA 10. *Let N be the number of reflex angles in a nonconvex polygon Q with any number of holes in it. No line L can intersect Q in more than 2N segments.*

Proof. We will prove the lemma in two parts: first assume that all of Q lies on one side of L. Assume wlog that L is horizontal and that Q lies below L. Although all the vertices of Q that lie on L are collinear, we can assume that among the other vertices, no two lie on a common horizontal line. This is only desirable for the sake of simplicity and does not restrict the generality of the problem in any way. Let $s_1, \cdots, s_k$ be the segments of $Q \cap L$ in left-to-right order, and let $v_1, \cdots, v_l$ be a list of the vertices of Q lying strictly below L, sorted vertically in descending order. If we translate the line L downwards along a vertical axis in a continuous motion, we observe that the segments s_i undergo continuous transformations. New segments may appear in the process, some may vanish from L, while others may merge. Eventually all of them will disappear from L. The crucial observation is that since Q is connected, no s_i will disappear before merging at least once. Therefore there will be at least $k/2$ merges in the process (actually, it would be easy to show that there will be at least $k-1$ merges). Note that the merges can occur only when L reaches a vertex v_i. Let L_i be the corresponding position of L (i.e. the horizontal line passing through v_i). Since all the v_i have distinct Y-coordinates, at most one merge can occur at L_i. Suppose that a and b are two segments merging on L_i. The endpoint common to both segments, v_i, is clearly a notch of Q, therefore Q has at least as many notches as we have merges, i.e. $k/2$, provided that $k > 1$.

Assume now that L may intersect Q in an arbitrary fashion, and let $s_1, \cdots, s_k$ be the intersecting segments. Let us *cut* along each segment s_i. This operation partitions Q into at most $k+1$ polygons, each lying entirely on one side of L, as in the previous case. Note that we may have strictly fewer than $k+1$ polygons if Q has holes. Also, since Q is connected, each segment s_i is the edge of at least one polygon which has at least another edge collinear with L (assuming that $k > 1$). It follows that among these polygons we can find j of them, say, $Q_1, \cdots, Q_j$, such that each has at least two edges collinear with L and each s_i is an edge of at least one of them. Let N_i be the number of reflex angles in Q_i and let k_i be the number of edges collinear with L. Since

Q_i has at least two edges adjacent to L, we can use the previous result to derive $k_i \leqq 2N_i$. Since $k_1 + \cdots + k_j \geqq k$, and all the reflex angles of Q involved in the j quantities $N_1, \cdots, N_j$ are distinct, we have $k \leqq 2N$, which completes the proof. □

5. Conclusions. The contribution of this work has been to describe a heuristic for decomposing a polyhedron into a set of convex pieces, with the cardinality of this set lying within a constant factor of the minimum in the worst case. We have also established a quadratic lower bound on the complexity of the minimum convex decomposition problem in three dimensions. Refinements of the algorithm given in this paper might take into account the particular shapes that most *practical* polyhedra are likely to have. For example, it is often the case that two notches will be adjacent and can be removed with the same cut. This simple observation may reduce the number of convex parts by half. More generally, we believe that efficient special-purpose heuristics could be developed along these lines. An interesting case is to restrict the domain of polyhedra to *architectural* designs where, for example, all the edges lie on three possible perpendicular directions. Another restriction may further require that the convex parts be rectangular parallelepipeds. All these problems are highly practical, yet still open.

Only in two and three dimensions is the concept of nonconvex polyhedra totally natural. In higher dimensions, convex polyhedra are still easily expressed as intersections of halfspaces, but nonconvex polyhedra do not lend themselves to such easy descriptions. One method is to express a polyhedron as a connected union of convex polyhedra. Note that the convex polyhedra may overlap, thus do not necessarily constitute a convex decomposition of the polyhedron. This representation is common in linear programming, when the constraints are expressed by k set of inequalities, and at least one set has to be satisfied. If we can find a convex decomposition of the polyhedron into p parts with $p \ll k$, and if each convex part has relatively few faces, testing the feasibility of a point can be greatly simplified by testing its inclusion in any of the p convex parts. Here again, because of the complexity of the problem (recall that the standard version of the decomposition problem is already NP-hard), only efficient heuristics should be sought.

Acknowledgments. I wish to thank Dana Angluin for many helpful discussions, as well as the referees for their valuable help in improving the presentation of this paper.

REFERENCES

[1] B. CHAZELLE, *Computational geometry and convexity*, Ph.D. thesis, Yale Univ., New Haven, CT, 1980. Also available as Carnegie–Mellon Tech. Report, CMU-CS-80–150, Carnegie–Mellon Univ., Pittsburgh.

[2] B. CHAZELLE AND D. P. DOBKIN, *Detection is easier than computation*, Proc. 12th ACM SIGACT Symposium, Los Angeles, May 1980, pp. 146–153.

[3] H. COXETER, *Regular Polytopes*, 3rd Ed., Dover, New York, 1973.

[4] M. GAREY, D. JOHNSON, F. PREPARATA AND R. TARJAN, *Triangulating a simple polygon*, Inform. Proc. Lett., 7 (1978), pp. 175–180.

[5] A. LINGAS, *The power of non-rectilinear holes*, Proc. 9th Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 140, Springer-Verlag, New York, 1982, pp. 369–383.

[6] W. MASSEY, *Algebraic Topology: An Introduction*, Springer-Verlag, New York, 1967.

[7] J. MUNKRES, *Topology: A First Course*, Prentice-Hall, Englewood Cliffs, NJ 1975.

[8] J. O'ROURKE AND K. J. SUPOWIT, *Some* NP-*hard polygon decomposition problems*, IEEE Trans. Inform. Theory, IT-29 (1983), pp. 181–190.

[9] F. PREPARATA, *A new approach to planar point location*, this Journal, 10 (1981), pp. 473–482.

[10] M. SHAMOS AND D. HOEY, *Geometric intersection problems*, 17th Annual IEEE Conference on Foundations of Computer Science, Houston, TX, Oct. 1976, pp. 208–215.

[11] G. B. THOMAS, JR., *Calculus and Analytic Geometry*, Addison-Wesley, Reading, MA, 1962.

SIAM J. COMPUT.
Vol. 13, No. 3, August 1984
© 1984 Society for Industrial and Applied Mathematics
004

CONSISTENCY AND SERIALIZABILITY IN CONCURRENT DATABASE SYSTEMS*

D. J. ROSENKRANTZ†, R. E. STEARNS‡ AND P. M. LEWIS II§

Abstract. The main results of this paper show that serialization is both necessary and sufficient for consistency in concurrent database systems. This is true for both the final database and the views of the database seen by individual transactions. The model of a transaction includes both read and write operations which may be performed in any order (except an entity must be read before being written).

The main results are presented in terms of an information flow model describing the source of each value read and the use of each value written. Since the model does not involve any concept of the "time" a value was read or written, it models any concurrency system producing information flow among transactions.

There is a section discussing the effect of changing the model to include write operations without preceding reads, and a section discussing the restriction to straight-line programs.

Key words. database, concurrency control, consistency, serialization, transaction

1. Introduction. There has been a lot of activity in the area of database concurrency controls. The goal of concurrency control is to allow transactions accessing a common database to run as concurrently as possible without destroying database consistency or preventing a transaction from eventually running to completion. It has generally been appreciated that consistency can be insured by designing a serializable control, where serializability means that the effect of running transactions concurrently is the same as if the transactions have been run in some serial order. Many practitioners have in fact made serializability a design requirement.

This paper investigates the relationship between serializability and consistency. We first develop a general concurrency control model based on information flow between transactions. We then show that serializability is both necessary and sufficient for consistency. (There is a small loophole for read-only transactions.) We consider both the consistency of the final database produced by the transactions and the consistency of the view seen by each transaction.

Our concurrency model is developed to reflect assumptions we believe appropriate for concurrency controls in mainstream commercial database systems. These assumptions and our reason for making them are as follows:

Assumption A1. The control can distinguish between a read access and a write access. *Reason.* Data manipulation languages have this feature.

Assumption A2. The control does not know the consistency criterion. *Reason.* In practice, consistency conditions are too complex to expect a user to write them down (or even fully comprehend them).

Assumption A3. The control does not make inferences from the particular values read or written. *Reason.* Because of Assumption A2, this information is, for practical purposes, useless. (In theory, inferences could be made from testing values for equality.)

* Received by the editors September 26, 1980, and in final revised form August 29, 1983.

† Department of Computer Science, State University of New York at Albany, Albany, New York 12222. The research of this author was supported in part by the National Science Foundation under grant MCS 78-03157.

‡ Department of Computer Science, State University of New York at Albany, Albany, New York 12222. The research of this author was supported in part by the National Science Foundation under grant MCS 79-03770.

§ General Electric Corporation, Research and Development Center, Schenectady, New York 12301.

Assumption A4. The control may respond to a read request with a value other than the last value written. *Reason.* Concurrency control designs have been proposed which have this feature [1], [2], [13], [14], [15].

Assumption A5. A value written in the database during the run of the transaction must be considered functionally dependent on all values read, rather than functionally dependent only on those read by the transaction before writing the value. *Reason.* Data manipulation languages usually permit branching and rewriting values. The control must assume a value written might have been rewritten if values read subsequently had been different.

Assumption A6. Before a transaction can write an entity, it must read the entity. *Reason.* The "necessity" results are false without this assumption. (See § 11.)

Assumption A7. There are no "lost updates." More precisely, the history of changes to a given entity is a sequence of changes, each change overwriting the change made by the preceding transaction in the sequence. *Reason.* Lost updates are usually considered undesirable. Also, the "necessity" results are false without this assumption. For instance, consider a concurrency control that presents each transaction with the original contents of the database, and, when a transaction terminates, throws out the values it wrote. This concurrency control preserves consistency (since the final database is identical to the initial database), but is not serializable.

We believe Assumptions A1 to A5 to be both reasonable and desirable. Assumption A6 is also reasonable in that most state-of-the-art database systems interface with the operating system and concurrency control by first reading a page from the disk and then perhaps writing the page. Once Assumption A6 is made, we believe Assumption A7 to be both reasonable and desirable. It implies that all but one version of each entity is overwritten. The single version that is not overwritten can be thought of as being retained in a final database produced by the transactions. Therefore every version of the entity is actually "used" in the sense that it is either overwritten or else is the unique "surviving" version of the entity.

When we say that serialization is necessary, we mean that in all nonserializable situations, there could be (from the viewpoint of the concurrency control) an inconsistency. Our results do not exclude the possibility that for specific consistency criteria or for specific transactions, consistency may be preserved in nonserializable situations, and indeed such cases have been considered in the literature [6], [7], [8], [9], [11].

This paper addresses two consistency questions:

1) Under what conditions is the final database consistent?

2) Under what conditions does an individual transaction see a consistent view of the database?

The second question is very important for several reasons:

a) The view of the individual user is the view seen by transactions. A report produced by a transaction which sees an inconsistent view might be regarded by a user as evidence that the database is being mismanaged.

b) A transaction may not be properly designed to accommodate "impossible" data, and may behave unpredictably when given an inconsistent view.

c) In a system which is always running transactions, there may never be a well-defined "final database," and consistency for individual transactions may be the only meaningful concept.

Previous papers on serialization have concentrated on question 1. Question 2 requires more complex proof techniques because inconsistency must be demonstrated using only that portion of the database that a transaction sees.

Most papers on serializability use the concept of a "schedule of accesses" (or history or log). This concept is not adequate here because of Assumption A4. Instead we use the concept of a "version graph" showing information flow. The version graph might appear inadequate because it does not show the order in which an individual transaction makes its requests. However, the order of requests is irrelevant because of Assumption A5. We examine this issue more closely in § 12.

The early work of [5] is based on "schedules." A *schedule* is *defined* to be consistent if it is serializable, and a database *state* is *defined* to be consistent if it satisfies a set of consistency constraints. The authors note that if the initial database state is consistent and if each transaction transforms a consistent state into a consistent state, then serializable schedules maintain consistency. From a schedule, they construct a "dependency" relation that is similar to the "augmented version graph" of this paper, and show that the schedule is serializable if and only if the dependency relation is acyclic.

Kung and Papadimitriou [10] show that for systems with only one type of access, which is a combined read-write access, a "schedule" maintains full database consistency if and only if it is serializable. They do not address the consistency of the view seen by individual transactions. The model in [10] differs from ours in Assumptions A1 and A5. In each case, we are making the more general assumption.

Casanova [3] and Casanova and Bernstein [4] study consistency in a model where all reads occur in a single combined access and then all writes occur in a single access. This does not permit all the possibilities of information flow permissible under our model. However, unlike our model, theirs allows the write to include entities which were not read.

The results in this paper improve an earlier version announced in [14, Thm. 1] without proof.

2. Concurrency control. In this section, we describe in nonmathematical terms our concept of a concurrency control. In later sections, we formalize those aspects of the control that pertain to consistency.

We think of a *transaction* as a computer program that reads information from and writes information into a database. The interface to the database is through system procedures READ and WRITE. A call on READ is referred to as a "read request" and a call on WRITE is referred to as a "write request."

If transactions were run on a system one at a time, read requests could be responded to by reading the value from the database, and write requests by replacing the old database value with the new. The problem comes with systems which attempt to run a number of transactions concurrently. The part of the system which determines the response to the read and write requests is called a *concurrency control.*

The concurrency control can grant read requests by supplying an input, and can grant write requests by saving the output somewhere. These values are not necessarily read or written directly into the database, as the control may sometimes have a temporary need to remember several different values for a single entity.

If a read request by a given transaction on a given entity is not the first request by that transaction on that entity, and if the concurrency control grants the read request, then the value supplied to the transaction is assumed to be the value associated with the preceding request on the entity by the transaction. If the preceding request was a write request, the value written is supplied to the new read request. If the preceding request was a read request, the value supplied to that request is assumed to be also supplied to the new read request. Thus the only time a concurrency control

must make a choice as to what value of an entity to supply when granting a read request is when the read request is the first request of the transaction on the entity. The value that the concurrency control is allowed to supply is either the value of the entity in an initial database, or the final value written by some other transaction that wrote the entity.

Database consistency deals with data values, and is independent of other aspects of the concurrency control. Thus we base our mathematical treatment of consistency on a model of data flow, rather than a model of concurrency control. As with any mathematical model, the appropriateness for the intended application is an issue to be addressed. Towards this end, we will relate our formal definitions to the above notion of a concurrency control and to Assumptions A1 to A6 of the introduction.

3. Databases. We now present our formal concepts of databases and database consistency.

DEFINITION 3.1. A *database* is specified by a pair (E, V) where E is a set of entities and V is a set of *values*. A mapping from E into V is called a *database state*.

In practice, a database entity could be an item, record, page, or file, depending on at what level a given system applies a concurrency control discipline. If s is a database state and e is an entity, then $s(e)$ is interpreted as the value stored in entity e.

DEFINITION 3.2. A *consistency criterion* for a database (E, V) is a set C of database states. A database state s in C is said to be *consistent*.

This definition of consistency allows for an arbitrary classification of database states being "consistent" or "inconsistent." There is no need to compute C, and in fact our interest is in controls which work for arbitrary criteria (Assumption A2).

It is our opinion that this definition is too weak to capture all aspects of preserving consistency. However, it serves the purposes of this paper very well, since this weak definition already gives the strongest possible result, namely that serialization is necessary.

4. Transactions. A given run of a transaction reads certain entity values and writes certain entity values. To study consistency, the run can be represented mathematically by the entities and entity values read and written. We call this mathematical object a "transaction effect." Unlike most other authors, we do not put the order of reads and writes into our model. The reason is Assumption A5, which implies the order is not relevant.

DEFINITION 4.1. Given a database (E, V), a *transaction effect* is specified by a four-tuple (READSET, WRITESET, r, w) where

a) READSET is a nonempty subset of E,

b) WRITESET is a subset of READSET,

c) r is a function r: READSET $\rightarrow V$ called the *input function*, and

d) w is a function w: WRITESET $\rightarrow V$ called the *output function*.

READSET represents the set of entities read by the run of the transaction and function r gives the values read. We rule out the trivial case where no entities are read, since we are only concerned with transactions that interact with the database. WRITESET represents the set of entities written by the run of the transaction and function w gives the values written. We require that WRITESET be a subset of READSET to conform to Assumption A6.

A given transaction can produce a variety of effects, depending on the values supplied in response to its read requests. Even the READSETs and WRITESETs can vary because the transaction can branch on values read.

DEFINITION 4.2. Given a database state s and a transaction effect $\sigma =$ (READSET, WRITESET, r, w) for database (E, V), we say that σ is *matched to* s if and only if $r(e) = s(e)$ for all e in READSET. We say that transaction effect σ *transforms* database state s to database state t if and only if σ is matched to s and

a) $t(e) = w(e)$ for e in WRITESET

b) $t(e) = s(e)$ otherwise.

The idea behind this definition is that when a transaction that can produce effect σ is run with a database state s matched to σ, the transaction changes the values in database state s to obtain database state t. If this same transaction were run with a database state u not matched to σ, then a different unspecified transaction effect would occur, one matched to u.

DEFINITION 4.3. Given a database (E, V) and a consistency criterion C, a transaction effect σ is said to be *valid* if and only if, for all database states s such that s is in C and σ is matched to s, transaction effect σ transforms s to a database state in C.

Intuitively, a debugged transaction can only produce valid transaction effects. If a debugged transaction terminates when run by itself with a consistent initial database state, the final database state when it terminates is also consistent. No matter what the consistent initial database state matched to the transaction effect is, the transaction effect transforms this consistent state into another consistent database state.

Note that a given transaction effect may not be matched to any consistent database state. From Definition 4.3, such a transaction effect is valid. It could be the effect of a debugged program presented with an inconsistent database (i.e. garbage in, garbage out).

The running of a sequence of transactions produces a sequence of transaction effects in the obvious way:

DEFINITION 4.4. Let $\sigma_1, \sigma_2, \cdots, \sigma_n$ be a sequence of transaction effects and let s_0 be a database state. The sequence is called a *serial run on* s_0 if and only if there exist database states $s_1, s_2, \cdots, s_n$ such that for $1 \leqq i \leqq n$, σ_i is matched to s_{i-1} and transforms s_{i-1} to s_i. Database state s_n is called the *result* of the serial run.

Note that if the s_i exist, they are unique. Thus, if the result exists, it is unique. The well known fact that serial runs preserve consistency is expressed in our notation as follows.

THEOREM 4.5. *If* $\sigma_1, \cdots, \sigma_n$ *is a serial run of valid transaction effects on consistent state* s_0 *and* $s_1, \cdots, s_n$ *are database states as given in Definition* 4.4, *then all the* s_i *are consistent.*

Proof. By induction on i using Definition 4.4. □

5. Version graphs. The outcome of concurrency decisions is a flow of information among transactions. We model this flow on two levels. One level is the "version graph" to be defined in this section. The other (more detailed) level is the "datatrace" defined in § 7.

The version graph represents those facts about concurrency decisions available to the concurrency control. These are the facts the concurrency control can use to assure that consistency is maintained or that a given transaction sees a consistent view of the database. In particular, these facts do not include knowledge of specific entity values (Assumption A3) or what the consistency criterion is (Assumption A2).

The "version graph" is a mathematical concept that models how versions of entities flow between transactions. The version graph has a node for each transaction, plus an extra node for the initial database state. The edges reflect the source of entity values read or overwritten by the transactions.

DEFINITION 5.1. Given a set of entities E, a *version graph* G for E is a directed graph with a finite number of nodes such that:

a) there is exactly one node I having no entering edges;
b) each edge is labelled R_e or W_e where e is in E;
c) for each node x and each e in E, node x has at most one entering edge labelled R_e and at most one entering edge labelled W_e;
d) if there is an edge from node x to node y labelled W_e, then there is also an edge from x to y labelled R_e;
e) if there is an edge labelled R_e or W_e leaving node x, then either x is the node I or x has an entering edge labelled W_e;
f) for each e in E, the edges labelled W_e form a chain (i.e. a cycle free path, possibly of zero length) beginning at I.

The node I with no entering edges represents the initial database state. Any other node represents a transaction, and will be called a *transaction node.* A transaction node with an entering edge labelled W_e for some e is called a *writing transaction node,* and is considered to have overwritten entity e. A transaction node that is not a writer is called a *read-only transaction node.* A node that is either I or a writing transaction node is said to be a *producer node,* and is considered to produce the value of some entity. Producer node I is considered to produce an initial value for each entity. A writing transaction node q with an entering edge labelled W_e is considered to produce a value for entity e. The entity value produced by q is considered to overwrite the value produced by the node exited by the edge (from Definition 5.1(e) this node produces a value for entity e).

As an example, Fig. 1 shows a version graph with $E = \{\alpha, \beta, \gamma\}$. For convenience, the transaction nodes have been labelled with transaction names. From the graph it is evident that transaction a reads and writes entity set $\{\alpha\}$, transaction b reads $\{\alpha, \gamma\}$ and writes $\{\gamma\}$, and transaction c reads $\{\alpha, \gamma\}$ and writes the null set. Entity β is not accessed. The edge labelled R_α from a to c means that c reads the version of entity α that transaction a wrote.

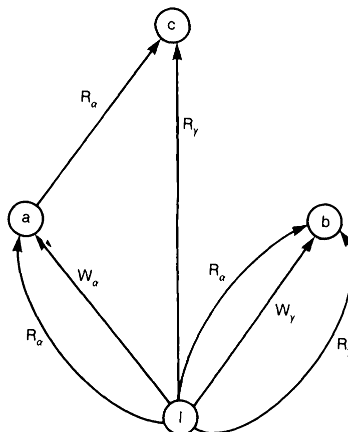


FIG. 1. *Version graph.*

The edges of a version graph can be considered to represent "information flow relations." For each e in E there is a relation R_e such that xR_eq holds if the version graph contains an edge labelled R_e from node x to node q, and signifies that the value of e produced by x was read by q. Relation xW_eq holds if the version graph contains an edge labelled W_e from node x to node q, and signifies that the value of e created by x was changed by q.

Each condition of Definition 5.1 has an interpretation in terms of the application we are modelling. Definition 5.1(a) reflects the inclusion of the initial database state in the version graph, and the assumption that each transaction actually accesses the database. Definition 5.1(b) merely says that an entity is associated with each read or write. Definition 5.1(c) reflects the assumption that the concurrency control need only supply an appropriate entity value when a transaction makes its initial request, since it supplies each subsequent request with the value from the preceding request. Definition 5.1(d) reflects Assumption A6. Definition 5.1(e) merely says that the source of each entity value is actually a producer of a value for that entity. Definition 5.1(f) reflects Assumption A7. The single version of an entity at the end of a chain is the version that is retained in a final database state produced by the transactions in the graph.

6. Version graph analysis. In the preceding section, version graphs were used to model the information flow between transactions. We will subsequently characterize consistency in terms of version graph analysis. The key concepts behind this analysis are "individual version graphs," "writers version graphs" and "augmented version graphs." These concepts, together with certain lemmas used in later sections, are developed below.

DEFINITION 6.1. Given a directed graph G, node x is called an *ancestor* of node y if and only if there is a directed path (perhaps of zero length) from x to y. Given a subset S of nodes for G, the *ancestor subgraph* for S is the graph with

a) node set A consisting of ancestors of nodes from S, and

b) edge set consisting of all edges from G connecting nodes in A.

An important fact about an ancestor subgraph of a version graph is that it is a version graph:

LEMMA 6.2. *If G is a version graph for entity set E, and H is a nonempty ancestor subgraph of G, then H is a version graph for entity set E.*

Proof. We check H for each condition of Definition 5.1. For Definition 5.1(a), first note that since each node of H has the same number of entering edges as the corresponding node of G, H has at most one node with no entering edges. Next, note that from Definition 5.1(e) and (f), node I is an ancestor of every node of G. Since H is nonempty, node I is included in H, and so H satisfies Definition 5.1(a).

Definition 5.1(b) and (c) are obvious because the edges in the ancestor subgraph H are a subset of the edges in version graph G.

Definition 5.1(d) and (e) hold because the required edges are inherited from G.

For Definition 5.1(f), observe that the chain for e in H must have a node x which is maximal distance from I on the chain. Because H is an ancestor subgraph, the nodes between x and I are also in H, so the W_e edges in H also form a chain. □

A node of an ancestor subgraph participates in the same chains as in the original graph:

LEMMA 6.3. *Let x be a node common to version graphs G and A where A is an ancestor subgraph of G. Then node x is the kth node on the chain for entity e in version graph G if and only if x is the kth node on the chain for e in version graph A.*

Proof. The portion of the chain from I to x is the same in both graphs. □

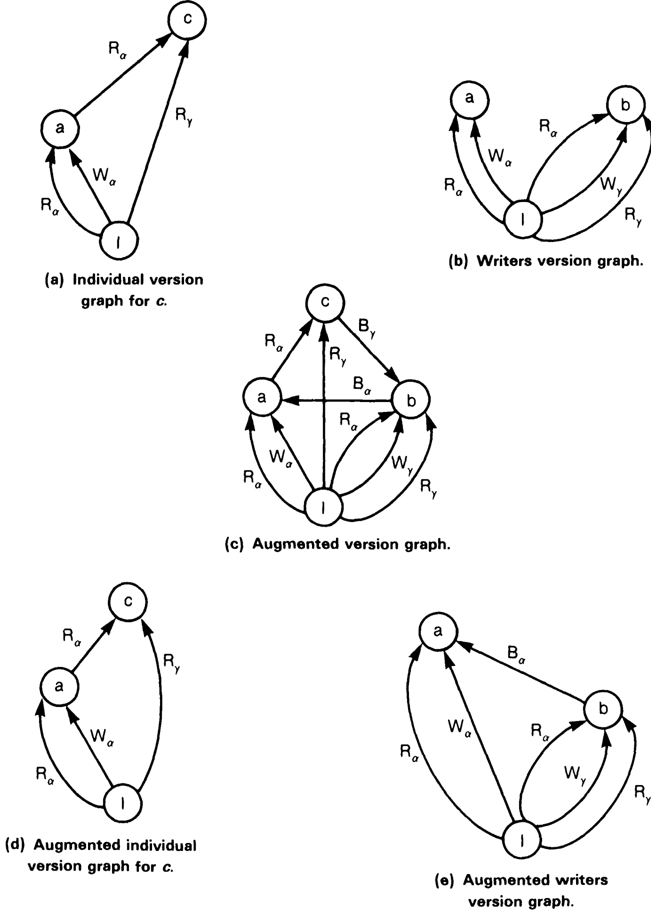
Certain ancestor subgraphs, defined below, play a key role in database consistency.

DEFINITION 6.4. Let G be a version graph. For each node x of G, the ancestor subgraph of G for $\{x\}$ is called the *individual version graph* (ivg) for x in G, and is denoted as ivg (G, x).

Let WRITERS be the set of nodes of G having an entering edge labelled W_e for some e. Define the *writers version graph* for G to be the ancestor subgraph for WRITERS.

For version graph G and entity e, define chainend (G, e) as the last node in G on the chain (see Definition 5.1(f)) for e. Define ivg (G, e) as ivg $(G$, chainend $(G, e))$.

As an example, consider the version graph, G, of Fig. 1. The individual version graph for transaction c, i.e. ivg (G, c), is shown in Fig. 2(a). The writers version graph is shown in Fig. 2(b). Also note that chainend $(G, \alpha) = a$, chainend $(G, \beta) = I$, and chainend $(G, \gamma) = b$.



(a) Individual version graph for c.

(b) Writers version graph.

(c) Augmented version graph.

(d) Augmented individual version graph for c.

(e) Augmented writers version graph.

FIG. 2

LEMMA 6.5. *For every version graph G and node x of G;*

a) *Every node of* ivg (G, x) *except possibly x is a producer node;*

b) *Every node of the writers version graph is a producer node.*

Proof. Immediate from Definition 5.1e. □

LEMMA 6.6. *If* $H = \text{ivg}(G, y)$ *and* $K = \text{ivg}(H, x)$, *then* $K = \text{ivg}(G, x)$.

Proof. Node x has the same set of ancestors in G and H. □

We now consider extra edges that can be added to a version graph in order to indicate when one transaction reads an entity version that was overwritten by a second transaction.

DEFINITION 6.7. Given a version graph G and given a database entity e, define the relation B_e among nodes of G by pB_eq if and only if $p \neq q$ and for some node x in G, xR_ep and xW_eq. Define the *augmented graph* for G, denoted as aug (G), to be G with directed labelled edges added for the relations B_e for all entities e.

The relationship pB_eq can be interpreted to mean that transaction p read the value of e that was changed by transaction q. More briefly, p read the value of entity e that existed *before* q overwrote that value. Note that since node I has no entering R_e or W_e edges, node I has no entering or exiting B_e edges.

Figures 2(c), 2(d) and 2(e) show the augmented graphs for Figs. 1, 2(a) and 2(b) respectively.

7. Datatraces. In studying consistency considerations, we are concerned not only with information flow, but also with the actual entity values that are read and written. This is because it is the actual values that determine if the data seen or the database state produced are actually consistent. In the following definition, we combine a version graph with the effects of its transactions to form a "datatrace." The datatrace represents both the flow and values of database information involved in the running of a finite set of transactions.

DEFINITION 7.1. Given a database (E, V), a *datatrace* is a triple (G, σ, s) where:

a) G is a version graph for E;

b) σ is a function that maps each transaction node p of G into a transaction effect $(\text{READSET}_p, \text{WRITESET}_p, r_p, w_p)$;

c) s is a database state (the *initial* state);

d) for each transaction node p in G, READSET_p equals the set of entities e such that an edge labelled R_e enters node p, and WRITESET_p equals the set of entities e such that an edge labelled W_e enters node p;

e) if qR_ep in G then e is in WRITESET_q and $w_q(e) = r_p(e)$, where the notation is extended by defining $\text{WRITESET}_I = E$ and $w_I(e) = s(e)$.

Datatraces lead naturally to a concept of a final database state.

DEFINITION 7.2. Given a database (E, V) and a datatrace (G, σ, s), define the final database state t for the trace to be the function such that $t(e)$ is given by the output function of chainend (G, e) (so that $t(e)$ equals $w_{\text{chainend}(G,e)}(e)$).

We now show that if a version graph is associated with a trace, then each nonempty ancestor subgraph is the version graph of an appropriately defined subtrace of the given trace.

LEMMA 7.3. *If (E, V) is a database, (G, σ, s) a datatrace and G' a nonempty ancestor subgraph of G, and if σ' is σ restricted to the nodes of G', then (G', σ', s) is a datatrace.*

Proof. From Lemma 6.2, G' is a version graph, so condition (a) of Definition 7.1 is satisfied. The other conditions of Definition 7.1 carry over directly from G, σ and s. □

Consider the flow of information when a set of transactions are run in order $p_1, \cdots, p_n$ with each transaction terminating before the next one begins. Describing this flow with a version graph, an edge $p_jR_ep_i$ is included whenever transaction p_i reads the value written by p_j. Because of the sequential order of execution, $p_jR_ep_i$ in the flow graph implies $j < i$ and further implies that e is not in WRITESET of any p_k for $j < k < i$. It is this characteristic that we build into the following definition:

DEFINITION 7.4. A *serialization* of datatrace (G, σ, s) is an ordering $p_0, \cdots, p_n$ of the nodes of G such that

a) $p_0 = I$ and

b) $p_jR_ep_i$ implies

$$j = \max\{k \mid k < i \text{ and } e \text{ is in } \text{WRITESET}_{p_k}\}.$$

A datatrace is *serializable* if it has a serialization.

The "serialization" of a datatrace is thus an ordering which shows that the trace could have resulted from a sequential running of the transactions. A "serializable trace" is a datatrace which has such an explanation.

As defined, serializability is essentially a property of the version graph rather than the whole datatrace. The next result addresses the entire datatrace and relates serializations of datatraces with serial runs.

THEOREM 7.5. *If $p_0, \cdots, p_n$ is a serialization of datatrace (G, σ, s) then*

a) *$\sigma(p_1), \cdots, \sigma(p_n)$ is a serial run on s, and*

b) *the result of the serial run is identical to the final state of the datatrace.*

Proof. (a) The database states required by Definition 4.4 are defined as follows. For $1 \leqq i \leqq n$, let $s_i(e) = w_{p_l}(e)$ where $l = \max\{k | k \leqq i$ and e is in $\text{WRITESET}_{p_k}\}$, and let $s_0 = s$. We will show that σ_{p_i} is matched to s_{i-1} and transforms s_{i-1} to s_i.

To prove that σ_{p_i} is matched to s_{i-1}, we must show that $r_{p_i}(e) = s_{i-1}(e)$ for all e in READSET_i. From Definition 7.1, $r_{p_i}(e) = w_{p_j}(e)$ where $p_j R_e p_i$ in G. By Definition 7.4(b), $j = \max\{k | k < i$ and e in $\text{WRITESET}_k\}$. Obviously $j = \max\{k | k \leqq i-1$ and e in $\text{WRITESET}_k\}$, and so $s_{i-1}(e) = w_{p_j}(e)$ by construction of s_{i-1}.

To prove that σ_{p_i} transforms s_{i-1} into s_i, consider e in WRITESET_{p_i}. From the construction of s_i, $s_i(e) = w_{p_i}(e)$ because the maximum possible value of k satisfying $k \leqq i$ is $k = i$. For e not in WRITESET_{p_i}, $\{k | k \leqq i$ and e in $\text{WRITESET}_{p_k}\} = \{k | k \leqq i-1$ and e in $\text{WRITESET}_{p_k}\}$, so $s_i(e)$ and $s_{i-j}(e)$ both equal $w_{p_l}(e)$ for the same value of l.

(b) The result of the serial run is the constructed s_n. Let t be the final state of the datatrace. For the entity e, the ordering of nodes of the writers chain for e conforms to the serialization order because of Definition 7.4(b). Let $\text{chainend}(G, e) = p_j$. Then $j = \max\{k | k \leqq n$ and e is in $\text{WRITESET}_{p_k}\}$ and $s_n(e) = w_{p_j}(e)$ by construction of s_n. Thus $t(e) = s_n(e)$. □

We now characterize the serializable traces. First, given an acyclic graph, a *topological sort* of the graph is an ordering of the nodes $p_0, \cdots, p_n$ such that (p_i, p_j) an edge implies $i < j$. Note that an acyclic graph may have more than one topological sort. We now show that if the augmented version graph of a trace is acyclic, then any topological sort of the augmented version graph is a serialization of the trace.

LEMMA 7.6. *Let (E, V) be a database and let (G, σ, s) be a datatrace. Suppose the nodes of* aug (G) *can be ordered $p_0, \cdots, p_n$ so that (p_i, p_j) an edge of* aug (G) *implies $i < j$. Then this order is a serialization of the trace.*

Proof. To see that $p_0 = I$, observe that every transaction node of G has an entering edge, and so cannot be the first node in the linear sequence. Thus, Definition 7.4(a) is satisfied.

Now assume that $p_i R_e p_j$ but that the equation in Definition 7.4(b) fails because of a k such that $i < k < j$ and e is in WRITESET_{p_k}. Nodes p_k and p_i are both on the chain for e (Definition 5.1(e)). Each step in the chain must by hypothesis have a larger subscript so there must be a k' such that $p_i W_e p_{k'}$ and $k' \leqq k$. (Node $p_{k'}$ is the next node on the chain after p_i and cannot come after p_k.) But $p_i R_e p_j$ and $p_i W_e p_{k'}$ implies $p_j B_e p_{k'}$ contrary to $k' \leqq k < j$ and the hypothesis. Thus k cannot exist and j must satisfy the condition of Definition 7.4. Order $p_0, \cdots, p_n$ is therefore a serialization. □

We next prove a converse to Lemma 7.6. It says that running transactions serially imposes an order on the nodes of the augmented version graph, an order which agrees with the direction of the edges of the graph.

LEMMA 7.7. *Let (G, σ, s) be a datatrace serialized by ordering $p_0, \cdots, p_n$ of the nodes of G. Then (p_i, p_j) an edge of* aug (G) *implies $i < j$.*

Proof. If $p_i R_e p_j$, the equation of Definition 7.4 requires $i < j$. If $p_i W_e p_j$, then also $p_i R_e p_j$ (Definition 5.1(d)) and again $i < j$.

If $p_i B_e p_j$, this means there is a p_k such that $p_k W_e p_j$ and $p_k R_e p_i$. We have already shown this implies $k<j$ and $k<i$. If $j<i$, then

$$k \neq \max \{l \mid l<i \text{ and } e \text{ is in WRITESET}_{p_l}\}$$

because of the possibility $l=j$. This violates Definition 7.4(b) and $j<i$ has led to a contradiction. Therefore $j>i$ and the lemma is proved. □

THEOREM 7.8. *A datatrace* (G, σ, s) *is serializable if and only if* aug (G) *is acyclic.*

Proof. This follows from Lemmas 7.6 and 7.7. □

Theorem 7.8 is the expression in our model of the well-known result from [5] that a schedule is serializable if and only if the constructed dependency relation is acyclic.

8. Main results. We now present two "if and only if" theorems to support our claims that serializability is equivalent to preserving consistency. The first addresses the issue of preserving consistency of the final database. By assumption, the concurrency control knows G and does not know V, C, σ, or s. The theorem says that consistency of the final database is guaranteed to be preserved if augmented G_w is acyclic, and consistency can be violated if it is not.

THEOREM 8.1. *For all entity sets E and all version graphs G on E, the augmented writers version graph is acyclic if and only if, for all sets of values V, all consistency criteria C on* (E, V) *and all datatraces* (G, σ, s) *such that s is in C and each* $\sigma(p)$ *is valid, the final database state is in C.*

THEOREM 8.2. *For all entity sets E and all version graphs G on E and all transaction nodes i of G, the augmented* ivg *of i is acyclic if and only if, for all sets of values V, all consistency criteria C on* (E, V), *and all datatraces* (G, σ, s) *such that s is in C and each* $\sigma(p)$ *is valid,* $\sigma(i)$ *is matched to some database state in C.*

These theorems are consequences of Theorems 9.1 and 10.1, proven in the next two sections.

Since serializability has been equated with acyclic version graphs by the results of the preceding section, Theorems 8.1 and 8.2 say that serializability is equivalent to preserving consistency.

9. Assuring consistency. We are now prepared to give sufficient conditions for a concurrency control to maintain consistency. Specifically, we combine the "only if" parts of Theorems 8.1 and 8.2 into a single theorem. For readability, the conjunction of the two "only if" parts is expressed in a logically equivalent form with the universal quantifiers moved to the front.

THEOREM 9.1. *Let* (E, V) *be a database, C a consistency criterion for the database, and* (G, σ, s) *a datatrace such that s is in C and transaction effect* $\sigma(p)$ *is valid for each transaction node p of G. If the augmented* ivg *for a given transaction node p is acyclic, then* $\sigma(p)$ *is matched to some database state in C. If the augmented writers version graph for G is acyclic, then the final database state for the trace is in C.*

Proof. Let G' be ivg (G, p) for a given transaction node p, and let σ' be the restriction of σ to the nodes of G'. Then from Lemma 7.3, (G', σ', s) is a datatrace. If aug (G') is acyclic, (G', σ', s) is serializable by Theorem 7.8 and $\sigma(p)$ is matched to a database state in a serial run (Theorem 7.5(a)). Since the transaction effects are valid, Theorem 4.5 says the state in the serial run is consistent.

Now let G' be the writers version graph and σ' again be the restriction of σ. Again (G', σ', s) is a trace and an acyclic aug (G') implies that (G', σ', s) is serializable. Since G' contains all the writers from G, the final database state of (G', σ', s) is identical to the final database state for (G, σ, s) (from Lemma 6.3 and Definition 7.2). Since (G', σ', s) is serializable, Theorem 7.5 says that the final state of the datatrace

is identical to the result of a serial run. Theorem 4.5 implies that the result of a serial run is consistent. □

COROLLARY 9.2. *Given a database consistency criterion and datatrace as in the statement of Theorem* 9.1, *the* $\sigma(p)$ *are matched to consistent database states and the final database state is consistent if the augmented graph for version graph G is acyclic.*

Proof. The augmented individual version graphs and the augmented writers version graph are all subgraphs of augmented G and inherit the acyclic property from G. □

Thus serializability of a trace is sufficient to ensure that the final database state is consistent and each transaction has seen a consistent database. The implication for concurrency control is the following:

COROLLARY 9.3. *A concurrency control starting with a consistent database state and running valid transactions will maintain database consistency if the overall effect of the granted requests is the same as if the transactions were run serially.*

Proof. Lemma 7.6 tells us that the serial run does produce an acyclic augmented version graph and the theorem applies. □

We note that these results can be applied to concurrency control design without any special knowledge about the consistency criterion. By keeping a record of which transactions supply information read by other transactions and preventing cycles in the augmented version graph, the concurrency control is assured that, starting with a consistent database state, a set of valid transactions will receive and produce a consistent database state.

As an example, consider Fig. 1. First we check if the transactions produce a consistent final database state. The augmented writers version graph is shown in Fig. 2(e). The graph is acyclic and the one node order satisfying Lemma 7.5 is *I*, *b*, *a*. Thus if the initial database state is consistent and transactions *a* and *b* are valid, the final database state is the one obtainable by running first transaction *b* to completion and then transaction *a*. This same order says that transactions *b* and *a* are also matched to consistent databases. The general principle is:

COROLLARY 9.4. *Under the conditions of Theorem* 9.1, *if the augmented writers version graph is acyclic, then each writing transaction is matched to a consistent database state and the final database state is consistent.*

Proof. The augmented individual version graph for each writing transaction is a subgraph of the augmented writers version graph and must also be acyclic. The conclusion is then immediate from the theorem. □

There remains transaction *c* which only read and did not write during its execution. The augmented individual version graph for *c* is shown in Fig. 2(d) and is acyclic. The one order satisfying Lemma 7.5 is *I*, *a*, *c* and transaction *c* is matched to the result of applying transaction *a* to the initial database.

Now look at the augmented version graph itself. It is shown in Fig. 2(c) and does have a cycle, so the graph is not part of a serializable trace. The conditions of Corollaries 9.2 and 9.3 are violated, yet each transaction was given a consistent database state and the final database state was consistent.

The loophole is that read-only transactions need not be checked in determining the consistency of the final database state, and certain "relativity effects" can occur. From the relative viewpoint of transaction *c*, the database appeared to be the result of applying transaction *a* to the initial database state. From the relative viewpoint of the final database state, transaction *a* was run after transaction *b*.

10. The converse result. We are now prepared to address the "if" parts of Theorems 8.1 and 8.2. We actually prove a stronger result. The theorems of § 8 require

that, given entities and a cyclic augmented version graph G, there exist some value set, consistency criterion, initial database state and datatrace for G whereby there is an inconsistency. We show here that, in fact, the values, criterion and initial database can be constructed before G is given. This implies that, even if the consistency criterion is announced in advance and the designer is permitted to tailor the control to the criterion, serializability is still necessary for certain criteria.

We also strengthen the theorems by showing that one construction works for both Theorems 8.1 and 8.2. We thus get the following strengthened converse to Theorem 9.1:

THEOREM 10.1. *Given a set of entities E, there exist*

a) *a set of values V,*

b) *a consistency criterion C for database (E, V) and*

c) *an (initial) database state s in C*

such that for every version graph G, there exists a function σ mapping the transaction nodes of G into valid transaction effects such that (G, σ, s) is a datatrace for (E, V) and

d) *if the augmented writers version graph for G is cyclic, then the final database state for the datatrace is not in C, and*

e) *if the augmented* ivg *for a given transaction node p of G is cyclic, then transaction effect $\sigma(p)$ is not matched to any element in C.*

Proof. Assume a set of entities E is given. We will construct V, C and s.

Construction of values V. Let V be the set of pairs (G, n) where G is a version graph, n is a node of G, and $G = \text{ivg}(G, n)$.

We now define a function TE that maps a value v from V into a transaction effect.

Construction of TE. Let $v = (G, n)$. Define

$$\text{TE}(v) = (\text{READSET}, \text{WRITESET}, r, w)$$

as follows:

$$\begin{aligned} &\text{READSET} = \{e \text{ in } E \mid xR_e n \text{ for some node } x \text{ in } G\},\\ &\text{WRITESET} = \{e \text{ in } E \mid xW_e n \text{ for some node } x \text{ in } G\},\\ &r(e) = (\text{ivg}(G, x), x) \text{ where } xR_e n,\\ &w(e) = v. \end{aligned}$$

For convenience, we write TE (G, n) as an abbreviation for TE $((G, n))$.

Construction of initial database state s_0. Let s_0 be the constant mapping of E into the pair (G_0, I) where G_0 is the version graph consisting of a single node I and no edges.

Construction of consistency criterion C. s is in C if and only if $s = s_0$ or there is a sequence of values

$$(G_1, n_1), \cdots, (G_k, n_k)$$

such that TE $(G_1, n_1), \cdots,$ TE (G_k, n_k) is a serial run on s_0 resulting in s.

LEMMA 10.2. *For all v in V, transaction effect* TE (v) *is valid.*

Proof. If TE (v) is matched to state s which is in C because of sequence of values $v_1, \cdots, v_k$, then the result of transforming s by TE (v) is in C because this state is the result of serial run TE $(v_1), \cdots,$ TE (v_k), TE (v). □

Continuing the proof of Theorem 10.1, we now assume some version graph G is given and construct σ.

Construction of datatrace from version graph G. The datatrace is (G, σ, s_0) where $\sigma(p)$ for transaction node p is TE $(\text{ivg}(G, p), p)$.

It is easily verified that the constructed (G, σ, s_0) satisfies Definition 7.1, the definition of a datatrace.

If the final database state or a state matched to an individual transaction is consistent, then it must be producible as the result of a serial run. This run conceivably could contain effects not part of (G, σ, s_0) and conceivably could contain part of this trace more than once or not at all. The object of the subsequent lemmas is to show that, in all runs, the effects of producer nodes from the constructed datatrace must appear exactly once. Furthermore, their order in the run must agree with edges in the augmented version graph. Therefore the existence of a run will imply no cycles in the graph.

The next lemma says that transaction effects preserve the ivg's of nodes in the values read. Thus the information read is not destroyed by a write operation, and in effect the transactions only append information to an entity.

LEMMA 10.3. *Let database state s_1 be matched to transaction effect $t = \mathrm{TE}(G, n)$ and let s_2 be the result of transforming s_1 by t. For some entity e, let $s_1(e) = (G_1, n_1)$ and $s_2(e) = (G_2, n_2)$. Let p be any node of G_1. Then p is a node of G_2 and* $\mathrm{ivg}(G_1, p) = \mathrm{ivg}(G_2, p)$.

Proof. Case 1. *e is not in* WRITESET *of t.* By Definition 4.2 of "transforms," $s_2(e) = s_1(e)$, and the result is immediate.

Case 2. *e is in* WRITESET *of t.* By Definition 4.2 of "transforms" and construction of TE, $s_2(e) = (G, n)$, so $G_2 = G$ and $n_2 = n$. From the construction of function TE, the value of e read is $(\mathrm{ivg}(G, x), x)$ where xR_en in G. By Definition 4.2 of "matched to," the value of e read equals $s_1(e)$. Hence

$$s_1(e) = (G_1, n_1) = (\mathrm{ivg}(G, x), x).$$

Thus $n_1 = x$ and $G_1 = \mathrm{ivg}(G, x)$. Since $G = G_2$,

$$G_1 = \mathrm{ivg}(G_2, n_1).$$

Since G_1 is a subgraph of G_2, p is a node of G_2 and it remains to be shown that $\mathrm{ivg}(G_1, p) = \mathrm{ivg}(G_2, p)$. Substituting for G_1, this is equivalent to

$$\mathrm{ivg}(\mathrm{ivg}(G_2, n_1), p) = \mathrm{ivg}(G_2, p),$$

which says we must show that p has the same set of ancestors in both $\mathrm{ivg}(G_2, n_1)$ and G_2. Obviously ancestors of p in subgraph $\mathrm{ivg}(G_2, n_1)$ are in G_2, so we must show ancestors of p in G_2 are in $\mathrm{ivg}(G_2, n_1)$.

By hypothesis, p is in G_1 which is $\mathrm{ivg}(G_2, n_1)$, so p is an ancestor of n_1 in G_2. This implies all ancestors of p in G_2 are also ancestors of n_1 and so belong to $\mathrm{ivg}(G_2, n_1)$. □

The next lemma says that a noninitial value read during a serial run must correspond to an earlier transaction in the run.

LEMMA 10.4. *Suppose a consistent database state results from s_0 by a serial run of transaction effects*

$$\mathrm{TE}(G_1, n_1), \cdots, \mathrm{TE}(G_k, n_k).$$

Suppose for some (G_j, n_j) in the run, there is a transaction node x of G_j and an entity e such that xR_en_j in G_j. Then there exists an $i < j$ such that $(G_i, n_i) = (\mathrm{ivg}(G_j, x), x)$.

Proof. Let $s_1, \cdots, s_k$ be the sequence of states specified in Definition 4.4. Since s_{j-1} is matched to $\mathrm{TE}(G_j, n_j)$, the definition of TE implies that $s_{j-1}(e) = (\mathrm{ivg}(G_j, x), x)$. Since $s_{j-1}(e)$ does not equal $s_0(e)$, one of the transactions must have written $s_{j-1}(e)$. The only transaction effect that writes this value is $\mathrm{TE}(\mathrm{ivg}(G_j, x), x)$. □

Next, the preceding lemma is generalized to show that every transaction node in a graph written during a serial run must correspond to an earlier transaction in the serial run.

LEMMA 10.5. *Suppose a consistent database state results from* s_0 *by a serial run of transaction effects*

$$\mathrm{TE}(G_1, n_1), \cdots, \mathrm{TE}(G_k, n_k).$$

Suppose for some (G_j, n_j) *in the run, x is a transaction node of* G_j. *Then there exists an* $i \leqq j$ *such that* $(G_i, n_i) = (\mathrm{ivg}(G_j, x), x)$.

Proof. Because x is in ivg (G_j, n_j), there is a path from x to n_j in G_j. Let the nodes on this path be $x_1 = x, \cdots, x_m = n_j$ such that for all l satisfying $1 \leqq l < m$ there is an entity e such that $x_l R_e x_{l+1}$. The lemma is true if $x = n_j$. Assume the lemma is true for x related to n_j by a path of length m or less, and consider x related to n_j by a path of length $m+1$. Then there is a y such that $xR_e y$ in G_j and a path of length m from y to n_j. From the induction hypothesis, there exists $q \leqq j$ such that $(G_q, n_q) = (\mathrm{ivg}(G_j, y), y)$. Since $xR_e y$ in G_j, we also have $xR_e y$ in G_q. Thus from Lemma 10.4 there exists $i < q$ such that $(G_i, n_i) = (\mathrm{ivg}(G_q, x), x)$. From Lemma 6.6, $\mathrm{ivg}(G_q, x) = \mathrm{ivg}(G_j, x)$. Thus $(G_i, n_i) = (\mathrm{ivg}(G_j, x), x)$. □

COROLLARY 10.6. *If s is a consistent state,* $s(e) = (G, n)$ *for some entity e, and x is a transaction node of G, then every serial run of transaction effects that results in s includes* $\mathrm{TE}(\mathrm{ivg}(G, x), x)$.

Proof. In a serial run that results in s, the last transaction effect whose WRITESET contains e must be $\mathrm{TE}(G, n)$. Apply Lemma 10.5 with $(G_j, n_j) = (G, n)$. □

Corollary 10.6 has established that for each node of the final value of entity e, a transaction effect occurs in the serial run. Next we show that members of the writers chain for an entity e actually occur in their chain order.

LEMMA 10.7. *Let s be a consistent database state resulting from* s_0 *by a serial run of transaction effects, let e be an entity, and let* $\mathrm{TE}(G_1, n_1), \cdots, \mathrm{TE}(G_k, n_k)$ *be the subsequence of the transaction effects which have e in* WRITESET. *Let* $s(e) = (G, n)$.

Then $n_1, \cdots, n_k$ *is the sequence of transaction nodes on the writers chain for e in G, and for* $1 \leqq i \leqq k$

$$G_i = \mathrm{ivg}(G, n_i).$$

Proof. For $1 \leqq j \leqq k$, let P_j be the following predicate:

for $1 \leqq i \leqq j$, n_i is the ith transaction node on the chain for e in G_j and $G_i = \mathrm{ivg}(G_j, n_i)$.

We want to prove P_j by induction on j.

Consider $j = 1$. The only permitted value of i is $i = 1$. Because (G_1, n_1) is in the constructed set of values V, n_1 is in G_1 and $G_1 = \mathrm{ivg}(G_1, n_1)$. Thus to prove P_1, we need only show that n_1 is the first transaction node on the writers chain for e in G_1. We know n_1 is on the writers chain because e is in the WRITESET for $\mathrm{TE}(G_1, n_1)$. Let x be the node such that $xW_e n_1$ (and hence also $xR_e n_1$) in G_j. Since $xR_e n_j$, the value of entity e read is $(\mathrm{ivg}(G_1, x), x)$ by construction of TE. This value equals the value of e in the database state transformed by the transaction effect. Since the transaction effect is the first to write entity e, the value read is the value from the initial database state s_0, namely (G_0, I). Thus $x = I$, and so n_1 is the first transaction node on the chain for e in G_1.

Now assume that P_j is true for some $j < k$, and consider P_{j+1}.

Case 1. $i < j+1$. Transaction effect $\mathrm{TE}(G_{j+1}, n_{j+1})$ reads the value (G_j, n_j) that was written by the preceding writer of entity e, and writes the value (G_{j+1}, n_{j+1}). Since P_j is assumed to be true, n_i is the ith transaction node on the chain for e in G_j and $G_i = \mathrm{ivg}(G_j, n_i)$. From Lemma 10.3, n_i is a node of G_{j+1} and $\mathrm{ivg}(G_j, n_i) = \mathrm{ivg}(G_{j+1}, n_i)$.

From Lemma 6.3, the ith transaction node (namely n_i) on the chain for e in G_j is also the ith transaction node in ivg (G_j, n_i). Thus n_i is the ith transaction node in the identical graph ivg (G_{j+1}, n_i) and (from Lemma 6.3 again) is the ith transaction node in G_{j+1}.

Case 2. $i = j+1$. Because (G_{j+1}, n_{j+1}) is in the constructed set of values V, n_{j+1} is in G_{j+1} and $G_{j+1} = \text{ivg}\,(G_{j+1}, n_{j+1})$. We now show that n_{j+1} is the $(j+1)$st transaction node on the writers chain for e in G_{j+1}. We know n_{j+1} is on the writers chain because e is in the WRITESET for TE (G_{j+1}, n_{j+1}). Let x be the node such that xW_en_{j+1} (and hence also xR_en_{j+1}) in G_{j+1}. Since xR_en_{j+1}, the value of entity e read is (ivg (G_{j+1}, x), x) by construction of TE. This value equals the value of e in the database state transformed by the transaction effect, namely the value (G_j, n_j). Therefore $x = n_j$ and $n_jW_en_{j+1}$ (definition of x). From case 1, letting $i = j$, node n_j is the jth transaction node on the chain in G_{j+1}. Since $n_jW_en_{j+1}$ is in G_{j+1}, node n_{j+1} is the $(j+1)$st transaction node on the chain in G_{j+1}.

The proof of P_j is now completed. To prove the lemma, observe that since the last value written on e is (G_k, n_k), $s(e) = (G_k, n_k)$, and hence $G_k = G$ and $n_k = n$. With these substitutions, P_k implies $G_i = \text{ivg}\,(G, n_i)$.

Finally, Corollary 10.6 implies that for every transaction node x on the writers chain for e in G, the serial run of transaction effects includes TE (ivg (G, x), x). Since every such x has e in WRITESET, the transaction effect appears in the subsequence, and x is one of the n_i. Thus $n_1 \cdots n_k$ is the entire writers chain. □

COROLLARY 10.8. *Let* $\sigma = \text{TE}\,(G, n)$ *be a transaction effect in a serial run on* s_0. *Then if* WRITESET (σ) *is nonull,* TE (G, n) *occurs only once in the serial run.*

Proof. Let e be an entity in WRITESET of σ. Let s be the result of the run. From Lemma 10.7, the subsequence of transaction effects from the serial run which have e in WRITESET are distinguished by their position on the chain for e in $s(e)$. □

We now complete the proof of Theorem 10.1.

Proof of Theorem 10.1(d). Assume the final database s for the constructed datatrace (G, σ, s_0) is consistent and results from the serial run TE $(G_1, n_1), \cdots,$ TE (G_k, n_k) applied to s_0. Let G_w be the writers version graph for G and let x be any transaction node of G_w. Node x writes some entity e (Lemma 6.5(b)) and is a member of the writers chain for e in G_w. Value $s(e)$ is the output of σ(chainend (G, e)) which is constructed to be (ivg $(G$, chainend $(G, e))$, chainend (G, e)). Let $H = \text{ivg}\,(G, \text{chainend}\,(G, e))$. Thus $s(e) = (H, \text{chainend}\,(G, e))$. Since x is on the chain for e in G, x is a transaction node of H. From Corollary 10.6, the serial run of transaction effects includes TE (ivg (H, x), x). From Lemma 6.6, ivg $(H, x) = \text{ivg}\,(G, x)$. Thus the serial run includes TE (ivg (G, x), x), which is the constructed $\sigma(x)$. Since this transaction effect has a nonnull WRITESET, Corollary 10.8 applies and $\sigma(x)$ only occurs once in the serial run.

We now know that each transaction node x of G_w corresponds to a unique transaction effect $\sigma(x)$ in the serial run. Define the *serial order* of the transaction nodes of G_w to be the order in which the corresponding transaction effects occur in the serial run. The serial order extends to all nodes of G_w by putting node I first. We want to show that the edges in aug (G_w) always go from an earlier node in the serial order to a later node. This will imply that aug (G_w) is acyclic.

Suppose xR_ey in G_w. Node y must be a transaction node. Assume x is also a transaction node. The corresponding transaction effects in the run are $\sigma(x)$ and $\sigma(y)$. Event $\sigma(y)$ by construction reads (ivg (ivg (G, y), x), x) from entity e because xR_ey in ivg (G, y). By Lemma 6.6, this value read is (ivg (G, x), x). This value can only have

been written by $\sigma(x)$, so the one occurrence of $\sigma(x)$ in the run must precede the occurrence of $\sigma(y)$. In the case where $x = I$, node x is defined to precede y in the serial order.

Suppose xW_ey. Then also xR_ey (Definition 5.1(d)), and again x precedes y in the serial order.

Suppose xB_ey. Then there is a z such that zR_ex and zW_ey. The value of e produced by z is the value read and overwritten by $\sigma(y)$. This value cannot be the value of entity e after $\sigma(y)$ occurs in the serial run. Since this value is the value read by $\sigma(x)$, $\sigma(x)$ must precede $\sigma(y)$ in the serial run. □

Proof of Theorem 10.1(e). Assume that transaction effect $\sigma(p)$ is matched to a consistent database state, say s. Then s results from a serial run $\text{TE}(G_1, n_1), \cdots, \text{TE}(G_k, n_k)$ applied to s_0. Without loss of generality, it can be assumed that each transaction effect in the serial run has a nonnull WRITESET, since a transaction with a null WRITESET can be deleted, yielding a shorter serial run that also results in state s.

Since s is matched to $\sigma(p)$ we can append $\sigma(p)$ to the above serial run, and obtain a longer serial run,

$$\text{TE}(G_1, n_1), \cdots, \text{TE}(G_k, n_k), \text{TE}(\text{ivg}(G, p), p).$$

The remainder of the proof refers to this longer serial run.

From Lemma 10.5, since $\text{TE}(\text{ivg}(G, p), p)$ is a member of the run, then for all transaction nodes x in $\text{ivg}(G, p)$, the serial run includes $\text{TE}(\text{ivg}(\text{ivg}(G, p), x), x)$. From Lemma 6.6 $\text{ivg}(\text{ivg}(G, p), x) = \text{ivg}(G, x)$, so the serial run includes $\text{TE}(\text{ivg}(G, x), x)$ which is the constructed $\sigma(x)$.

From Lemma 6.5(a), every node in $\text{ivg}(G, p)$, except possibly p, is a producer node and, from Corollary 10.8, occurs only once in the serial run. If p is not a producer node, then $\sigma(p)$ has a null WRITESET, and so $\sigma(p)$ occurs only once in the serial run (since the original serial run resulting in s was assumed to have no read-only transactions).

We now know that each transaction node x of $\text{ivg}(G, p)$ corresponds to a unique transaction effect $\sigma(x)$ in the serial run. As in the proof of Theorem 10.1(d), we can define the *serial order* of these nodes, and extend the order to all nodes of $\text{ivg}(G, p)$ by putting node I first. In a manner similar to that in the proof of Theorem 10.1(d), it can be shown that the edges of $\text{aug}(\text{ivg}(G, p))$ always go from an earlier node in the serial order to a later node. This implies that $\text{aug}(\text{ivg}(G, p))$ is acyclic. □

Discussion of proof techniques. The version graphs in the proof can be thought of as "generalized Herbrand values." Each entity value incorporates the total historic record of the flow of information used to create the value. The usual Herbrand technique [12] involving strings as values does not work here because the strings can only be defined if the values are developed in a sequential manner (each string includes as substrings the relevant previously computed values). However, our model does not have sequential evaluations and, when a version graph (not augmented) is cyclic, there is no sequence of operations which represents the flow.

Theorem 10.1 shows the necessity of serializability for both the consistency of the final state (part (d)) and the consistency of the view seen by individual transactions (part (e)). The complexity of the proof is due to part (e). If Theorem 10.1 were stated without part (e), a much simpler proof would suffice. Instead of writing version graphs, the transactions would need to write only enough information so that the edges of the version graph could be deduced from the final database state. The consistency criterion

would be that the augmented deduced graph be acyclic. For example, a transaction could append to entities written a set of pairs each of which is the name of an entity read and the name of the producer node that created the value read.

11. The read-before-write assumption. As discussed in § 4, Definition 4.1(b) embodies Assumption A6 that a transaction reads an entity before it writes. Here we want to consider the possibility that a transaction could instruct the concurrency control to write an entity without having read the entity. Will Theorem 10.1 generalize to such situations? We show here by example that the answer is "no."

Consider three transactions P, Q, and R which are run concurrently and access the database with the following schedule of events:

P Writes α (and terminates)
Q Reads α
R Reads α
Q Writes β
Q Writes α (and terminates)
R Reads β
R Writes γ (and terminates).

The "generalized augmented version graph" showing the information flow is shown in Fig. 3. It is no longer appropriate to show W_e edges because writes may not replace a specified value. Instead the entities written are indicated beside each transaction node. The augmented edge from R to Q was added because the value of α read by R was read and then overwritten by Q.

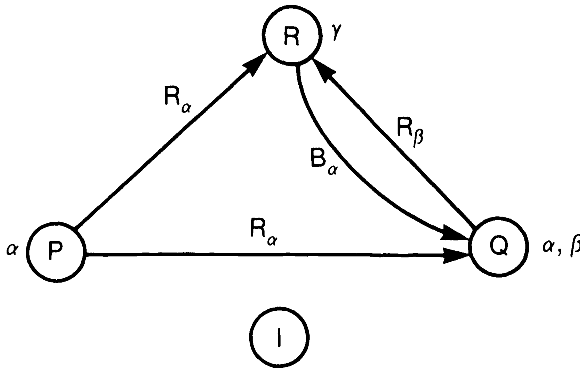


FIG. 3

As indicated by the cycle in the graph, the schedule of events is not serializable. Entity β was written by Q and then read by R so Q must be scheduled before R. On the other hand, the value of α written by P was read by R before being overwritten by Q so R must be scheduled before Q.

If Theorem 10.1 were generalized, an interpretation would exist such that R was not matched to any consistent database state and the final database state was not consistent. However, this cannot be true since R sees the database state obtained by running the sequence

$$P, Q, P$$

and the final database is that obtained by running the sequence

$$P, Q, P, R.$$

Remark. In going from a schedule to a "generalized version graph," information about the final database state can be lost as the chainend concept depends on the read-before-write assumption, and the information flow may not show which value was written last. Thus the graph may be inadequate for further study of the write-without-read case.

Remark. It should be remembered that the counterexample is for the consistency concept given in Definition 3.2. It is possible that some intuitive but stronger criteria can be found which do imply serialization. Producing the example result of P, Q, P, R is intuitively unsatisfactory when transaction P has in fact been initiated only once by a system user.

Remark. Another example is provided by the following schedule, where all reads of a transaction occur in a single combined access, and all writes of a transaction occur in a single combined access:

S Writes α (and terminates)
R Reads α
P Writes α and β (and terminates)
Q Reads β and γ
Q Writes δ (and terminates)
R Writes γ (and terminates).

The final database state is consistent because it can be obtained by running the sequence P, Q, S, R, P. Any serial schedule producing the final database state must repeat a transaction.

12. Time assumptions. By looking at datatraces in terms of information flow instead of schedules, certain timing information is lost. For example, the version graph makes no distinction between the following two schedules:

P READS α	P READS α
P READS β	P WRITES α
P WRITES α	P READS β

The second schedule gives the illusion that the value written into entity α is unrelated to the value read from β, since α was written before the value of β was known. However, it must be remembered that the schedule is not a program, and the transaction might have produced a different schedule had the data read from the database been different.

To be more specific, suppose P read the value 1 from α, wrote the value 2 into α, then read the value 2 from β. Is it a coincidence that the transaction leaves α and β with the same value? It would be if the program for the transaction were the following:

$I \leftarrow$ ENTITY (α)
ENTITY $(\alpha) \leftarrow I+1$
$J \leftarrow$ ENTITY (β).

If, however, the program were

$I \leftarrow$ ENTITY (α)
ENTITY $(\alpha) \leftarrow I+1$
$J \leftarrow$ ENTITY (β)
If $J \neq I+1$ THEN ENTITY $(\beta) \leftarrow I+1$,

then the transaction would always produce a database state with ENTITY $(\alpha) =$ ENTITY (β). The two programs often produce different results, but they have the same effect when presented with a database state with value 1 for α and 2 for β.

The point is that the values read and the values written could have any relationship, regardless of the order in which the reads and writes are performed in the particular running of the transaction. Thus it is reasonable to work with a model (the datatrace in our case) where the ordering is not considered.

Some authors have worked with models in which transactions are taken to be straight-line programs accessing a fixed sequence of entities. In this case, the order of access operations influences consistency.

For example, consider the following sequences of actions by two consistency-preserving straight-line programs P and Q:

P reads α	Q reads β
P writes α	Q writes β
P reads β	Q reads α

These programs can be run concurrently as follows so that the information flow is that shown in Fig. 4(a):

P reads α
Q reads β
P writes α
Q writes β
P reads β
Q reads α

The version graph and its augmented graph are cyclic. The run is not serializable: P must precede Q because P reads the value of α written by Q, and Q must precede P because of β.

However, if programs P and Q are modified to end after the final write and to omit subsequent reads, the programs produce the same effect on the database as the original. The modified programs therefore also preserve consistency when run alone. Concurrent running of the modified programs produces:

P reads α
Q reads β
P writes α
Q writes β

which gives the information flow shown in the version graph of Fig. 4(b) (which is Fig. 4(a) with the two read operations deleted). Augmented 4(b) is acyclic (hence serializable) and produces a consistent final database state. But the two sequences of operations produce the same database state, and the final database state is the same for both 4(a) and 4(b). The final database state in 4(a) is consistent in spite of not being serializable. Thus the "if" part of Theorem 8.1 fails for straight-line programs.

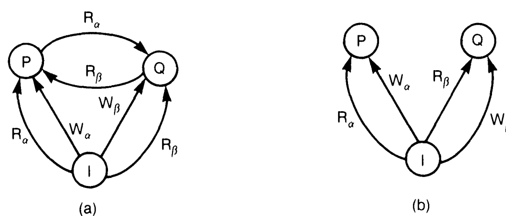


FIG. 4

Understanding consistency for the straight-line case involves the concept of a "trailing read." We refer to READ operations after the last WRITE operation performed by a process as *trailing* READs. These reads are the ones that do not affect the values written and can be deleted from the straight-line transactions without affecting their validity.

The following can be proven for the straight-line case:

Modify the version graph by deleting edges corresponding to trailing READs. If the transactions are valid, if the original database is consistent, and if the augmented version graph is acyclic, then the final database state is consistent. If the augmented version graph is cyclic, there exist a consistency criterion, a set of valid straight-line transactions, and an initial consistent database state such that the given history transforms the initial state into an inconsistent state.

Thus the straight-line case provides a loophole for trailing reads.

The problem of describing the consistency of data read is more complex. Consider the following three straight-line programs P, Q, and R:

P reads α	Q reads β	R reads α
P reads γ	Q writes β	R reads β
P writes α	Q reads γ	
P reads β	Q writes γ	
P reads δ		
P writes δ		

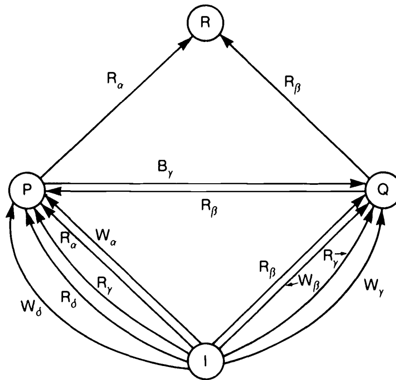


FIG. 5

These programs can be run concurrently as follows, so that the information flow is that shown in the augmented version graph of Fig. 5:

P reads α
P reads γ
P writes α
Q reads β
Q writes β
P reads β
R reads α
R reads β
Q reads γ
Q writes γ
P reads δ
P writes δ

In spite of the cycle in the ivg of transaction R, R does see consistent data as demonstrated by running the transactions in order P, Q, R. This serial schedule produces a different version graph from Fig. 5, but R gets the same information in both cases. This example shows that the "if" part of Theorem 8.2 fails for straight-line programs.

The information from Fig. 5 relevant to the consistency of data read by transaction R is shown in Fig. 6. A number of edges have been deleted because they did not impact on the data seen by R. For example, the edge from Q to P labelled R_β was deleted because the value read does not affect any entity value read by R. A new kind of edge labelled γ between P and Q was added because transaction R read a value of α reflecting the fact that P read the original version of γ. Since the straight-line program for Q requires that Q write γ, from R's viewpoint P must precede Q. The serialization P, Q, R is a topological sort of Fig. 6.

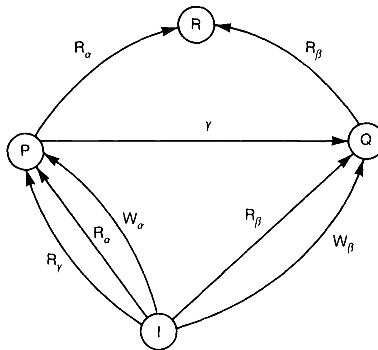


FIG. 6

13. Conclusion. We believe the model used above is the correct one for general purpose concurrency controls. "General purpose" means the control must maintain consistency no matter what the criterion happens to be and regardless of the structure (straight-line or otherwise) used to program the transactions. The only imposition on the user is that he or she start with consistent data and only run transactions which preserve consistency if run alone. For consistency, the concurrency control need remember no information other than the version graph. The control must operate to keep the augmented version graph acyclic, for *any* cyclic augmented (writers) version graph is associated with an instance of inconsistency.

There is also danger in allowing a temporary cyclic flow of information, even when the control plans to break the cycle later with a rollback. The danger is that *any* cyclic augmented individual version graph is associated with an inconsistency and the individual transaction may be processing garbage.

There is a loophole for read-only transactions, although it is probably not worth exploiting in practice. Also, such exploitation may be considered unsatisfactory if one takes a stronger view of consistency than our admittedly weak mathematical definition. The definition is good for inferring things that should not be allowed (e.g. cyclic information flow), but additional criteria may be appropriate in deciding what should be allowed.

The general conclusion is that general purpose concurrency controls should be designed so that the augmented version graph is acyclic. Except for the minor read-only loophole, this is the only way to maintain consistency.

REFERENCES

[1] R. BAYER, E. ELHARDT, H. HELLER AND A. REISER, *Distributed concurrency control in database systems*, in Proc. Sixth International Conference on Very Large Data Bases, Montreal, Oct. 1980, pp. 275–284.

[2] R. BAYER, H. HELLER AND A. REISER, *Parallelism and recovery in database systems*, ACM Trans. Database Systems, 5 (1980), pp. 139–156.

[3] M. A. CASANOVA, *The Concurrency Control Problem for Database Systems*, Lecture Notes in Computer Science, 116, Springer-Verlag, Berlin, 1981.

[4] M. A. CASANOVA AND P. A. BERNSTEIN, *General purpose schedules for database systems*, Acta Inform., 14 (1980), pp. 195–220.

[5] K. P. ESWARAN, J. N. GRAY, R. A. LORIE AND I. L. TRAIGER, *The notions of consistency and predicate locks in database systems*, Comm. ACM, 19 (1976), pp. 624–633.

[6] G. GARDARIN, *Contributions to the theory of concurrency in databases*, in Mathematical Foundations of Computer Science 1978, J. Winkowski ed., Lecture Notes in Computer Science, 64, Springer-Verlag, Berlin, 1978, pp. 201–212.

[7] G. GARDARIN AND P. LEBEUX, *Scheduling algorithms for avoiding inconsistency in large databases*, in Proc. Third International Conference on Very Large Data Bases, Tokyo, Oct. 1977, pp. 501–506.

[8] G. GARDARIN AND M. MELKANOFF, *Proving consistency of database transactions*, in Proc. Fifth International Conference on Very Large Data Bases, Rio de Janeiro, Oct. 1979, pp. 291–298.

[9] J. N. GRAY, R. A. LORIE, G. R. PUTZOLU AND I. L. TRAIGER, *Granularity of locks and degrees of consistency in a shared data base*, in Modelling in Data Base Management Systems, G. M. Nijssen, ed., North-Holland, Amsterdam, 1976, pp. 365–394.

[10] H. T. KUNG AND C. H. PAPADIMITRIOU, *An optimality theory of concurrency control for databases*, in Proc. SIGMOD International Conference on Management of Data, Boston, May 1979, pp. 116–126.

[11] L. LAMPORT, *Towards a theory of correctness for multi-user data base systems*, Rept. CA-7610-0712, Mass. Comput. Assoc. Inc., Oct. 1976.

[12] Z. MANNA, *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

[13] D. P. REED, *Naming and synchronization in a decentralized computer system*, Ph.D. thesis, MIT/LCS/TR-205, Dept. of Electrical Engineering and Computer Science, Massachusetts Inst. of Technology, Cambridge, MA, Sept. 1978.

[14] R. E. STEARNS, P. M. LEWIS II AND D. J. ROSENKRANTZ, *Concurrency controls for database systems*, in Proc. Seventeenth Annual Symposium on Foundations of Computer Science, Houston, TX, Oct. 1976, pp. 19–32.

[15] R. E. STEARNS AND D. J. ROSENKRANTZ, *Distributed database concurrency controls using before values*, in Proc. SIGMOD International Conference on Management of Data, Ann Arbor, MI, April 1981, pp. 74–83.

SIAM J. COMPUT.
Vol. 13, No. 3, August 1984
© 1984 Society for Industrial and Applied Mathematics
005

THE ORGAN PIPE PERMUTATION*

M. KEANE,† A. G. KONHEIM,‡ AND I. MEILIJSON§

Abstract. Let $(p_1, p_2, \cdots, p_n)$ be a probability distribution, $\pi = (\pi_1, \pi_2, \cdots, \pi_n)$ be a permutation of $1, 2, \cdots, n$ and $X_1, X_2, \cdots, X_k$ be k independent and identically distributed random variables with distribution $P(X = i) = p\pi_i$. It is known that the *organ pipe permutation* π^* makes the range

$$D(k, \pi) = \max_{1 \leqq j \leqq k} X_j - \min_{1 \leqq j \leqq k} X_j$$

a stochastic minimum for $k = 2$ (P. P. Bergmans, Information and Control, 20 (1972), pp. 331–350), and minimal on the average for general k (J. R. Bitner and C. K. Wong, 8 (1979), pp. 479–498). We prove the stochastic minimality for general k and study a natural extension of the organ pipe permutation that is optimal when certain constraints are placed on the possible choices of π

Key words. file storage, disk, optimization

1. Introduction. Let $x = (x_1, x_2, \cdots, x_n)$ be a vector of positive real numbers and let $s = \sum_1^n x_i$. For each permutation π of $N = (1, 2, \cdots, n)$, let x^π denote the vector with $x_i^\pi = x_{\pi i}$, $1 \leqq i \leqq n$.

The present note deals with the following two subjects:

1. For $s = 1$, let F^π be the distribution of a random variable X^π defined by

$$P(X^\pi = i) = x_i^\pi, \qquad 1 \leqq i \leqq n. \tag{1}$$

For a random sample $X_1^\pi, X_2^\pi, \cdots, X_k^\pi$ from F^π, let $D(k, \pi)$ be its *range*, i.e.,

$$D(k, \pi) = \max_{1 \leqq i \leqq k} X_i^\pi - \min_{1 \leqq i \leqq k} X_i^\pi. \tag{2}$$

A permutation π is an *organ pipe* permutation if

$$x_1^\pi \leqq x_n^\pi \leqq x_2^\pi \leqq x_{n-1}^\pi \leqq x_3^\pi \leqq \cdots \tag{3}$$

or if

$$x_n^\pi \leqq x_1^\pi \leqq x_{n-1}^\pi \leqq x_2^\pi \leqq x_{n-2}^\pi \leqq x_3^\pi \leqq \cdots. \tag{4}$$

Bergmans [1] has shown that the organ pipe permutations minimize $D(2, \pi)$ stochastically, i.e., for every permutation π, every organ pipe permutation π^* and every nondecreasing function $\phi: R \to R$,

$$E(\phi(D(2, \pi^*))) \leqq E(\phi(D(2, \pi))). \tag{5}$$

We will prove:

THEOREM 1. *The organ pipe permutations minimize $D(k, \pi)$ stochastically for every $k \geqq 2$. If π is not an organ pipe permutation and ϕ is strictly increasing on $\{1, 2, \cdots, n-1\}$, then the inequality in* (5) (*with k replacing* 2) *is strict.*

2. Let f be a symmetric concave function of one real variable (with a vertical axis of symmetry somewhere in the plane), $x = (x_1, x_2, \cdots, x_n)$ a vector of positive real numbers, $s = x_1 + x_2 + \cdots + x_n$ and α a real number. Let $\pi = (\pi_1, \cdots, \pi_n)$ be a permutation of the vector $(1, 2, \cdots, n)$. (π_i is used to denote that coordinate of $(1, 2, \cdots, n)$ which is mapped into the ith coordinate of π.) For each permutation π

* Received by the editors October 23, 1981, and in final revised form May 12, 1983.
† Technische Hogeschool Delft, The Netherlands.
‡ Department of Computer Science, University of California, Santa Barbara, California 93106.
§ Department of Statistics, Tel-Aviv University, Ramat Aviv, Israel.

of $(1, 2, \cdots, n)$ let

$$V(f, \pi, \alpha, x) = V(\pi) = f(\alpha) + \sum_{i=1}^{n} f\left(\alpha + \sum_{j=1}^{i} x_{\pi_j}\right). \tag{6}$$

Suppose that $x_1 \leqq x_2 \leqq \cdots \leqq x_n$. Define a permutation π to be greedy (on the triple (f, x, α)) recursively as follows:

If $f(\alpha) < f(\alpha + s)$ set $\pi_1 = 1$.
If $f(\alpha) = f(\alpha + s)$ set $\pi_1 = 1$ or $\pi_n = 1$.
If $f(\alpha) > f(\alpha + s)$ set $\pi_n = 1$.

Having defined either the first or the last coordinate of π (i.e., π_1 or π_n) consider the reduced vector $x' = (x_2, x_3, \cdots, x_n)$ and the modified value of α,

$$\alpha' = \begin{cases} \alpha + x_1 & \text{if } \pi_1 = 1, \\ \alpha & \text{if } \pi_n = 1, \end{cases}$$

and apply the rule specifying π_1 or π_n, to the reduced triple (f, α', x'). The remaining values of π are determined recursively by this procedure.

To be more specific, suppose that $f(\alpha) = f(\alpha + s)$. Then the following are greedy permutations (since f is symmetric about $\alpha + s/2$):

$$\pi^* = \begin{cases} (1, 3, 5, \cdots, n-1, n, n-2, \cdots, 6, 4, 2) & \\ \quad \text{or } (2, 4, 6, \cdots, n-2, n, n-1, \cdots, 5, 3, 1) & \text{if } n \text{ is even}, \\ (1, 3, 5, \cdots, n-2, n, n-1, \cdots, 6, 4, 2) & \\ \quad \text{or } (2, 4, 6, \cdots, n-1, n, n-2, \cdots, 5, 3, 1) & \text{if } n \text{ is odd}. \end{cases}$$

Clearly, $V(\pi)$ is the same for all greedy permutations.

THEOREM 2. *The function $V(\pi)$, given by* (6), *defined on the set of all permutations, attains its minimum at the greedy permutations.*

For each of exposition, consider f to describe the roof of a house (see Fig. 1) with one-dimensional floor and rooms having preassigned lengths $\{x_i\}$. Theorem 2 claims that the greedy builder, who minimizes at every stage of the construction the height of the current wall being built, makes the sum of the heights of the walls actually minimal.

It should be observed that the optimality of the greedy permutations does not necessarily hold for concave, *non*symmetric functions. We will provide a counter-example in which f is the minimum of two straight lines at different absolute slopes.

The connection between subjects 1 and 2 is that a weaker form of Theorem 1, the minimization by the organ pipe permutations of the *expectation* of $D(k, \pi)$, is an immediate corollary of Theorem 2, because when $f(\alpha) = f(\alpha + s)$, π is greedy if and only if it is in an organ pipe permutation, and for $s = 1$ and $f(t) = -(t^k + (1-t)^k)$,

$$E(D(k, \pi)) = n + V(f, \pi, 0, x). \tag{7}$$

The minimization of $E(D(k, \pi))$ by the organ pipe permutation has been proved recently by Bitner and Wong [2]. Related work was done by Wong [5]. The motivation for this question is the need for specifying the order in which data should be recorded on a disk. A disk is a storage device in a computing system consisting of several surfaces like phonograph records sharing a common spindle. Data is recorded magnetically on the concentric tracks of these surfaces. A storage location on the disk is specified by the triple (r, θ, z) where z specifies the surface, r the track on the surface and θ the angular position on the track. A *cylinder* on a disk is the set of all locations

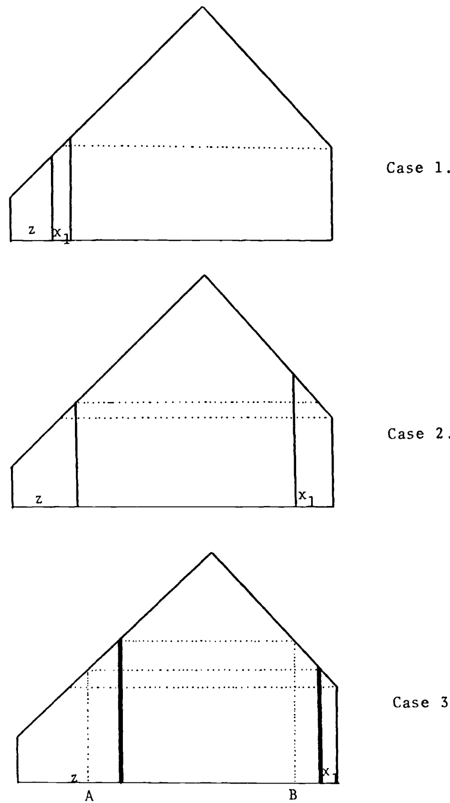


FIG. 1

(r, θ, z) with r fixed. The time to access information stored on the arc $\{(r, \theta, z): \theta_1 \leqq \theta < \theta_2\}$ is composed of the *seek latency*—the time to move the reading head from r' to r, and the *rotational latency*—the time for the surface to rotate to the position (r, θ_1, z). Now suppose that information on a disk is organized into units each requiring one cylinder. A sequence of k requests

$$X_1, X_2, \cdots, X_k \rightarrow \text{read cylinder } X_1, \text{ read cylinder } X_2, \cdots, \text{ read cylinder } X_k$$

will be processed by moving the reading head to one of the extremes $\text{MIN}_i X_i$ or $\text{MAX}_i X_i$ (the *initial movement*) and thereafter accessing the information in the order of the cylinders encountered (the *scanning period*). To minimize the access time we wish to distribute the information on the n cylinders so as to minimize some expression related to the range $\text{MAX}_i X_i - \text{MIN}_i X_i$, assuming that $X_i = j$ with probability $p_{\pi j}$.

Bitner and Wong [2] consider three simple rules for moving the reading head. The *leftist* rule (move to $\text{MIN}_i X_i$, read through $\text{MAX}_i X_i$, leave the head there), the *rightist* rule (the other way around) and the *alternating* rule (if the last batch was read from MIN to MAX, the present one is read MAX to MIN and vice versa). They prove the organ pipe to minimize the total expected traveling time of the reading head under each one of these rules, besides minimizing the expected range of the batch.

We will show all of these optimal properties of the organ pipe permutations to be immediate consequences of Theorem 2, by exhibiting in each case the corresponding roof function f. These optimal properties remain true if the reading head moves at higher speed during the initial movement than during the scanning period, or even if these speeds are different for outward or inward movements.

A novel contribution provided by Theorem 2, beyond the optimality of the organ pipe permutation, is the following. Suppose that some of the data must be recorded at the inmost cylinders and some at the outmost cylinders. The best permutation of the central part need no longer be the organ pipe nor need it be the same for all abovementioned rules. The sum of the probabilities of the inmost cylinders takes the role of α (see (6)), the probability of the central section is s, and Theorem 2 implies that the best permutation is built greedily on the corresponding f.

In an entirely different direction, Meilijson and Tamir [4] have applied Theorem 2 to a scheduling problem on many processors. The total flow time on parallel identical processors with nonincreasing unit processing time is minimized by shortest processing time scheduling: start with the shortest tasks and assign the shortest unassigned task to the first processor that becomes available.

Theorem 1 claims the stochastic minimization rather than the (weaker) average minimization of the range by the organ pipe. There is an important distinction between the two. It may happen that not only the long run average head moving time is relevant, but also, for some timing requirements involving, say, parallel computation, all reading ought to be performed within a preassigned time span. Whatever this span is, Theorem 1 states that the organ pipe will maximize the probability of meeting it. More generally, if the penalty for memory access time is the head moving time under any of the abovementioned rules *plus* any nondecreasing function of the scanning time, then both theorems are needed to infer that the organ pipe permutations are optimal.

2. Proof of Theorem 1.

DEFINITION 1. The *j-flip* $T_j x$ of a vector x, for $2 \leqq j \leqq n+1$, is the vector with

$$(T_j x)_i = \begin{cases} \min(x_i, x_{j-1}) & \text{if } i \leqq j/2, \\ \max(x_i, x_{j-1}) & \text{if } j/2 < i < j, \\ x_i & \text{if } i \geqq j. \end{cases} \tag{8}$$

The 2-flip of x is, of course, x itself. The flip $T_1 x$ of x is the vector with $(T_1 x)_i = x_{n+1-i}$, for all $1 \leqq i \leqq n$.

LEMMA 1. *For every permutation π of N and every organ pipe permutation π^*, there exists a finite sequence $j_1, j_2, \cdots, j_m$ with $1 \leqq j_i \leqq n+1$, such that $T_{j_m} \cdots T_{j_2} T_{j_1} x^\pi = x^{\pi^*}$.*

Proof. Let x denote x^{π^*}. We may assume that x satisfies (3), as, had it satisfied (4), $T_1 x$ would have satisfied (3).

Let

$$j_1 = 1 + \min\{i \mid x^\pi = x_1\}, \qquad j_2 = 1 + \min\{i \mid (T_1 T_{j_1} x^\pi)_i = x_n\}. \tag{9}$$

The vector $T_1 T_{j_2} T_1 T_{j_1} x^\pi$ agrees with x on the first and last coordinates.

To see the inductive step, assume that we have a vector that agrees with x on the first m and on the last m coordinates, for some m with $2m \leqq n-2$. Remove those $2m$ coordinates and perform the construction (j_1, j_2) described above on the $(n-2m)$-dimensional vector that is left, thus bringing x_{2m+1} to its first coordinate and x_{2m+2} to its last. Whenever a j-flip is performed in this construction, its effect on the $(n-2m)$-dimensional vector is identical to the effect on those $(n-2m)$ coordinates of the application of the $(j+2m)$-flip to the full n-dimensional vector. As is easy to infer from (3), this $(j+2m)$-flip leaves the first and last m coordinates unchanged. The inductive step is complete. □

The random variable X is said to be *stochastically larger* than the random variable Y if for all real x, $P(X>x) \geqq P(Y>x)$. It is well known (see, e.g., Lehmann [3, Lemma 1, p. 73]) that X is stochastically larger than Y if and only if, for some joint distribution of (X, Y) with the given marginals, $P(X \geqq Y) = 1$. From this it follows that X is stochastically larger than Y if and only if $E\phi(X) \geqq E\phi(Y)$, for every nondecreasing function ϕ for which $E\phi(X)$, $E\phi(Y)$ exist (use $X \geqq Y \Rightarrow \phi(X) \geqq \phi(Y)$ for one direction and the functions $\phi_x(y) = 0$ if $y \leqq x$, $\phi_x(y) = 1$ if $y > x$ for the other).

We thus have to show that for all integers $d \geqq 1$ and $k \geqq 2$, any permutation π and any organ pipe permutation π^*,

$$P(D(k, \pi) \geqq d) \leqq P(D(k, \pi^*) \leqq d). \tag{10}$$

In view of Lemma 1, it is enough to prove (10) with π^* replaced by the application of T_j to x^π, for arbitrary $j \geqq 3$ (as T_1 and T_2 obviously yield equality of the distributions of the two D's in question). This will be proved in Lemma 3.

LEMMA 2. *Let π be the identity permutation and omit it as a superscript. Then, for all integers $d \in [1, n-2]$,*

$$P(D(k) \leqq d) = \sum_{l=1}^{n-d} \left(\sum_{i=l}^{l+d} x_i \right)^k - \sum_{l=2}^{n-d} \left(\sum_{i=l}^{l+d-1} x_i \right)^k. \tag{11}$$

Proof.

$$\begin{aligned} P(D(k) \leqq d) &= \sum_{m=1}^{n} P\left(D(k) \leqq d, \max_{1 \leqq i \leqq k} x_i = m \right) \\ &= \sum_{m=1}^{n} \{P(\{X_1, X_2, \cdots, X_k\} \subseteq [m-d, m]) \\ &\qquad - P(\{X_1, X_2, \cdots, X_k\} \subseteq [m-d, m-1])\} \\ &= \sum_{m=1}^{n} \left\{ \left(\sum_{i=\max(1, m-d)}^{m} x_i \right)^k - \left(\sum_{i=\max(1, m-d)}^{m-1} x_i \right)^k \right\} \\ &= \sum_{l=1}^{n-d} \left(\sum_{i=l}^{l+d} x_i \right)^k - \sum_{l=2}^{n-d} \left(\sum_{i=l}^{l+d-1} x_i \right)^k. \end{aligned} \tag{12}$$

□

If $D(k)$ is the range of a random sample of size k with distribution given by the probability vector x (see (2)) and j is an integer with $j \geqq 3$, then let $D^j(k)$ be the corresponding range for the probability vector $T_j x$ and define

$$\Delta(x, j, d, k) = P(D^j(k) \leqq d) - P(D(k) \leqq d). \tag{13}$$

LEMMA 3. *For all probability vectors x and all integers $k \geqq 2$, $j \in [3, n+1]$, $d \in [1, n-2]$,*

$$\Delta(x, j, d, k) \geqq 0. \tag{14}$$

Proof. The proof will be divided into three cases:

(i) $j - d \leqq 2$;

(ii) $j - d$ is even and exceeds 2;

(iii) $j - d$ is odd and exceeds 2.

Case (i).

$$\begin{aligned} \Delta(x, j, d, k) &= \left[\left(\sum_{i=1}^{1+d} (T_j x)_i \right)^k - \left(\sum_{i=1}^{1+d} x_i \right)^k \right] \\ &\quad + \sum_{l=2}^{n-d} \left[\left(\sum_{i=l}^{l+d} (T_j x)_i \right)^k - \left(\sum_{i=l}^{l+d-1} (T_j x)_i \right)^k - \left(\sum_{i=l}^{l+d} x_i \right)^k + \left(\sum_{i=l}^{l+d-1} x_i \right)^k \right]. \end{aligned} \tag{15}$$

The first term on the right-hand side of (15) is zero, because (i) implies that

$$\sum_{i=1}^{1+d} (T_jx)_i = \sum_{i=1}^{1+d} x_i.$$

We will now show that each summand in the second term is nonnegative. Fix $l \in [2, n-d]$.

Let α denote $\sum_{i=l}^{l+d-1} (T_jx)_i$ and β denote $\sum_{i=l}^{l+d-1} x_i$.

By assumption (i), $(T_jx)_{l+d} = x_{l+d}$. Let γ denote this value.

By definition of T_j, $\alpha \geqq \beta$.

Because the function $f(t) = t^k$ is convex, the summand in question, which is $((\alpha+\gamma)^k - \alpha^k) - ((\beta+\gamma)^k - \beta^k)$, is nonnegative.

Case (ii). Every set of indices which is symmetric with respect to $j/2$ is mapped onto itself by T_j; in particular this holds for the set $\{l^*, l^*+1, \cdots, l^*+d\}$, where $l^* = (j-d)/2$.

Hence, $\sum_{i=l^*}^{l^*+d} (T_jx)_i = \sum_{i=l^*}^{l^*+d} x_i$. Set $L = j-d-1$. By eliminating a zero and shifting the location of some of the negative sums in (11), Δ may be expressed as

$$\begin{aligned} \Delta(x, j, d, k) = &\sum_{l=1}^{l^*-1} \left[\left(\sum_{i=l}^{l+d} (T_jx)_i \right)^k - \left(\sum_{i=l+1}^{l+d} (T_jx)_i \right)^k - \left(\sum_{i=l}^{l+d} x_i \right)^k + \left(\sum_{i=l+1}^{l+d} x_i \right)^k \right] \\ &+ \sum_{l=l^*+1}^{L} \left[\left(\sum_{i=l}^{l+d} (T_jx)_i \right)^k - \left(\sum_{i=l}^{l+d-1} (T_jx)_i \right)^k - \left(\sum_{i=l}^{l+d} x_i \right)^k + \left(\sum_{i=l}^{l+d-1} x_i \right)^k \right] \\ &+ \sum_{l=L+1}^{n-d} \left[\left(\sum_{i=l}^{l+d} (T_jx)_i \right)^k - \left(\sum_{i=l}^{l+d-1} (T_jx)_i \right)^k - \left(\sum_{i=l}^{l+d} x_i \right)^k + \left(\sum_{i=l}^{l+d-1} x_i \right)^k \right]. \end{aligned} \tag{16}$$

By the proof of Case (i), it is seen that each summand in the third term of the right-hand side of (16) is nonnegative. (Remark: So is each one in the second term, but not in the first. The second will now be seen to compensate for the first.)

The sum of the first two terms on the right-hand side of (16) may be expressed as

$$\begin{aligned} S = \sum_{l=1}^{l^*-1} \Bigg[&\left(\sum_{i=l}^{l+d} (T_jx)_i \right)^k - \left(\sum_{i=l+1}^{l+d} (T_jx)_i \right)^k - \left(\sum_{i=l}^{l+d} x_i \right)^k + \left(\sum_{i=l+1}^{l+d} x_i \right)^k \\ &+ \left(\sum_{i=j-l-d}^{j-l} (T_jx)_i \right)^k - \left(\sum_{i=j-l-d}^{j-l-1} (T_jx)_i \right)^k - \left(\sum_{i=j-l-d}^{j-l} x_i \right)^k + \left(\sum_{i=j-l-d}^{j-l-1} x_i \right)^k \Bigg]. \end{aligned} \tag{17}$$

We claim that each summand (in square brackets) in (17) is nonnegative. Fix $l \in [1, l^*-1]$. Let S_l denote the corresponding summand. Let $\alpha, \beta, \gamma, \delta, A, B, C, D$ be defined by the following relations:

$$\begin{aligned} \alpha &= \sum_{i=l}^{l+d} (T_jx)_i = (T_jx)_l + \beta, \qquad & A &= \sum_{i=j-l-d}^{j-l} (T_jx)_i = B + (T_jx)_{j-l}, \\ \gamma &= \sum_{i=l}^{l+d} x_i = x_l + \delta, & C &= \sum_{i=j-l-d}^{j-l} x_i = D + x_{j-l}. \end{aligned} \tag{18}$$

Then

$$S_l = \alpha^k - \beta^k - \gamma^k + \delta^k + A^k - B^k - C^k + D^k. \tag{19}$$

We will show that $S_l \geqq 0$ for each of two cases $x_l = (T_jx)_l$ and $x_l = (T_jx)_{j-l}$.

First case. $x_l = (T_jx)_l$.

Set

$$h = (T_jx)_{j-l} - x_l \geqq 0, \quad B' = A - x_l = B + h, \quad D' = C - x_l = D + h, \tag{20}$$

and let S'_l be the expression S_l with B replaced by B' and D replaced by D'. $S_l - S_{l'}$ and S'_l will be shown to be nonnegative.

$$S_l - S_{l'} = (B+h)^k - B^k - (D+h)^k + D^k \tag{21}$$

is nonnegative because $B \geqq D$ (by the definition of T_j) and $f(t) = t^k$ is convex. If a function f has a convex derivative, then for fixed $u \geqq 0$ and $v \geqq 0$, the function

$$g(t) = f(t+u+v) - f(t+u) - f(t+v) + f(t) \tag{22}$$

is nondecreasing. Set $f(t) = t^k$, $u = x_l$ and

$$v = \delta - \beta = \sum_{i=l}^{l+d} (x_i - (T_jx)_i) = \sum_{i=l}^{j-l-d-1} (x_i - (T_jx)_i) = \sum_{i=l}^{\min(l+d, j-l-d-1)} (x_i - (T_jx)_i). \tag{23}$$

Then $v \geqq 0$ because every summand in the last sum is nonnegative.

Now,

$$S'_l = (A^k - B'^k - C^k + D'^k) - (\gamma^k - \delta^k - \alpha^k + \beta^k) = g(D') - g(\beta) \tag{24}$$

and

$$D' - \beta \geqq D - \beta = \sum_{i=l+1}^{l+d} (x_{j-i} - (T_jx)_i) = \sum_{i=l+1}^{j-l-d-1} (x_{j-i} - (T_jx)_i) \geqq 0, \tag{25}$$

so the right-hand side of (24) is nonnegative.

Second case. $x_l = (T_jx)_{j-l}$. Repeat the proof of the previous case with a redefinition of B and δ (rather than B and D) as B' and δ' in $A - B' = \gamma - \delta' = x_{j-l}$.

Instead of (24), use

$$S'_l = (A^k - B'^k - \gamma^k + \delta'^k) - (C^k - D^k - \alpha^k + \beta^k) = g(\delta') - g(\beta), \tag{26}$$

where in the definition of g, $u = x_{j-l}$ and $v = D - \beta$.

$D - \beta$ was shown to be nonnegative in (25), and $\delta' - \beta \geqq \delta - \beta$, which was shown to be nonnegative in (23).

Case (iii). Set $l^* = (j-d-1)/2$. Then $\sum_{i=l^*+1}^{l^*+d} (T_jx)_i = \sum_{i=l^*+1}^{l^*+d} x_i$.

Set $L = j-d-1$. By adding two zeros and conveniently shifting the indexing of the sums, we have

$$\begin{aligned}\Delta(x, j, d, k) = &\sum_{l=1}^{l^*} \left[\left(\sum_{i=l}^{l+d} (T_jx)_i \right)^k - \left(\sum_{i=l+1}^{l+d} (T_jx)_i \right)^k - \left(\sum_{i=l}^{l+d} x_i \right)^k + \left(\sum_{i=l+1}^{l+d} x_i \right)^k \right] \\ &+ \sum_{l=l^*+1}^{L} \left[\left(\sum_{i=l}^{l+d} (T_jx)_i \right)^k - \left(\sum_{i=l}^{l+d-1} (T_jx)_i \right)^k - \left(\sum_{i=l}^{l+d} x_i \right)^k + \left(\sum_{i=l}^{l+d-1} x_i \right)^k \right] \\ &+ \sum_{l=L+1}^{n-d} \left[\left(\sum_{i=l}^{l+d} (T_jx)_i \right)^k - \left(\sum_{i=l}^{l+d-1} (T_jx)_i \right)^k - \left(\sum_{i=l}^{l+d} x_i \right)^k + \left(\sum_{i=l}^{l+d-1} x_i \right)^k \right].\end{aligned} \tag{27}$$

From this point on, the proof is identical to that of Case (ii). □

It should be observed that the proof of the nonnegativity of the summands in Case (i), the proofs of the nonnegativity of $S_l - S_{l'}$ for the summands corresponding to $l \leqq l^* - 1$ in Case (ii) and to $l \leqq l^*$ in Case (iii), as well as the proof of the nonnegativity of the summands corresponding to $l \geqq L+1$ in Cases (ii) and (iii) depend on the *convexity* of t^k and not on the convexity of its *derivative*. Hence, for all $k \geqq 2$, *if* $T_jx \equiv x$ then $\Delta(x, j, d, k)$ is strictly positive simultaneously for all $d \in [1, n-2]$. So, the organ pipe permutations are the only stochastic minimizers of D_k.

This finishes the proof of Theorem 1. □

3. Proof of Theorem 2.

LEMMA 4. *For fixed x and f (symmetric and concave), let $W(\alpha)$ be the value of $V(\pi)$ for greedy permutations π, as a function of the left endpoint of the house. Then $W(\alpha)$ is continuous.*

If $\alpha' \leqq \alpha$ and $f(\alpha) \leqq f(\alpha+s)$ or if $\alpha' \geqq \alpha$ and $f(\alpha) \geqq f(\alpha+s)$, then $W(\alpha') \leqq W(\alpha)$. In particular, if $f(\alpha) = f(\alpha+s)$, α is a maximum point of W.

Proof. We first prove the continuity of the function $W(\alpha)$. Let α_1 and α_2 denote the leftmost maximum and rightmost maximum of f, respectively. In particular each x such that $\alpha_1 \leqq x \leqq \alpha_2$ is a maximum point of f. It is clear that W is continuous at all points α at which the greedy permutation does not change, since the heights of the walls vary continuously. (For example, W is continuous for $\alpha \geqq \alpha_2$ or $\alpha \leqq \alpha_1 - s$.) Consider a point α at which the greedy permutation changes. Such an α is reached, when "sliding" the house to the right, whenever two walls become of the same height. At any such point reverse the order of the rooms in the subhouse between these two walls. (If more than two walls become of equal height, consider the two walls, having the same height, which are furthest away from each other.) This reversal will not affect W at α, and will give an equivalent permutation that will remain greedy when α is increased. Therefore W is continuous at α.

Next we prove that $f(\alpha) \leqq f(\alpha+s)$ implies that W is monotone nondecreasing on the interval $(-\infty, \alpha)$. (The proof for the case $f(\alpha) > f(\alpha+s)$ will then follow from the symmetry of f.) Let $\beta < \alpha$. The monotonicity at β is obviously satisfied if $\beta + s < \alpha_2$. Thus, suppose $\beta < \alpha_1 \leqq \alpha_2 \leqq \beta + s$, and consider the $n+1$ walls defining $W(\beta)$. Those walls to the right of or at α_2 can be mapped in a one-to-one way into the walls to the left of that point in such a way that each wall on the right is mapped into a wall of strictly smaller height. (The shortest wall to the right of or at α_2 is mapped into the shortest wall to the left of α_2, the second shortest wall to the left of or at α_2 is mapped into the second shortest wall to the left of α_2, etc.) Because of the concavity and symmetry of f, the sum of the heights of the walls on the right will decrease (as β increases) by an amount which is less than or equal to the amount by which the sum of the heights of the walls they are mapped into will increase. Since the sum of the unmapped excess walls on the left will obviously not decrease, our local argument, coupled with the continuity of W, completes the proof. □

Proof of Theorem 2. Using an inductive argument, it is enough to prove that any permutation π that places a nonminimal room at a lowest end and proceeds thereafter greedily on the remaining subhouse, can be improved by some π' that places a minimal room at a lowest end. For concreteness suppose that $x_1 \leqq x_2 \leqq \cdots \leqq x_n$ and

$$f(\alpha) \leqq f(\alpha+s). \tag{28}$$

Let π be a permutation that places first (at α) a room of length $z > x_1$. We may assume that on the remaining house, with floor $[\alpha+z, \alpha+s]$, it behaves greedily. There are three cases to consider (see Fig. 1):

Case 1. $f(\alpha+z) \leqq f(\alpha+s)$ and π places in second place a room of length x_1.

Case 2. $f(\alpha+z) \geqq f(\alpha+s)$, π places in second place a room of length $y > x_1$ and $f(\alpha+z) \leqq f(\alpha+s-x_1)$.

Case 3. $f(\alpha+z) \geqq f(\alpha+s)$, π places in second place a room of length $y > x_1$ and $f(\alpha+z) > f(\alpha+s-x_1)$.

In Case 1, exchange the first two rooms. Since f is monotone nondecreasing on $[\alpha, \alpha+z]$, $f(\alpha+x_1) \leqq f(\alpha+z)$. The change in V produced by this exchange is the difference $f(\alpha+x_1) - f(\alpha+z) \leqq 0$.

In Case 2, consider the subhouse with floor $[\alpha+z, a+s-x_1]$. Keeping the lengths of its rooms constant and their inner order greedy, "slide" the subhouse to the left until its leftmost point is at a distance x_1 to the right of α. By Lemma 4, this sliding process does not increase the value of V. The result is a permutation as required. Finally, in Case 3, we cannot slide the house as in Case 2 because that may increase V for some sliding distance. To circumvent this difficulty, observe that the subhouse with floor $[\alpha+z, \alpha+s-x_1]$ is the mirror image (with respect to the axis of symmetry of f) of another subhouse, with floor $[A, B]$ (see Fig. 1). Note that the induced assignment of the $n-2$ rooms to the interval $[A, B]$ is greedy and has the same V value as the original assignment of these rooms to the interval $[\alpha+z, \alpha+s-x_1]$. Thus, consider the subhouse with the floor $[A, B]$ instead of that with floor $[\alpha+z, a+s-x_1]$. Now, since $f(A)=f(\alpha+s-x_1)<f(\alpha+z)=f(B)$, the sum of the heights of the walls does not increase when we slide this subhouse to the left (as in Case 2) until its leftmost point is at a distance x_1 to the right of α. (Note that $f(A)=f(\alpha+s-x_1)$ and $f(\alpha)\leqq f(\alpha+s)$ imply that A is at a distance of at least x_1 from α.) Again, the result is a permutation as required. □

COROLLARY 1. *For $n, k\geqq 2$ the organ pipe permutations are optimal for the range, the leftist, rightist and alternating rules.*

Proof. Following [2], let $\mathrm{MIN}_i X_i=L$, $\mathrm{MAX}_i X_i=R$ and let (L_0, R_0) be distributed like and independently of (L, R). Let COST stand for expected head moving time under leftist or rightist (L) and alternating (A) rules. Let $R-L=\mathrm{RANGE}$. Then (see [2, Thm. 1])

$$\mathrm{COST}_L=2\sum_{i=1}^{n-1}P(L\leqq i)P(R>i)=2\sum_{i=1}^{n-1}\left(1-\left(\sum_{j=i+1}^{n}p_j\right)^k\right)\left(1-\left(\sum_{j=1}^{i}p_j\right)^k\right)=2\sum_{i=1}^{n}f_L\left(\sum_{j=1}^{i}p_j\right), \tag{29}$$

where

$$f_L(x)=(1-x^k)(1-(1-x)^k). \tag{30}$$

This function is obviously symmetric around $\frac{1}{2}$. It is enough to check concavity on $[0, 1]$, as f_L may be extended outside $[0, 1]$ by straight lines.

$$\frac{d^2f_L}{dx^2}=-k(k-1)[1-(1-x^{k-2})(1-(1-x)^{k-2})]-2k(2k-1)(x(1-x))^{k-1},$$

which is clearly negative on $[0, 1]$.

As for the range, (see [2, Lemma 1])

$$E(\mathrm{RANGE}_k)=\sum_{i=1}^{n-1}(1-P(L>i)-P(R\leqq i))=\sum_{i=1}^{n-1}\left(1-\left(\sum_{j=i+1}^{n}p_j\right)^k-\left(\sum_{j=1}^{i}p_j\right)^k\right)=\sum_{i=1}^{n}f\left(\sum_{j=1}^{i}p_j\right), \tag{31}$$

where

$$f(x)=1-x^k-(1-x)^k \tag{32}$$

is obviously symmetric and concave on $[0, 1]$. As for the alternate rule, (see [2, Thm. 2])

$$\mathrm{COST}_A=E(\mathrm{RANGE}_{2k}). \tag{33}$$

□

We will now present a counterexample to Theorem 2 to show that the symmetry assumption may not be dropped.

Counterexample. The greedy permutations can be described in two different forms that are equivalent when f is a symmetric function;

(i) place the shortest room at the lowest end; or

(ii) minimize the height of the first wall to be erected.

When f is not symmetric, (i) and (ii) are not necessarily equivalent. Neither yields an optimal permutation. We will show by example that even a permutation satisfying both (i) and (ii) may fail to be optimal. Let $x_1 = 0.2$, $x_2 = x_3 = 0.4$ and define f on $[0, 1]$ by

$$f(x) = \min\{0.5 + x, 0.75(1 - x)\}.$$

If x_1 is placed last (lowest end, lowest first wall of height 0.15) the sum of the heights of the two inner walls is $f(0.4) + f(0.8) = 0.45 + 0.15 = 0.6$; whereas, if x_1 is placed first (highest end, first wall of height 0.25), the sum of the heights is $f(0.2) + f(0.6) = 0.25 + 0.3 = 0.55$. The greedy and optimal solutions are shown in Fig. 2.

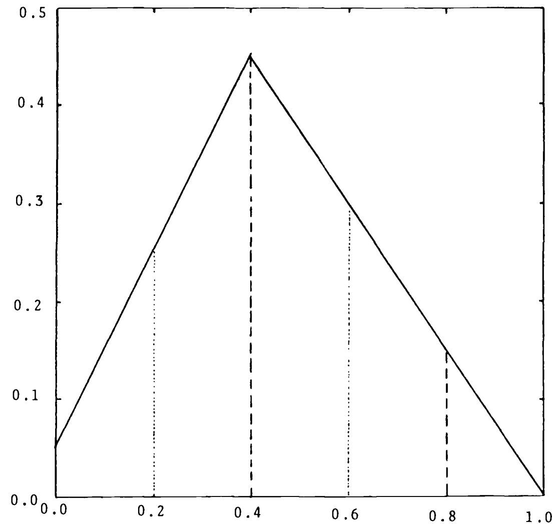


FIG. 2. *Comparison of greedy and optimal wall placements,*
···· *Optimal* --- *Greedy*

REFERENCES

[1] P. P. BERGMANS, *Minimizing expected travel time on geometrical patterns by* OPTIONAL *probability rearrangements*, Inform. and Control, 20 (1972), pp. 331–350.

[2] J. R. BITNER AND C. K. WONG, *Optimal and near-optimal scheduling algorithms for batched processing in linear storage*, this Journal, 8 (1979), pp. 479–498.

[3] E. V. LEHMANN, *Testing Statistical Hypotheses*, John Wiley, New York, 1959.

[4] I. MEILIJSON AND A. TAMIR, *Scheduling tasks on several processors with variable speed*, Oper. Res., to appear.

[5] C. K. WONG, *Minimizing expected head movement in a one-dimensional and two-dimensional mass storage system*, Computing Surveys, 12 (1980), pp. 167–178.

SIAM J. COMPUT.
Vol. 13, No. 3, August 1984

© 1984 Society for Industrial and Applied Mathematics
006

A MATHEMATICAL MODEL FOR THE VERIFICATION OF SYSTOLIC NETWORKS*

RAMI G. MELHEM† AND WERNER C. RHEINBOLDT‡

Abstract. A mathematical model for systolic architectures is suggested and used to verify the operation of certain systolic networks. The data items appearing on the communication links of such a network at successive time units are represented by data sequences and the computations performed by the network cells are modeled by a system of difference equations involving operations on the various data sequences. The input/output descriptions, which describe the global effect of the computations performed by the network, are obtained by solving this system of difference equations. This input/output description can then be used to verify the operation of the network. The suggested verification technique is applied to four different systolic networks proposed in the literature.

Key words. systolic networks, verification, data sequences, causal operators

1. Introduction. Systolic architectures, pioneered by H. T. Kung, are becoming increasingly attractive because of continuous advances in VLSI technology. This type of network architectures has two properties very desirable in VLSI implementations, namely, regularity and the local nature of the interconnections.

A systolic network can be viewed as a network composed of a few types of computational cells, regularly interconnected via local data links and organized such that streams of data flow smoothly within the network. For an introduction to systolic architectures, we refer to [1] where further references to specific examples are given.

As an introductory example, we briefly review a simple systolic network for the computation of one-dimensional convolution expressions [1]. More specifically, given a sequence of numbers $\{x_1, x_2, \cdots, x_n\}$, and a sequence of weights $\{w_1, w_2, \cdots, w_k\}$, we want to compute the sequence $\{y_1, y_2, \cdots, y_{n+1-k}\}$ where each y_i is defined by

$$y_i = \sum_{j=1}^{k} w_j x_{i+j-1}. \tag{1.1}$$

Figure 1 shows the building cell of the one-dimensional convolution network under discussion. It is a multiply/add cell with a one-word memory to store a real number w.

At each clock pulse, the cell receives two input data items, x_{in} and y_{in}, performs its computation, and delivers at the next clock pulse the outputs $x_0 = x_{\text{in}}$ and $y_0 = y_{\text{in}} + wx_{\text{in}}$. Figure 2 shows three such cells connected into a network that performs the convolution calculation for the case $k = 3$. The elements $x_1, x_2, \cdots, x_n$ are pumped in at the left end of the network, each separated from the other by one time unit, and zeros are pumped in at the right end. To illustrate the operation of the array, we show in Fig. 3 the relative location and value of each data item at times $t = 3$, 4, 5 and 6, where $t = 1$ is the time at which the array started its execution. By following the data paths, we can convince ourselves that the output of the array will include the sequence $\{y_1, y_2, \cdots, y_{n+1-k}\}$.

* Received by the editors October 20, 1982, and in revised form May 13, 1983. This work was supported in part by the Office of Naval Research under contract N-0014-80-C-0455 and the U.S. Air Force Office of Scientific Research under grant 80-0176.

† Department of Computer Science and Department of Mathematics and Statistics, University of Pittsburgh, Pittsburgh, Pennsylvania 15260.

‡ Department of Mathematics and Statistics, University of Pittsburgh, Pittsburgh, Pennsylvania 15260.

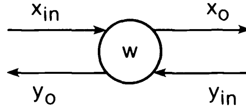


FIG. 1

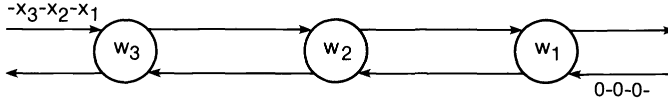
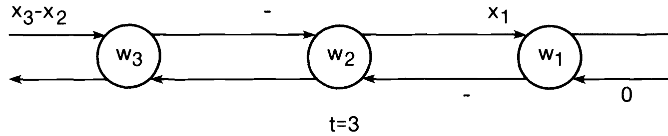
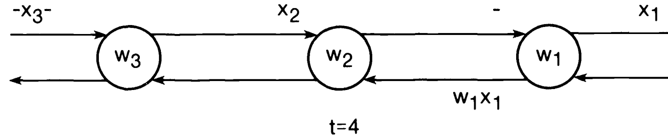
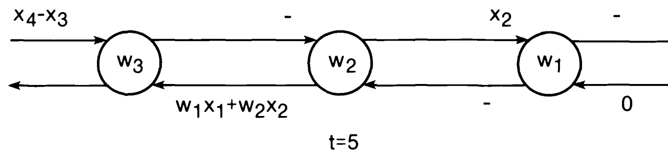


FIG. 2







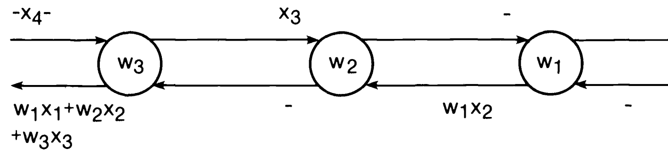


FIG. 3

Although the concept of systolic networks is very well developed, few techniques appear to be known for a formal verification of the operation of such networks. In [2] and [3], methods for proving correctness properties of systolic networks were given based on proof techniques similar to those used in the verification of programs and distributed systems [4]. These methods do not make use of the special properties of systolic networks and hence give only rather general results.

In [5], a formal approach for the representation of computational networks was proposed. This approach was elaborated in [6], [7], [8] where the so-called wave-front notation was used to map algorithmic descriptions into systolic implementations. Although this notation provides a powerful tool that can be used in the automatic design of systolic arrays [9], it does not appear to have the flexibility needed to describe general systolic networks.

In this paper, we suggest a technique designed specifically for verifying the operation of systolic networks for which the pattern of the input is known in advance. This excludes networks for which the arrival time of the inputs is not predictable, as in the case with the systolic priority queue suggested in [10]. In § 2.1 the data sequences are introduced to represent the data appearing on the communication links at successive time intervals. In the same section, we discuss the causal operators which model the computations performed by a cell of the network. This concept was primarily inspired by corresponding approaches in systems theory [11].

In §§ 2.2 and 2.3, we present the mathematical model on which the verification technique is based. This model carries some of the properties of a model called "automaton networks" [12], which in turn is a modification of the von Neumann cellular array [13], [14]. It also carries some properties of an abstract model [15] used by Leiserson and Saxe to prove that any synchronous system can be converted to an equivalent systolic system.

In § 3 we describe the different steps of the suggested techniques and give a simple illustrative example. Finally, in §§ 4, 5 and 6, we demonstrate the technique by applying it to the verification of some realistic systolic networks that have appeared in the literature.

2. An abstract systolic model.

2.1. Data sequences and causal relations. We define a data sequence to be an infinite sequence whose elements are members of the set $R_\delta = R \cup \{\delta\}$, where R is the set of real numbers and δ denotes a special element, not belonging to R, called the "don't care element". We extend any operator defined on R to R_δ in one of the following two ways: 1) By adding the rule that the result of any operator involving δ is δ. For example,

$$\delta \text{ ``op'' } x = x \text{ ``op'' } \delta = \delta \quad \text{for all } x \in R_\delta.$$

This class of operators on R_δ will be called δ-regular operators, 2) By treating δ as a special symbol that affects the result of the operation. This class will be called non-δ-regular operators. For example, we will consider later the binary operator $\oplus$ such that for any $x, y \in R_\delta$

$$x \oplus y = x + y \quad \text{if } x, y \neq \delta, \qquad x \oplus \delta = \delta \oplus x = x. \tag{2.1}$$

Two other non-δ-regular operators that will be used in § 6 are the operators $\min_\delta$ and $\max_\delta$ defined on an ordered pair (x, y), $x, y \in R_\delta$, by

$$\min_\delta(x, y) = \begin{cases} \min\{x, y\} & \text{if } x, y \neq \delta, \\ y & \text{if } x = \delta \text{ or } y = \delta, \end{cases}$$

and

$$\max_\delta(x, y) = \begin{cases} \max\{x, y\} & \text{if } x, y \neq \delta, \\ x & \text{if } x = \delta \text{ or } y = \delta, \end{cases}$$

where min { } and max { } carry the usual meaning on R. The reason for distinguishing between δ-regular and non-δ-regular operators will become clearer in § 2.3.

Let N be the set of positive integers. Then any data sequence η is defined as a mapping from N to R_δ; that is, the image element $\eta(i)$, $i \in N$, is the ith element in the sequence. The set of all data sequences, that is the set of all such mappings, will be denoted by $R_\delta^* = \{\eta \mid \eta : N \to R_\delta\}$.

Any arithmetic operation on R_δ is extended to R_δ^* by applying the operation elementwise to the elements of the sequences with δ being the result of any undefined operation. For example, if "op" is a binary operation defined on R_δ, then for all $\eta_1, \eta_2 \in R_\delta^*$, we have η_1 "op" $\eta_2 = \eta_3$ where for all $i \in N$. $\eta_3(i)$ is given by

$$\eta_3(i) = \begin{cases} \eta_1(i) \text{ "op" } \eta_2(i) & \text{if } \eta_3(i) \text{ is defined,} \\ \delta & \text{otherwise.} \end{cases}$$

We will also use scalar operations on sequences. For example, the scalar product of a sequence $\eta \in R_\delta$ and a number $w \in R$ is defined as the sequence $\zeta = w \cdot \eta \in R_\delta^*$ for which $\zeta(i) = w\eta(i)$, $i \in N$.

Given the previous definition of data sequence, we define the set of bounded data sequences $\bar{R}_\delta \subset R_\delta^*$ to contain those sequences having only a finite number of non-δ-elements. It is then natural to introduce the termination function $T: \bar{R}_\delta \to N$ such that, for any $\eta \in \bar{R}_\delta$, $T(\eta)$ is the position of the last non-δ element in η; in other words:

$$\text{for any } \eta \in \bar{R}_\delta, \quad T(\eta) = i \leftrightarrow \eta(i) \neq \delta \text{ and } \eta(j) = \delta \text{ for } j > i.$$

In this paper, we will denote bounded data sequences by small Greek letters and simply refer to them as sequences. This will not cause any confusion because we will never consider anything but bounded data sequences.

In addition to the operators extended from R_δ to $\bar{R}_\delta$, we may also define operators directly on $\bar{R}_\delta$. In general, an n-ary sequence operator Γ is a transformation $\Gamma: [\bar{R}_\delta]^n \to \bar{R}_\delta$ where $[\bar{R}_\delta]^n = \bar{R}_\delta \times \bar{R}_\delta \times \cdots \times \bar{R}_\delta$ is the cartesian product space of n copies of $\bar{R}_\delta$. Two basic unary operators that will be frequently used in this paper are the shift operator Ω^k and the spread operator θ^r defined by:

$$\Omega^k \xi = \eta \quad \text{and} \quad \theta^r \xi = \zeta,$$

where

$$\eta(i) = \begin{cases} \delta & \text{if } i \leqq k, \\ \xi(i-k) & \text{if } i > k, \end{cases}$$

$$\zeta(i) = \begin{cases} \xi\left(\dfrac{i+r}{r+1}\right), & i = 1, r+2, 2r+3, \cdots, (n-1)r+n, \cdots, \\ \delta & \text{otherwise.} \end{cases}$$

More descriptively, Ω^k inserts k δ-elements at the beginning of a sequence, while θ^r inserts r δ-elements between every two elements of a sequence. For example, if $\xi = a_1$, a_2, a_3, a_4, δ, $\delta, \cdots$, then $T(\xi) = 4$ and

$$\xi(i) = a_i, \qquad 1 \leqq i \leqq T(\xi),$$

$$\Omega^3 \xi = \delta, \delta, \delta, a_1, a_2, a_3, a_4, \delta, \delta, \delta, \cdots,$$

$$\theta^2 \xi = a_1, \delta, \delta, a_2, \delta, \delta, a_3, \delta, \delta, a_4, \delta, \delta, \cdots.$$

It is easy to verify that the termination function generally satisfies

$$T(\Omega^k \xi) = T(\xi) + k, \qquad T(\theta^r \xi) = (r+1)T(\xi) - r.$$

It is also clear that we can define a sequence operator by combining previously defined sequence operators. For example, we might define an operator $\Gamma: \bar{R}_\delta \times \bar{R}_\delta \times \bar{R}_\delta \to \bar{R}_\delta$ as follows:

$$\Gamma(\xi, \eta, \zeta) = \Omega[\xi + \eta * \zeta]$$

where square brackets are used for grouping and parenthesis for enclosing the arguments of the operator.

We next define a causal operator to be any n-ary sequence operator $\Gamma:[\bar{R}_\delta]^n \to \bar{R}_\delta$ which satisfies the causality property in the sense that the ith element of any of its operands can only affect the jth element of its image for $j > i$. In order to formulate this more precisely, assume that for any given sequences $\eta_r \in \bar{R}_\delta$, $r = 1, 2, \cdots, n$, the image under Γ is $\xi = \Gamma(\eta_1, \cdots, \eta_r, \cdots, \eta_n)$. Then Γ is a causal operator if by replacing any operands η_r by another sequence η'_r satisfying

$$\eta'_r(t) = \eta_r(t), \qquad 1 \leqq t < i,$$

the resulting image $\xi' = \Gamma(\eta_1, \cdots, \eta'_r, \cdots, \eta_n)$ satisfies

$$\xi'(t) = \xi(t), \qquad 1 \leqq t \leqq i.$$

In other words, the value of $\xi(i)$ depends only on the first $i-1$ elements of η_r, $1 \leqq r \leqq n$.

Similarly, we may define weakly causal operators for which the ith element of the image sequence $\xi(i)$ depends only on the first i elements of the operands η_r, $1 \leqq r \leqq n$, instead of the first $i-1$ elements. With this, it is easily seen that the combination $\Gamma^1\Gamma^2$ (or $\Gamma^2\Gamma^1$) of a causal operator Γ^1 and a weakly causal operator Γ^2 is a causal operator. For instance, the shift operator Ω^k is causal and the spread operator θ^r is weakly causal; hence, the combined operator $\Omega^k\theta^r$ is causal.

2.2. The abstract model. In order to define the mathematical model used in our verification technique, we define as usual a loopless multigraph $G(V, E, \varphi_-, \varphi_+)$ to be composed of

(a) a set V of nodes,

(b) a set E of directed edges,

(c) two functions $\varphi_-, \varphi_+ : E \to V$ satisfying the condition that for any edge $e \in E$

$$\varphi_-(e) \neq \varphi_+(e). \tag{2.2}$$

For each edge $e \in E$, the nodes $\varphi_-(e)$ and $\varphi_+(e)$ are the source and destination node, respectively, of that edge. Clearly, the condition (2.2) prevents any direct loops in the graph. This definition of a multigraph allows any two nodes to be connected by more than one edge in the same direction, a property that may be useful when we represent systolic networks by this abstract model.

As usual in graph terminology, for any node $v \in V$, the edges $\{e;\ \varphi_-(e) = v\}$ directed out of v are termed the OUT edges of v, while the edges $\{e;\ \varphi_+(e) = v\}$ directed into v are termed the IN edges of v. Accordingly, the IN-degree and OUT-degree of v are the number of IN edges and OUT edges of v, respectively. Any node $v \in V$ with IN-degree zero or OUT-degree zero is called a source or a sink, respectively. All other nodes are called interior nodes of G. We shall use the notation V_S, V_T and V_I for the subsets of V containing the source, sink and interior nodes of V, respectively. Of course, the condition $V_S \cup V_T \cup V_I = V$ is always satisfied.

With this notion of a multigraph, we define our abstract systolic model to be composed of the following components:

[A1] A multigraph $G(V, E, \varphi_-, \varphi_+)$.

[A2] A coloring function col: $E \to C_E$, which maps E into a given finite set of colors C_E, and hence assigns a color to each edge in E. The coloring function is assumed to satisfy the condition that the different IN edges of a node have different colors, and correspondingly that the different OUT edges of a node have different colors. Edge colors $y = \text{col}(e)$ will be denoted by lower case letter.

[A3] For each edge $e \in E$, a sequence $\xi_e \in \bar{R}_\delta$ is specified.

[A4] For each interior node $v \in V$ with IN-degree m and OUT-degree n, we are given n causal m-ary operators $\Gamma_v^i: [\bar{R}_\delta]^m \to \bar{R}_\delta$ which specify the "node I/O description". More specifically, if η^j, $j = 1, 2, \cdots, m$, and ξ^i, $i = 1, 2, \cdots, n$, are the sequences associated with the IN and OUT edges of v, respectively, then the n relations

$$\xi^i = \Gamma_v^i(\eta^1, \eta^2, \cdots, \eta^m), \qquad i = 1, 2, \cdots, n,$$

are the I/O description of v. The different IN and OUT edges of v are distinguished in the I/O description by their colors.

Since by condition [A2] all edges terminating at a given node v have different colors, it follows that any edge $e \in E$ is uniquely identified by a pair (y, v), where $y = \text{col}(e)$ and $v = \varphi_+(e)$. To simplify the notation, the pair (y, v) will often be written in the form y_v, and the sequence associated with that edge will be identified by the symbol η_v, where we replaced the letter y by its corresponding Greek letter η.

For practical applications, it is generally desirable to identify the nodes of the network by appropriate labels which correspond to the problem at hand. This means that we introduce a set L of labels together with a one-to-one function $\psi: V \to L$ from V onto L. In our examples, we usually identify the nodes with their labels directly.

Having defined the general abstract model, we next show how it can be used to define a general systolic network.

2.3. The general systolic network. By giving a physical interpretation to each component in the general abstract model we obtain a general systolic network. The basic idea of this interpretation may be summarized as follows:

Each interior node represents a computational cell and each source/sink node corresponds to an input/output cell for the overall network. To distinguish in our figures the computational cells from the I/O cells, we depict computational cells by circles and I/O cells by squares.

Each edge $x_v \in E$ represents a unidirectional communication link between the two cells it connects. The sequence associated with x_v then comprises the data items that appeared on it in consecutive time units. More specifically, if ξ_v is the sequence associated with x_v, then the ith element of ξ_v, namely $\xi_v(i)$, is the data item that appeared on x_v at time $t = i$ units, where $t = 1$ is the time at which the network started its operation.

For an interior node, the node I/O description describes the computation performed by the cell corresponding to that node. We illustrate this with two simple examples.

Example 1. The node shown in Fig. 4 represents a simple latch cell which produces at any time $t > 1$ on its output link the same data item that appeared on its input link

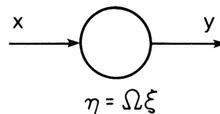


FIG. 4

at time $t-1$. At time $t = 1$, we have $\eta(1) = \delta$, which corresponds to the fact that at the beginning of the network operation, no specific data item appeared on the output link.

Example 2. The operation of the multiply/add cell mentioned in § 1 and shown in Fig. 1 may be represented by the following node I/O descriptions:

(2.3a) $$\xi_o = \Omega \xi_{in},$$

(2.3b) $$\eta_o = \Omega[\eta_{in} + w \cdot \xi_{in}]$$

where $w \in R$ is a given real number and ξ_{in}, η_{in}, ξ_o and η_o are the input and output sequences of the node as shown in Fig. 5.

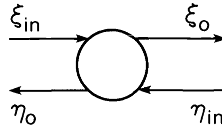


FIG. 5

Since in any practical dynamical system any data item produced by a computational cell at time t depends only on the data provided to that cell at times less than t, we immediately see the importance of the condition imposed in § 2.2 on the node I/O descriptions, namely that only causal operators in the sense of § 2.1 are used. We also note that with the model described above, the computational power of each cell is not limited to simple arithmetical operations. In other words, a cell could be an intelligent cell that can perform elaborate calculations provided that we can express these calculations in terms of causal operators.

At this point, it may be useful to note that if a δ-regular operator is used to model a computational cell, then this cell treats δ as a "don't know" quantity, and consequently, the result of any operation cannot be known if any of the operands is not known. On the other hand, non-δ-regular operators are used to model computational cells which treat δ as a special symbol that affects the result of the operation. Hence, each physical communication link in networks containing cells of this type should be augmented by an additional wire to indicate whether the link carries valid data or not. The operation of each cell is then dependent on this additional piece of information.

We call "network output sequences" those sequences associated with the IN edges of sink nodes, and "network input sequences" those associated with the OUT edges of source nodes. Then the system of all node I/O descriptions provides a specification of the computation performed by the network in the form of an implicit relation between the network input and output sequences. This relation will be called the "network I/O description".

As a simple example, consider the hypothetical network with the graph shown in Fig. 6. In this graph, we assume that the edges directed to the left are given the color y and those directed to the right the color x. We also follow the naming convention mentioned in § 2.2 in identifying the different edges in the graph. To complete the network description, a node I/O description has to be specified for each node in the

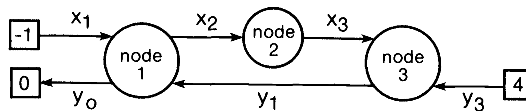


FIG. 6

graph. Assume that these are given by the following causal relations:

(2.4a) For node 1: $\xi_2 = \Omega[\xi_1 + \eta_1]$,

(2.4b) $\eta_0 = \Omega[\xi_1 * \eta_1]$,

(2.5) For node 2: $\xi_3 = \Omega \xi_2$,

(2.6) For node 3: $\eta_1 = \Omega[\xi_3 * \eta_3]$.

For this network, η_3 and ξ_1 are the network input sequences and η_0 is the network output sequence. In order to obtain the network I/O description explicitly, we have to solve (2.4), (2.5) and (2.6); that is, we have to obtain an explicit expression for η_0 in terms of ξ_1 and η_3.

Generally, it is very difficult, and sometimes impossible, to derive an explicit solution of the system of node I/O equations. In the next section, we show that this task may be greatly simplified in the case of certain networks with a homogeneous structure.

3. Homogeneous systolic networks. By condition [A2], any edge $e \in E$ is uniquely identified by its color and one of its incident nodes. In fact, we used this already as a convenient means for identifying edges by their color and terminal node. Let $M \subset C_E \times V_I$ be the set of all pairs (y, v), $y \in C_E$, $v = V_I$, for which there is an edge $e \in E$ with $y = \operatorname{col}(e)$ and $v = \varphi_-(e)$. Then the terminal node $u = \varphi_+(e)$ is uniquely given, and hence the successor function $\mu: M \to V_I \cup V_T$ is well defined by the association

$$(y, v) \in M,\ y = \operatorname{col}(e),\ v = \varphi_-(e) \to \mu(y, v) = \varphi_+(e).$$

In other words, if there exists an edge e with color y and starting node v, then $\mu(y, v)$ is the terminal node of e.

Given a systolic network based on the graph $G = \{V, E, \varphi_-, \varphi_+\}$, a subset $V'_I \subset V_I$ of interior nodes is said to be a homogeneous set if:

[H1] All the nodes in V'_I have identical IN and OUT degrees, say m and n, respectively.

[H2] The m colors of the IN edges of any interior node $v \in V'_I$ are identical and so are the n colors of the OUT edges of v. Denote the colors of the IN and OUT edges of v by $y^1, y^2, \cdots, y^m$ and $z^1, z^2, \cdots, z^n$, respectively.

[H3] The node I/O descriptions of any interior node $v \in V'_I$ are generic in the sense that they may be written in the form:

$$\zeta^i_{\mu(z^i, v)} = \Gamma^i(\eta^1_v, \eta^2_v, \cdots, \eta^m_v), \qquad i = 1, 2, \cdots, n,$$

where Γ^i, $i = 1, 2, \cdots, n$ are given n-ary operators which are independent of the particular node in V'_I, μ is the successor function defined earlier in this section, and η^j_v, $j = 1, 2, \cdots, m$, and $\zeta^i_{\mu(z^i, v)}$, $i = 1, 2, \cdots, n$, are the sequences associated with the IN and OUT edges of v, respectively.

A network is said to be homogeneous if the set of interior nodes V_I in its graph G is a homogeneous set. More generally, if there exists a partition $V_I = V^1_I \cup V^2_I \cup \cdots \cup V^k_I$ of V_I into k nonempty homogeneous subsets $V^1_I, V^2_I, \cdots, V^k_I$, then the network is said to be k-partially homogeneous.

The main advantage of having a homogeneous (or partially homogeneous) network is that the resulting system of equations has a repetitive pattern, which, in many cases, allows us to obtain an analytical solution to the system. This should become clearer as we proceed with the different examples.

To verify the operation of a systolic network, we are generally interested in its behavior for specific inputs; that is, we wish to find the form of the network output sequences for specific network input sequences. This is usually accomplished by substituting the given input sequences in the network I/O description and manipulating the resulting equations to obtain the description of the network output sequences.

As a first example of our verification technique, we consider again the one-dimensional convolution network described in § 1. The graph of this network is shown in Fig. 7, where we assumed that the edges directed to the left have the color s, while

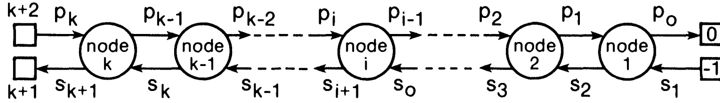


FIG. 7

those directed to the right have the color p. The nodes of the graph are identified by the integers $-1, 0, 1, 2, \cdots, k+1, k+2$, where nodes -1 and $k+2$ are source nodes, nodes 0 and $k+1$ sink nodes, and nodes 1 through k interior nodes. The successor function is defined for any interior node $i=1, 2, \cdots, k$ by

$$\mu(y, i)=\begin{cases} i+1 & \text{if } y=s, \\ i-1 & \text{if } y=p. \end{cases}$$

Our goal is to verify that the network indeed produces the results of (1.1) for the network input sequences described by

$$\sigma_1=\Omega^{k-1}\theta\iota, \tag{3.1a}$$

$$\pi_k=\theta\xi \tag{3.1b}$$

where

$$T(\iota)=n-(k-1), \quad \iota(t)=0, \qquad T(\xi)=n, \quad \xi(t)=x_t.$$

The I/O description of a typical interior node i in the graph, $1\leqq i\leqq k$, is given by the following causal relations:

$$\pi_{i-1}=\Omega\pi_i, \tag{3.2a}$$

$$\sigma_{i+1}=\Omega[\sigma_i+w_i\cdot\pi_i]. \tag{3.2b}$$

This system of difference equations is easily solved. First, note that the solution of (3.2a) obviously is

$$\pi_i=\Omega^{k-i}\pi_k. \tag{3.3}$$

By substituting this in (3.2b) we obtain

$$\sigma_{i+1}=\Omega\sigma_i+w_i\cdot[\Omega^{k-i+1}\pi_k]. \tag{3.4}$$

The solution of (3.4) is then given by Lemma 1 in the appendix as:

$$\begin{aligned}\sigma_{k+1}&=\Omega^k\sigma_1+\sum_{j=1}^{k}\Omega^{j-1}[w_{k-j+1}\cdot\Omega^{k-(k-j+1)+1}\pi_k]\\&=\Omega^k\sigma_1+\sum_{j=1}^{k}\Omega^{2j-1}[w_{k-j+1}\cdot\pi_k].\end{aligned}\tag{3.5}$$

This is the I/O description for the network.

In order to find the specific form of the output sequence σ_{k+1} for the input sequences (3.1), we substitute these sequences into (3.5) and obtain

$$\sigma_{k+1}=\Omega^{2k-1}\theta\iota+\sum_{j=1}^{k}\Omega^{2j-1}[w_{k-j+1}\cdot\theta\xi].$$

By the properties P1, P2, P3 and P4 in the Appendix, this may be rewritten as

$$\begin{aligned}\sigma_{k+1}&=\Omega^{2k-1}\theta\iota+\Omega\sum_{j=1}^{k}\Omega^{2(j-1)}\theta[w_{k-j+1}\cdot\xi]\\&=\Omega^{2k-1}\theta\iota+\Omega\theta\sum_{j=1}^{k}\Omega^{j-1}[w_{k-j+1}\cdot\xi]\\&=\Omega^{2k-1}\theta\iota+\Omega\theta\sum_{j=1}^{k}\Omega^{j-1}\eta_j,\end{aligned}$$

where $T(\eta_j)=T(\xi)=n$ and $\eta_j(t)=w_{k-j+1}\xi(t)=w_{k-j+1}x_t$. Finally, applying P5 of the Appendix we find:

$$\sigma_{k+1}=\Omega^{2k-1}\theta\iota+\Omega\theta\Omega^{k-1}\eta=\Omega^{2k-1}\theta[\iota+\eta]=\Omega^{2k-1}\theta\eta \tag{3.6}$$

where η is defined by

$$T(\eta)=n-(k-1),$$

$$\eta(t)=\sum_{j=1}^{k}\eta_j(t+k-j)=\sum_{j=1}^{k}w_{k-j+1}x_{t+k-j}=\sum_{q=1}^{k}w_q x_{t+q-1},\qquad 1\leqq t\leqq T(\eta).$$

In the last summation the index was changed to $q=k-j+1$ in order to provide for the same expression as in (1.1).

Evidently, (3.6) represents the output of the array in a clear and precise form; it indicates that after an initial period of $2k-1$ time units, the elements $\eta(t)=y_t$, $1\leqq t\leqq n-(k-1)$, will appear on the output link, each separated from the other by one time unit.

In the previous example we applied our technique to a homogeneous network. The technique is equally applicable to k-partially homogeneous networks if k is reasonably small. In that case, a system of difference equations is formed by writing the generic I/O description for a typical node from each homogeneous subset of interior nodes V_I^i, $i=1,2,\cdots,k$. The network I/O description is then obtained by solving this system of equations. The back substitution network and the sorting networks discussed in §§ 5 and 6 are examples of 2-partially homogeneous networks. The *LU* decomposition network described in [1] is a 4-partially homogeneous network that can be verified by the same technique.

Finally, we note that the explicit derivation of the network I/O description depends on our ability to solve the resulting system of difference equations. However, even if these equations cannot be solved explicitly, we may still verify the operation of the network if we have an idea about the network behavior and consequently about the sequences on the different edges of the graph. In fact, we need to show only that for the given input sequences, the expected sequences satisfy the system of difference equations. We demonstrate this procedure in § 6 by verifying the operation of a sorting network for which we could not solve the system of equations explicitly.

4. A band matrix multiplication network. In [1], Kung and Leiserson suggested a systolic network for the computation of the product of two band matrices $C=A*B$,

where both A and B have lower bandwidth k_1 and upper bandwidth k_2. In this section, we shall consider only the case $k_1 = k_2 = k$ and prove formally that the suggested network indeed produces the product matrix C. Moreover, the sequence notation used in the verification procedure will provide an accurate representation of the I/O data including the input required for proper operation and the timing of the output data.

In Fig. 8a we show the directed graph of the matrix multiplication network. The nodes of the graph are regularly laid out so that each node can be labeled by a pair (i, j) of integers, where i and j are the relative position of the node with respect to the two perpendicular axes shown in the figure. The set of colors C_E has three elements, namely p, r, and s, and the coloring function col$(\cdot)$ maps the edges directed to the southwest, southeast and north to the colors p, r and s, respectively.

The network is homogeneous; it consists of only one type of computational cell, namely the multiply/add type shown in Fig. 8b. Its generic I/O description is given by the causal relations:

(4.1a) $$\rho_{i,j-1} = \Omega \rho_{i,j},$$

(4.1b) $$\pi_{i-1,j} = \Omega \pi_{i,j},$$

(4.1c) $$\sigma_{i+1,j+1} = \Omega[\sigma_{i,j} + \rho_{i,j} * \pi_{i,j}].$$

In line with the definition of homogeneous networks, this description is valid for any cell (i, j), $-k \leqq i, j \leqq k$.

As an illustration of the network topology and its different data streams, we show in Fig. 8c the general network for the special case $k = 1$, that is, for the case of two tridiagonal matrices A and B. In the figure, the source/sink cells were omitted for clarity.

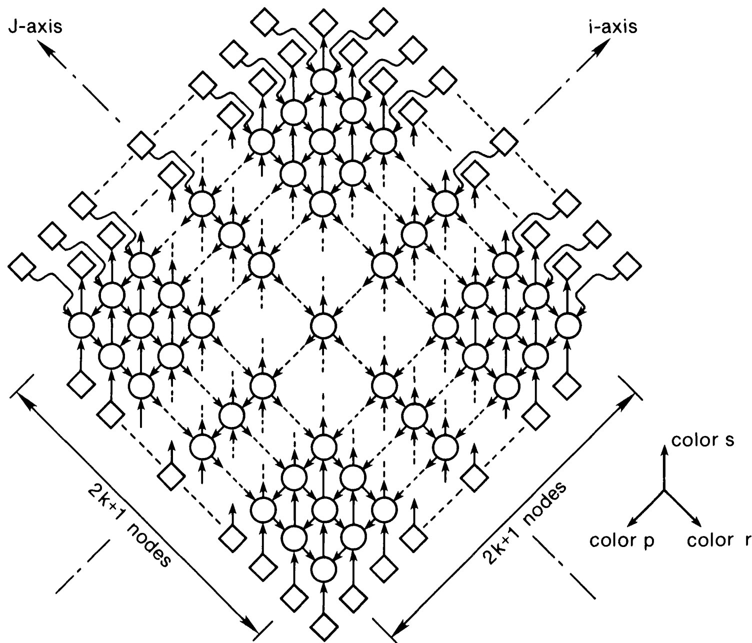


FIG. 8a

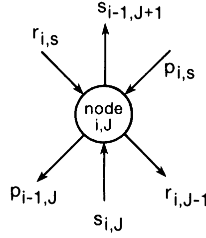


FIG. 8b

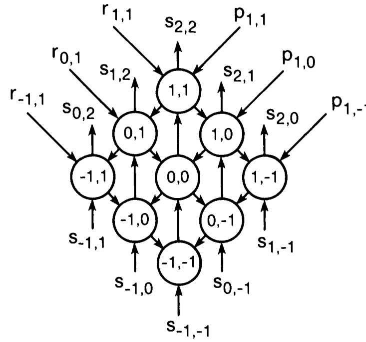


FIG. 8c

In order to obtain the I/O description of the network, we have to solve the system of difference equations (4.1), and express the network output sequences $\sigma_{q,k+1}$ and $\sigma_{k+1,q}$, $-(k-1)\leqq q\leqq k+1$ in terms of the network input sequences $\rho_{u,k}$, $\pi_{k,u}$, $\sigma_{-k,u}$ and $\sigma_{u,-k}$, $-k\leqq u\leqq k$. For this, consider first the simple equations (4.1a) and (4.1b) which have the solutions

$$\rho_{i,j}=\Omega^{k-j}\rho_{i,k},\qquad \pi_{i,j}=\Omega^{k-i}\pi_{k,j}.$$

By substituting these values into (4.1c) we obtain

$$\sigma_{i+1,j+1}=\Omega[\sigma_{i,j}+\Delta_{i,j}] \tag{4.2}$$

where $\Delta_{i,j}=\Omega^{k-j}\rho_{i,k}*\Omega^{k-i}\pi_{k,j}$.

By an inductive argument similar to the one given in the Appendix for Lemma 1, it is easily shown that for $-(k-1)\leqq i,\ j\leqq k+1$, (4.2) has the solution:

$$\sigma_{i,j}=\begin{cases}\Omega^{i+k}\sigma_{-k,j-i-k}+\sum_{q=1}^{k+i}\Omega^{q}\Delta_{i-q,j-q}, & i\leqq j,\\ \Omega^{j+k}\sigma_{i-j-k,-k}+\sum_{q=1}^{k+j}\Omega^{q}\Delta_{i-q,j-q}, & i>j.\end{cases}$$

With the definition of $\Delta_{i,j}$ and properties P1 and P4 we find the network output sequences to be

$$\sigma_{i,k+1}=\Omega^{i+k}\sigma_{-k,1-i}+\sum_{q=1}^{i+k}[\Omega^{2q-1}\rho_{i-q,k}*\Omega^{2q+k-i}\pi_{k,k-q+1}] \tag{4.3a}$$

$$\text{if } -(k-1)\leqq i\leqq k+1,$$

$$\sigma_{k+1,j} = \Omega^{k+j}\sigma_{1-j,-k} + \sum_{q=1}^{k+j} [\Omega^{2q+k-j}\rho_{k-q+1,k} * \Omega^{2q-1}\pi_{k,j-q}] \tag{4.3b}$$

$$\text{if } -(k-1) \leqq j \leqq k.$$

These are the network I/O descriptions. Of course, the network is not expected to produce the elements of the product matrix C unless the elements of the matrices $A = \{a_{i,j}\}$ and $B = \{b_{i,j}\}$ are fed into the proper input links of the network with the right timing. We will now prove that the network output sequences will contain the elements of C if the input sequences are specified as follows:

$$\rho_{u,k} = \Omega^{2(k+u)}\theta^2\alpha_u, \qquad -k \leqq u \leqq k, \tag{4.4a}$$

$$\pi_{k,u} = \Omega^{2(k+u)}\theta^2\beta_u, \qquad -k \leqq u \leqq k, \tag{4.4b}$$

$$\sigma_{-k,u} = \Omega^{2(2k+u)}\theta^2\iota_u, \qquad -k \leqq u \leqq k, \tag{4.4c}$$

$$\sigma_{u,-k} = \Omega^{2(2k+u)}\theta^2\iota_u, \qquad -k < u \leqq k, \tag{4.4d}$$

where

$$T(\beta_u) = T(\alpha_u) = n, \quad T(\iota_u) = n - (k+u), \quad \iota_u(t) = 0$$

and the sequences β_u, α_u are defined as follows:

For $u < 0$,

$$\alpha_u(t) = \begin{cases} 0 & \text{if } 1 \leqq t \leqq -u, \\ a_{t,t+u} & \text{if } -u < t \leqq n, \end{cases} \tag{4.5a}$$

$$\beta_u(t) = \begin{cases} 0 & \text{if } 1 \leqq t \leqq -u, \\ b_{t+u,t} & \text{if } -u < t \leqq n. \end{cases} \tag{4.5b}$$

For $u \geqq 0$,

$$\alpha_u(t) = \begin{cases} a_{t,t+u} & \text{if } 1 \leqq t \leqq n-u, \\ 0 & \text{if } n-u < t \leqq n, \end{cases} \tag{4.5c}$$

$$\beta_u = (t) = \begin{cases} b_{t+u,t} & \text{if } 1 \leqq t \leqq n-u, \\ 0 & \text{if } n-u < t \leqq n. \end{cases} \tag{4.5d}$$

Roughly speaking, the input link $p_{k,u}$, $-k \leqq u \leqq k$, contains the uth off-diagonal of the matrix B, while the input link $r_{u,k}$, $-k \leqq u \leqq k$, contains the $(-u)$th off-diagonal of the matrix A. Of course the exact timing of the input data is defined by the formulas (4.4).

For the sake of brevity, we consider here only (4.3a) and show that the output links $s_{i,k+1}$, $-(k-1) \leqq i \leqq k+1$, will carry the elements in the lower band of the product matrix $C = A * B$, including the diagonal. By a similar procedure, one can use (4.3b) to show that the links $s_{k+1,j}$, $-(k-1) \leqq j \leqq k$, will carry the upper band of C.

By introducing the specifications (4.4) of the network input sequences into (4.3a), we obtain for $-(k-1) \leqq i \leqq k+1$ the following formula:

$$\begin{aligned} \sigma_{i,k+1} &= \bar{\iota}_i + \sum_{q=1}^{k+i} [\Omega^{2k+2i-1}\theta^2\alpha_{i-q} * \Omega^{5k-i+2}\theta^2\beta_{k-q+1}] \\ &= \bar{\iota}_i + \Omega^{2k+2i-1} \sum_{q=1}^{k+i} [\theta^2\alpha_{i-q} * \Omega^{3(k-i+1)}\theta^2\beta_{k-q+1}] \\ &= \bar{\iota}_i + \Omega^{2k+2i-1}\theta^2 \sum_{q=1}^{k+i} [\alpha_{i-q} * \Omega^{k-i+1}\beta_{k-q+1}] \end{aligned}$$

where $\bar{\iota}_i = \Omega^{5k-i+2}\theta^2\iota_{1-i}$. With property P7 the product term becomes

$$\sigma_{i,k+1} = \bar{\iota}_i + \Omega^{2k+2i-1}\theta^2 \sum_{q=1}^{k+i} \Omega^{k-i+1}\gamma_i^q \tag{4.6}$$

where $T(\gamma_i^q) = n-(k-i+1)$ and

$$\gamma_i^q(t) = \alpha_{i-q}(t+k-i+1) * \beta_{k-q+1}(t).$$

Simplifying (4.6) and using the definition of $\bar{\iota}_i$, we find that

$$\sigma_{i,k+1} = \Omega^{5k-i+2}\theta^2\iota_{1-i} + \Omega^{5k-i+2}\theta^2 \sum_{q=1}^{k+i} \gamma_i^q = \Omega^{5k-i+2}\theta^2[\iota_{1-i} + \eta_i]$$

where $T(\eta_i) = n-(k-i+1)$ and

$$\eta_i(t) = \sum_{q=1}^{k+i} \gamma_i^q(t) = \sum_{q=1}^{k+i} \alpha_{i-q}(t+k-i+1) * \beta_{k-q+1}(t), \qquad -(k-1) \leqq i \leqq k+1. \tag{4.7}$$

Finally, from the definition of ι_{1-i} we obtain that

$$\sigma_{i,k+1} = \Omega^{5k-i+2}\theta^2\eta_i, \qquad -(k-1) \leqq i \leqq k+1. \tag{4.8}$$

(4.8) describes the timing of the output data on any link $s_{i,k+1}$, $-(k-1) \leqq i \leqq k+1$. It indicates that on $s_{i,k+1}$, there will be an initial setup time of $5k-i+2$ units, after which the elements $\eta_i(t)$, $t = 1, 2, \cdots, n-(k-i+1)$, will appear separated each from the other by two time units. We still need to show that $\eta_i(t) = c_{t+k-i+1,t}$, that is, that $s_{i,k+1}$ carries the $(k-i+1)$st subdiagonal of the matrix C.

To evaluate $\eta_i(t)$ from (4.7), we use the definitions (4.5) to write $\alpha_{i-q}(t+k-i+1)$ and $\beta_{k-q+1}(t)$ for the values of t between 1 and $n-(k-i+1)$, which are the values of t assumed in (4.7). The resulting formulas are

$$\alpha_u(t+d) = \begin{cases} 0 & \text{if } u<0 \text{ and } 1 \leqq t \leqq q-(k+1), \\ a(t,i,q) & \text{if } u<0 \text{ and } q-(k+1) < t \leqq n-d, \\ a(t,i,q) & \text{if } u \geqq 0 \text{ and } 1 \leqq t \leqq n+q-(k+1), \\ 0 & \text{if } u \geqq 0 \text{ and } n+q-(k+1) < t \leqq n-d; \end{cases} \tag{4.9a}$$

$$\beta_v(t) = \begin{cases} 0 & \text{if } v<0 \text{ and } 1 \leqq t \leqq q-(k+1), \\ b(t,q) & \text{if } v<0 \text{ and } q-(k+1) < t \leqq n-d, \\ b(t,q) & \text{if } v \geqq 0 \text{ and } 1 \leqq t \leqq n+q-(k+1), \\ 0 & \text{if } v \geqq 0 \text{ and } n+q-(k+1) < t \leqq n-d \end{cases} \tag{4.9b}$$

where, for simplicity, we introduced the notation $u = i-q$, $v = k-q+1$, $d = (k+1)-i$, $a(t,i,q) = a_{t+d,t+d+u}$ and $b(t,q) = b_{t+v,t}$, which will be used repeatedly in the remainder of this section.

It is clear from (4.9) that the evaluation of $\eta_i(t)$ by (4.7) is nontrivial and depends on the relative values of i and q. For this purpose, we consider two different cases.

Case 1. *If* $-(k-1) \leqq i \leqq 0$. In this case and for $1 \leqq q \leqq k+i$, the inequalities $u = i-q < 0$ and $v = k-q+1 \geqq 0$ always hold. Moreover, we have $q-(k+1) \leqq 0$ and $n+q-(k+1) > n-d$. Accordingly, we can use the above conditions to determine the appropriate values of $\alpha_u(t+d)$ and $\beta_v(t)$ from (4.9), and with these in (4.7) we obtain

$$\eta_i(t) = \sum_{q=1}^{k+i} a_{t+d,t+k+1-q} b_{t+k+1-q,t}, \qquad 1 \leqq t \leqq n-d.$$

By changing the summation index to $j=t+k+1-q$ this is indeed

$$(4.10)\qquad \eta_i(t)=\sum_{j=t+d-k}^{t+k} a_{t+d,j}b_{j,t}, \qquad 1\leqq t\leqq n-d.$$

Case 2. *If* $1\leqq i\leqq k+1$. In this case we always have $u=i-q\leqq v=k-q+1$. Accordingly, we divide the sum in (4.7) into the three partial sums

$$\sum_{q=1}^{k+i}=\sum_{q=1}^{i}+\sum_{q=i+1}^{k}+\sum_{q=k+1}^{k+i}.$$

For simplicity, we refer to these three sums as Σ_1, Σ_2 and Σ_3, respectively, and evaluate them separately.

In the case of $\Sigma_1=\sum_{q=1}^{i}\gamma_i^q(t)$, we note that the condition $1\leqq q\leqq i$ implies that $v\geqq u\geqq 0$. Hence, by (4.9) we have

$$\gamma_i^q(t)=\begin{cases} a(t,i,q)b(t,q) & \text{if } 1\leqq t\leqq n+q-(k+1),\\ 0 & \text{if } n+q-(k+1)<t\leqq n-d.\end{cases}$$

By standard rules of operations with summation symbols, Σ_1 can be expressed as

$$(4.11)\qquad \Sigma_1=\begin{cases} \displaystyle\sum_{q=1}^{i} a(t,i,q)b(t,q) & \text{if } 1\leqq t\leqq n-k,\\ \displaystyle\sum_{q=t-n+k+1}^{i} a(t,i,q)b(t,q) & \text{if } n-k<t\leqq n-d.\end{cases}$$

We turn next to $\Sigma_2=\sum_{q=i+1}^{k}\gamma_i^q(t)$. In this case, we have $u<0\leqq v$, $q-(k+1)<0$ and $n+q-(k+1)>n-d$. Hence, from (4.9) it follows that

$$\gamma_i^q(t)=a(t,i,q)b(t,q), \qquad 1\leqq t\leqq n-d,$$

which gives directly

$$(4.12)\qquad \Sigma_2=\sum_{q=i+1}^{k} a(t,i,q)b(t,q), \qquad 1\leqq t\leqq n-d.$$

Finally, in the case of Σ_3 the inequality $u\leqq v<0$ holds. Therefore, we have

$$\gamma_i^q(t)=\begin{cases} 0 & \text{if } 1\leqq t\leqq q-(k+1),\\ a(t,i,q)b(t,q) & \text{if } q-(k+1)<t\leqq n-d,\end{cases}$$

which gives

$$(4.13)\qquad \Sigma_3=\begin{cases} \displaystyle\sum_{q=k+1}^{k+t} a(t,i,q)b(t,q) & \text{if } 1\leqq t\leqq i,\\ \displaystyle\sum_{q=k+1}^{k+i} a(t,i,q)b(t,q) & \text{if } i<t\leqq n-d.\end{cases}$$

Now $\eta_i(t)$ is obtained by adding the sums (4.11), (4.12) and (4.13) on three different intervals for t. This sum is given by

$$\eta_i(t)=\begin{cases} \displaystyle\sum_{q=1}^{k+t} a(t,i,q)b(t,q), & 1\leqq t\leqq i,\\ \displaystyle\sum_{q=1}^{k+i} a(t,i,q)b(t,q), & i<t\leqq n-k,\\ \displaystyle\sum_{q=t-n+k+1}^{k+i} a(t,i,q)b(t,q), & n-k<t\leqq n-d.\end{cases}$$

By changing the summation index to $j=t+k+1-q$ and substituting the appropriate values for $a(t, i, q)$ and $b(t, q)$, we obtain

$$\eta_i(t)=\begin{cases}\sum_{j=1}^{t+k} a_{t+d,j}b_{j,t}, & 1\leqq t\leqq i,\\ \sum_{j=t+d-k}^{t+k} a_{t+d,j}b_{j,t}, & i<t\leqq n-k,\\ \sum_{j=t+d-k}^{n} a_{t+d,j}b_{j,t}, & n-k<t\leqq n-d.\end{cases} \tag{4.14}$$

Note that (4.14) is valid for $1\leqq i\leqq k+1$ while (4.10) is valid for $-(k-1)\leqq i\leqq 0$. These two formulas are equivalent to those resulting from multiplying the two band matrices A and B, which proves that for $t=1,2,\cdots,n-(k-i+1)$ and $-(k-1)\leqq i\leqq k+1$, we have indeed

$$\eta_i(t)=c_{t+d,t}=c_{t+k-i+1,t}.$$

5. A back substitution network. In this section, we apply our verification technique to a systolic network that contains two different types of computational cells, namely the back substitution network suggested in [16]. This network performs the back substitution operation to solve the linear system of equations

$$Lu=y \tag{5.1}$$

where L is an $n\times n$ nonsingular, banded, lower triangular matrix with the band width $k+1$, and y is a given n-dimensional vector. The solution of the system (5.1) is given by

$$u_i=\begin{cases}\dfrac{y_i}{l_{i,i}}, & i=1,\\ \dfrac{y_i-\sum_{j=1}^{i-1} l_{i,i-j}u_{i-j}}{l_{i,i}}, & 2\leqq i\leqq k,\\ \dfrac{y_i-\sum_{j=1}^{k} l_{i,i-j}u_{i-j}}{l_{i,i}}, & k<i\leqq n,\end{cases}$$

where $l_{i,j}$ is the (i,j)th element of the matrix L, and y_i and u_i are the ith elements of the vectors y and u, respectively.

Figure 9 shows the graph of the suggested network. It is a 2-partially homogeneous network, composed of k multiply-add (M/A) type cells, and one subtract-divide (S/D) cell. The computational cells are labeled by integers such that the cells 1 through k are of the M/A type, and the cell 0 is the S/D cell. As for the I/O cells, we must be

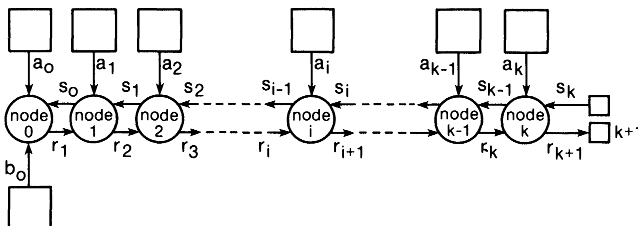


FIG. 9

careful to assign labels to the sink cells because these labels will be used to identify the network output links. The labels given to source nodes are immaterial, as they do not affect the verification procedure, and consequently they are not shown in Fig. 9.

In the regular layout shown in Fig. 9, the edges directed to the south, north, east and west are given the colors a, b, r and s, respectively. The set V_I of interior nodes in G is divided into two homogeneous subsets $V_I^1=\{0\}$ and $V_I^2=\{i: i=1,2,\cdots,k\}$. The operation of the cell represented by node 0 is described by the causal relation

$$\rho_1=\Omega[[\beta_0-\sigma_0]\div\alpha_0] \tag{5.2}$$

and the operation of any M/A cell represented by a node i, $1\leqq i\leqq k$, is described by the generic I/O description

$$\rho_{i+1}=\Omega\rho_i, \qquad i=1,2,\cdots,k, \tag{5.3a}$$

$$\sigma_{i-1}=\Omega[\sigma_i\oplus\alpha_i*\rho_i], \qquad i=1,2,\cdots,k, \tag{5.3b}$$

where the $\oplus$ was defined by (2.1).

To solve the system of difference equations (5.2), (5.3a, b), we first write the solution of (5.3a) as

$$\rho_i=\Omega^{i-1}\rho_1, \qquad 1<i\leqq k+1, \tag{5.4}$$

from which we find that

$$\rho_{k+1}=\Omega^k\rho_1. \tag{5.5}$$

Substitution of (5.4) into (5.3b) then gives

$$\sigma_{i-1}=\Omega[\sigma_i\oplus\Delta_i] \tag{5.6}$$

where $\Delta_i=\alpha_i*(\Omega^{i-1}\rho_1)$. Using an inductive argument similar to that in the Appendix for the proof of Lemma 1, we can show that the solution of (5.6) is

$$\sigma_0=\Omega^k\sigma_k\oplus\sum_{j=1}^{k}{}'\,\Omega^j[\alpha_j*\Omega^{j-1}\rho_1] \tag{5.7}$$

where $\sum'$ is defined by $\sum_{j=1}^{\prime k}\eta_j=\eta_1\oplus\eta_2\oplus\cdots\oplus\eta_k$.

For given ρ_1, the network output sequence ρ_{k+1} is easily obtained from (5.5). The next step will be to eliminate σ_0 from (5.2) and (5.7) and to obtain ρ_1 explicitly in terms of the network input sequences σ_k, β_0 and α_j, $j=0,1,\cdots,k$. Unfortunately, if we try to solve (5.2) and (5.7) simultaneously, we will obtain a recursive equation in ρ_1, which is very difficult to manipulate in general. For this reason, we consider only specific forms of the network input sequences, namely those required for the proper operation of the network. They are given by

$$\alpha_i=\Omega^{k+i}\theta\lambda_i, \qquad i=0,1,\cdots,k, \tag{5.8a}$$

$$\beta_0=\Omega^k\theta\eta, \tag{5.8b}$$

$$\sigma_k=\theta\iota, \tag{5.8c}$$

with $T(\lambda_i)=n-i$, $T(\iota)=T(\eta)=n$ and

$$\lambda_i(t)=l_{t+i,t}, \qquad 1\leqq t\leqq n-i,$$

$$\eta(t)=y_t, \qquad 1\leqq t\leqq n,$$

$$\iota(t)=0, \qquad 1\leqq t\leqq n.$$

Substituting (5.8) into (5.2) and (5.7), we find that

$$\rho_1 = \Omega[[\Omega^k\theta\eta - \sigma_0] \div \Omega^k\theta\lambda_0], \tag{5.9a}$$

$$\sigma_0 = \Omega^k\theta\iota \oplus \sum_{j=1}^{k}{}' [\Omega^{2j+k}\theta\lambda_j * \Omega^{2j-1}\rho_1]. \tag{5.9b}$$

Since $\delta - x = \delta$ for any $x \in R_\delta$, (5.9a) implies the existence of a sequence ξ such that

$$\rho_1 = \Omega^{k+1}\theta\xi \tag{5.10}$$

whence, by (5.9b), we find that

$$\sigma_0 = \Omega^k\theta\left[\iota \oplus \sum_{j=1}^{k}{}' \Omega^j[\lambda_j * \xi]\right]$$

where we used property P2 to interchange Ω^{2j} and θ. If in addition we let

$$\gamma = \iota \oplus \sum_{j=1}^{k}{}' \Omega^j[\lambda_j * \xi], \tag{5.11}$$

then we can substitute for σ_0 and ρ_1 in (5.9a) and obtain

$$\Omega^{k+1}\theta\xi = \Omega[[\Omega^k\theta\eta - \Omega^k\theta\gamma] \div \Omega^k\theta\lambda_0],$$

which reduces to

$$\xi = [\eta - \gamma] \div \lambda_0. \tag{5.12}$$

For an explicit description of the sequence γ, we need to examine (5.11) more closely. We start by evaluating the product term, namely

$$\Omega^j[\lambda_j * \xi] = \Omega^j\mu_j$$

where

$$T(\mu_j) = \min\{T(\lambda_j), T(\xi)\} \leq n - j \tag{5.13a}$$

and

$$\mu_j(t) = \lambda_j(t) * \xi(t). \tag{5.13b}$$

This enables us to rewrite (5.11) as

$$\gamma = \iota \oplus \sum_{j=1}^{k}{}' \Omega^j\mu_j. \tag{5.14}$$

From (5.14) and the definition of the $\oplus$ operator, we conclude that $T(\gamma) = \max\{T(\iota), T(\mu_j) + j\} = n$, and consequently from (5.12) that

$$T(\xi) = \min\{T(\eta), T(\gamma), T(\lambda_0)\} = n.$$

Using this in (5.13a) we easily see that $T(\mu_j) = n - j$. Now, we apply property P6 to (5.14) and explicitly describe γ by

$$T(\gamma) = T(\iota) = n$$

and

$$\gamma(t)=\begin{cases}0, & t=1,\\ \sum_{j=1}^{t-1}\mu_j(t-j), & t=2,3,\cdots,k,\\ \sum_{j=1}^{k}\mu_j(t-j), & t=k+1,k+2,\cdots,n.\end{cases}$$

Finally, with these specific descriptions of η, λ_0 and γ, we directly find the explicit form of the sequence ξ in (5.12) to be

$$\xi(t)=\frac{\eta(t)-\gamma(t)}{\lambda_0(t)};$$

that is,

$$\xi(t)=\begin{cases}\dfrac{y_t}{l_{t,t}}, & t=1,\\ \dfrac{y_t-\sum_{j=1}^{k}\xi(t-j)l_{t,t-j}}{l_{t,t}}, & 2\leqq t\leqq k,\\ \dfrac{y_t-\sum_{j=1}^{k}\xi(t-j)l_{t,t-j}}{l_{t,t}}, & k+1\leqq t\leqq n.\end{cases}$$

A comparison of this expression with the formula given in the beginning of the section for the solution of (5.1) shows readily that

$$\rho_{k+1}=\Omega^{2k+1}\theta\xi$$

where $T(\xi)=n$ and $\xi(t)=u_t$.

6. A sorting network. The sorting network [17] described here accepts an indexed set $X=\{x_1,\cdots,x_k\}$ of k different real numbers, $x_i\in R$, $i\in K=\{1,\cdots,k\}$, and produces as output the same numbers sorted in ascending order. Figure 10 shows the general graph of the network and the labels given to each node. In the figure, the edges directed to the right and left are colored p and s, respectively.

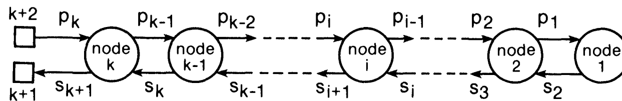


FIG. 10

For any $j\in K$, let $y_1,\cdots,y_j$ be the result of sorting the j elements $x_1,\cdots,x_j$ of X in ascending order. Then for all (i,j) of $D=\{(i,j)\in K\times K;1\leqq i\leqq j\leqq k\}$, the ranking function $f_x\colon D\to X$ is defined by $f_x(i,j)=y_i$.

With this, we will prove that if the network input sequence π_k is given by

(6.1) $$\pi_k=\theta\xi$$

where $T(\xi)=k$ and $\xi(t)=x_t$, then the network output sequence σ_{k+1} has the form

(6.2) $$\sigma_{k+1}=\Omega^{2k-1}\theta\eta$$

where $T(\eta)=k$ and $\eta(t)=f_x(t,k)$.

The network considered in Fig. 10 is a 2-partially homogeneous network. The cell labeled 1 is a simple latch cell whose operation is described by

$$\sigma_2 = \Omega \pi_1 \tag{6.3a}$$

while the I/O description of the cells $i = 2, \cdots, k$ is given by

$$\pi_{i-1} = \Omega \max_\delta (\pi_i, \sigma_i), \tag{6.3b}$$

$$\sigma_{i+1} = \Omega \min_\delta (\pi_i, \sigma_i) \tag{6.3c}$$

where $\max_\delta$ and $\min_\delta$ were defined in § 2.1. In other words, the cells $i = 2, \cdots, k$ are comparison cells which operate as follows: At any time t, if neither one of the two inputs $\sigma_i(t)$ or $\pi_i(t)$ is a "don't care element" δ, then the cell compares the two inputs, and produces as output at time $t+1$, the largest and the smallest numbers on the links p_{i-1} and s_{i+1} respectively. However, if any of the inputs is δ, then the cell acts as a simple latch cell; that is, if $\sigma_i(t) = \delta$ or $\pi_i(t) = \delta$ then

$$\pi_{i-1}(t+1) = \pi_i(t) \quad \text{and} \quad \sigma_{i+1}(t+1) = \sigma_i(t).$$

To obtain the network I/O description, the system of equations (6.3a, b, c) should be solved for σ_{k+1}. However, the recursive nature of (6.3b) and (6.3c) makes this very difficult, if not impossible. One alternative is to suggest a tentative value for the sequences π_i and σ_i, and then to verify that these suggested solutions indeed satisfy (6.3). Of course, any assumed value for π_i should reduce to the input sequence (6.1) for $i = k$.

Let us assume that π_i and σ_i are given by

$$\pi_i = \Omega^{k-i} \theta \alpha_i, \qquad 1 \leqq i \leqq k, \tag{6.4a}$$

$$\sigma_i = \Omega^{k+i-2} \theta \beta_i, \qquad 2 \leqq i \leqq k+1, \tag{6.4b}$$

where $T(\alpha_i) = T(\beta_i) = k$,

$$\alpha_i(t) = \begin{cases} x_t, & 1 \leqq t \leqq i, \\ \max \{x_t, f_x(t-i, t-1)\}, & i < t \leqq k, \end{cases}$$

and

$$\beta_i(t) = \begin{cases} f_x(t, t+i-2), & 1 \leqq t \leqq k+1-i, \\ f_x(t, k), & k+1-i < t \leqq k. \end{cases}$$

It is very easy to verify that (6.4a) reduces to (6.1) for $i = k$. Hence, our next step will be to check that (6.4) does satisfy (6.3). For $i = 1$, (6.4a) reduces to

$$\pi_1 = \Omega^{k-1} \theta \alpha_1$$

where $T(\alpha_1) = k$, and

$$\alpha_1(t) = \begin{cases} x_t, & t = 1, \\ \max_\delta \{x_t, f_x(t-1, t-1)\}, & 1 < t \leqq k. \end{cases}$$

Since $f_x(j, j)$ is the maximum element in $\{x_1, x_2, \cdots, x_j\}$, it follows that $x_1 = f_x(1, 1)$ and $\max_\delta \{x_t, f_x(t-1, t-1)\} = f_x(t, t)$. Hence, we may write

$$\alpha_1(t) = f_x(t, t), \qquad 1 \leqq t \leqq k.$$

But from (6.4b), we obtain for $i = 2$

$$\sigma_2 = \Omega^k \theta \beta_2$$

where $T(\beta_2)=k$ and $\beta_2(t)=f_x(t,t)$, $1\leqq t\leqq k$, which proves that $\beta_2=\alpha_1$, and hence $\sigma_2=\Omega\pi_1$.

The next step is to show that (6.4) does satisfy (6.3b). For this, we substitute (6.4) into the right-hand side of (6.3b) and denote the resulting sequence by ρ. This gives

$$\rho=\Omega \max_\delta (\Omega^{k-i}\theta\alpha_i, \Omega^{k+i-2}\theta\beta_i), \qquad 2\leqq i\leqq k.$$

Using property P2 to interchange $\Omega^{2(i-1)}$ and θ in the second operand of $\max_\delta$, we obtain

$$\rho=\Omega^{k-(i-1)}\theta\gamma_i \tag{6.5}$$

where $\gamma_i=\max_\delta\{\alpha_i, \Omega^{i-1}\beta_i\}$. By definition of $\max_\delta$, it follows that $T(\gamma_i)=T(\alpha_i)=k$, and

$$\gamma_i(t)=\begin{cases}\alpha_i(t), & 1\leqq t\leqq i-1,\\ \max\{\alpha_i(t),\beta_i(t-i+1)\}, & i-1<t\leqq k.\end{cases}$$

Hence with the definition of $\alpha_i(t)$ and $\beta_i(t)$ we obtain

$$\gamma_i(t)=\begin{cases}x_t, & 1\leqq t\leqq i-1,\\ \max\{x_t, f_x(t-i+1,t-1)\}, & t=i,\\ \max\{\max\{x_t, f_x(t-i,t-1)\}, f_x(t-i+1,t-1)\}, & i<t\leqq k.\end{cases}$$

Because $\max\{\max\{a,b\},c\}=\max\{a,b,c\}$, and $f_x(t-i,t-1)<f_x(t-i+1,t-1)$, we may rewrite γ_i as

$$\gamma_i(t)=\begin{cases}x_t, & 1\leqq t\leqq i-1,\\ \max\{x_t, f_x(t-(i-1),t-1)\}, & i-1<t\leqq k,\end{cases}$$

from which we find that $\gamma_i(t)=\alpha_{i-1}(t)$, and hence, by (6.5) and (6.4a), that $\rho=\pi_{i-1}$. This proves that (6.3b) is satisfied for the values of σ_i and π_i given by (6.4).

Finally, to check that (6.4) does satisfy (6.3c), we substitute (6.4) into (6.3c) and denote the resulting sequence by τ. This gives

$$\tau=\Omega \min_\delta\{\Omega^{k-i}\theta\alpha_i, \Omega^{k+i-2}\theta\beta_i\}=\Omega^{k-i+1}\theta \min_\delta\{\alpha_i, \Omega^{i-1}\beta_i\}, \qquad 2\leqq i\leqq k.$$

In view of

$$\min_\delta\{\alpha_i, \Omega^{i-1}\beta_i\}=\Omega^{i-1}\varphi_i$$

where $T(\varphi_i)=T(\beta_i)=k$ and

$$\varphi_i(t)=\begin{cases}\min\{\alpha_i(t+i-1),\beta_i(t)\}, & 1\leqq t\leqq k-(i-1),\\ \beta_i(t), & k-(i-1)<t\leqq k,\end{cases}$$

we write

$$\tau=\Omega^{k+(i+1)-2}\theta\varphi_i. \tag{6.6}$$

From (6.6) and (6.3c), it follows that $\tau=\sigma_{i+1}$ only if $\varphi_i=\beta_{i+1}$. To prove this, we substitute the definitions of $\alpha_i(t+i-1)$ and $\beta_i(t)$ into $\varphi_i(t)$ and obtain

$$\varphi_i(t)=\begin{cases}\min\{\max\{x_{t+i-1}, f_x(t-1,t+i-2)\}, f_x(t,t+i-2)\}, & 1\leqq t\leqq k-(i-1),\\ f_x(t,k), & k-(i-1)<t\leqq k.\end{cases}$$

But from Lemma 2 in the Appendix, and the fact that $f_x(t, t+i-1) = f_x(t, k)$ for $t = k-i+1$, we may write $\varphi_i(t)$ as

$$\varphi_i(t) = \begin{cases} f_x(t, t+i-1), & 1 \leqq t \leqq k-i, \\ f_x(t, k), & k-i < t \leqq k. \end{cases}$$

It follows that $\varphi_i(t) = \beta_{i+1}(t)$ and therefore that $\tau = \sigma_{i+1}$. This completes the proof that the sequences π_i and σ_i of (6.4) indeed satisfy the system of equations (6.3).

Now that (6.4b) is known to be a valid formula for the sequence σ_i, we can easily obtain the network output sequence σ_{k+1} by setting $i = k+1$. This gives

$$\sigma_{k+1} = \Omega^{2k-1}\theta\beta_{k+1}$$

where $T(\beta_{k+1}) = k$ and $\beta_{k+1}(t) = f_x(t, k)$, $1 \leqq t \leqq k$, which is identical with the expected output sequence (6.2).

7. Concluding remarks. This work is meant to contribute to the area of systolic architectures in three different ways by providing, namely, a mathematical model for systolic networks, an unambiguous description of its input and output data, and a technique for the verification of its operation.

The central concepts in the present model are those of data sequences and sequence operators. Although we only defined the few operators that were used in the examples, it should be clear that other sequence operators may be introduced to model other types of computational cells.

Because of the nature of our examples, the non-δ entries in the data sequences were restricted to be members of the set R of real numbers. In a more general setting, R may be any set of items that can be transmitted on the communication links of the network, provided that the operators are defined appropriately.

A further step in this area is to develop a more complete sequence algebra to provide a basis for a solvability theory of the resulting system of difference equations on sequences. More specifically, it would be desirable to determine under which conditions an explicit analytical solution for the system of difference equations can be obtained. For a given network, this might determine the properties to be satisfied by the successor function μ and the node I/O operators in order to verify analytically the operation of the network. If a sufficiently flexible algebra of this type were available, our model might prove to be very powerful in the design of new systolic networks.

At this point, we note that even if we cannot solve the resulting system of equations analytically, we can still use a numerical iterative procedure to solve it. This approach is very close to the simulation of systolic networks, but appears to be more general and systematic [18].

Finally, we note that throughout this paper we assumed the systolic network to operate synchronously. However, the same model and techniques can be used for asynchronous networks. The only difference is in the interpretation of the ith element of a data sequence, which now has to denote the ith data item that appeared on a communication link instead of the data item that appeared on that link at time $t = i$.

Appendix. In the first part of this Appendix, we list some properties of sequence operators that have been used in the paper. The verification of these properties is straightforward from the definitions of the operators involved. In the second part of the Appendix, we prove two lemmas; the first gives an analytical solution to a difference equation that appears frequently in the verification of networks containing multiply/add cells, while the second one proves an equality that was needed in § 6.

Let ξ, ζ and η_j, $j=0,1,2,\cdots,k$, be sequences in $\bar{R}_\delta$, and $w \in R$. Then:
Property P1.

$$\Omega^r\Omega^k\xi=\Omega^{r+k}\xi.$$

Property P2.

$$\Omega^{(r+1)k}\theta^r\xi=\theta^r\Omega^k\xi.$$

Property P3.

$$w\cdot[\theta^k\xi]=\theta^k[w\cdot\xi], \qquad w\cdot[\Omega^r\xi]=\Omega^r[w\cdot\xi].$$

Property P4. For any binary operator "op" extended from R_δ to $\bar{R}_\delta$ with the property that δ "op" $\delta=\delta$, we have

$$\Omega^k[\xi \text{ "op" } \zeta]=\Omega^k\xi \text{ "op" } \Omega^k\zeta, \qquad \theta^r[\xi \text{ "op" } \zeta]=\theta^r\xi \text{ "op" } \theta^r\zeta.$$

Property P5. If η_j, $j=1,2,\cdots,k$, are such that $T(\eta_j)=n$, then

$$\sum_{j=1}^{k}\Omega^{j-1}\eta_j=\Omega^{k-1}\eta$$

where $T(\eta)=n-(k-1)$ and $\eta(t)=\sum_{j=1}^{k}\eta_j(t+k-j)$.

The next result uses the $\oplus$ of (2.1).

Property P6. Let the sequences η_j, $j=0,1,\cdots,k$, satisfy $T(\eta_j)=n-j$. Then

$$\eta_0\oplus\Omega\eta_1\oplus\Omega^2\eta_2\oplus\cdots\oplus\Omega^k\eta_k=\gamma$$

where $T(\gamma)=n$ and

$$\gamma(t)=\begin{cases}\sum_{j=0}^{t-1}\eta_j(t-j), & t=1,2,\cdots,k,\\ \sum_{j=0}^{k}\eta_j(t-j), & t=k+1,k+2,\cdots,n.\end{cases}$$

Property P7. Given $\xi,\zeta\in\bar{R}_\delta$. Then

$$\zeta*\Omega^r\xi=\Omega^r\gamma$$

where γ is described by

$$T(\gamma)=\min\{T(\zeta)-r, T(\xi)\} \quad \text{and} \quad \gamma(t)=\zeta(t+r)*\xi(t).$$

LEMMA 1. *The difference equation*

$$\sigma_{i+1}=\Omega\sigma_i+\Delta_i, \qquad i=1,2,\cdots,k+1, \tag{A.1}$$

has the solution

$$\sigma_r=\Omega^{r-1}\sigma_1+\sum_{j=1}^{r-1}\Omega^{j-1}\Delta_{r-j}, \qquad r=2,3,\cdots,k+1. \tag{A.2}$$

Proof. The proof uses induction on i. Evidently, for $i=1$ in (A.1) we obtain

$$\sigma_2=\Omega\sigma_1+\Delta_1$$

which is identical to (A.2) for $r=2$. Hence, assume that for any $r=1,2,\cdots,k$, σ_r is

given by (A.2); then from (A.1) it follows that

$$\sigma_{r+1} = \Omega\sigma_r + \Delta_r = \Omega\left[\Omega^{r-1}\sigma_1 + \sum_{j=1}^{r-1} \Omega^{j-1}\Delta_{r-j}\right] + \Delta_r$$

$$= \Omega^r\sigma_1 + \sum_{j=1}^{r-1} \Omega^j\Delta_{r-j} + \Delta_r$$

$$= \Omega^r\sigma_1 + \sum_{j=0}^{r-1} \Omega^j\Delta_{r-j}$$

$$= \Omega^r\sigma_1 + \sum_{j=1}^{r} \Omega^{j-1}\Delta_{r+1-j},$$

which proves that σ_{r+1} is also given by (A.2).

LEMMA 2. *Let f_x be the ranking function for the set $X = \{x_1, x_2, \cdots, x_n\}$, as defined in* § 6. *Then*

$$\min\{\max\{x_k, f_x(i-1, k-1)\}, f_x(i, k-1)\} = f_x(i, k). \tag{A.3}$$

Proof. Let $y_1, \cdots, y_{k-1}$ be the result of sorting $x_1, \cdots, x_{k-1}$ in ascending order, and let $z_1, \cdots, z_k$ be the corresponding result for $x_1, \cdots, x_k$. Hence, $f_x(i-1, k-1) = y_{i-1}$, $f_x(i, k-1) = y_i$ and $f_x(i, k) = z_i$. Now consider the following cases:

Case 1. If $x_k < y_{i-1} < y_i$ then the left side of (A.3) is

$$\min\{\max\{x_k, y_{i-1}\}, y_i\} = \min\{y_{i-1}, y_i\} = y_{i-1}.$$

Since $z_1, \cdots, z_k$ are obtained from $y_1, \cdots, y_{k-1}$ by inserting x_k in some position before y_{i-1}, we immediately see that $y_{i-1} = z_i$.

Case 2. If $y_{i-1} < x_k < y_i$, then the left side of (A.3) is

$$\min\{\max\{x_k, y_{i-1}\}, y_i\} = x_k,$$

and in this case it is clear that $x_k = z_i$.

Case 3. If $y_{i-1} < y_i < x_k$, then the left side of (A.3) is equal to y_i, which in turn is equal to z_i because, in this case, x_k is inserted in some position after y_i.

REFERENCES

[1] H. T. KUNG, *Why systolic architecture*, Comput. Mag., 15 (1982), pp. 37–46.

[2] M. OSSEFORT, *Correctness proofs of communicating processes—Three illustrative examples from the literature*, TR-LCS-8201, Dept. Computer Science, Univ. of Texas, Austin, Jan. 1982.

[3] M. J. FOSTER, *Syntax directed verification of circuit functions*, in VLSI Systems and Computations, H. T. Kung, B. Sproull and G. Steele, eds., Computer Science Press, Rockville, MD, 1981.

[4] J. MISRA AND K. M. CHANDI, *Proofs of networks of processes*, IEEE Trans. Software Engrg., SE7 (1981), pp. 417–426.

[5] D. COHEN, *Mathematical approach to iterative computational networks*, in Proc. of the Fourth Symposium on Computer Arithmetics, Oct. 1978, pp. 226–238.

[6] L. JOHNSSON, U. WEISER, D. COHEN AND A. DAVIS, *Toward a formal treatment of* VLSI *arrays*, Technical Report 4191, Dept. Computer Science, California Institute of Technology, Pasadena, CA, 1981.

[7] L. JOHNSSON AND D. COHEN, *A mathematical approach to modelling the flow of data and control in computational networks*, in VLSI Systems and Computations, H. T. Kung, B. Sproull and G. Steele, eds., Computer Science Press, Rockville, MD, 1981.

[8] U. WEISER AND A. DAVIS, *Mathematical representation for* VLSI *arrays*, Technical Report UUCS-80-111, Dept. of Computer Science, Univ. of Utah, Salt Lake City, Sept. 1980.

[9] ———, *A wavefront notation tool for* VLSI *array design*, in VLSI Systems and Computations, H. T. Kung, B. Sproull and G. Steele, eds., Computer Science Press, Rockville, MD, 1981.

[10] C. E. LEISERSON, *Systolic priority queues*, in Proc. Conference on VLSI: Architecture, Design, Fabrication, California Institute of Technology, Pasadena, CA, Jan. 1979.
[11] P. FAURRE AND M. DEPEYROT, *Elements of System Theory*, North-Holland, Amsterdam, 1977.
[12] J. GREFENSTETTE, *Automaton networks and parallel rewriting systems*, Ph.D. dissertation, Dept. Computer Science, Univ. of Pittsburgh, Pittsburgh, PA, 1980.
[13] J. VON NEUMANN, *Theory of Self Reproducing Automata*, Univ. of Illinois Press, Urbana, IL, 1966.
[14] A. W. BURKS, *Essays on Cellular Automata*, Univ. of Illinois Press, Urbana, IL, 1970.
[15] C. E. LEISERSON AND J. B. SAXE, *Optimizing synchronous systems*, in 22nd Annual ACM Symposium on the Foundations of Computer Science, Oct. 1981, pp. 23–36.
[16] H. T. KUNG AND C. E. LEISERSON in Systolic Arrays for VLSI, C. A. Mead and L. A. Conway, eds., Addison-Wesley, Reading, MA, 1980.
[17] H. T. KUNG, Private communication, 1981.
[18] R. G. MELHEM, *Simulation of systolic networks with a syntax directed solver for systems of sequence equations*, Techn. Rept. ICMA-83-58, Institute of Computational Mathematics and its Applications, Univ. Pittsburgh, 1983.

SIAM J. COMPUT.
Vol. 13, No. 3, August 1984

© 1984 Society for Industrial and Applied Mathematics
007

SIMPLE LINEAR-TIME ALGORITHMS TO TEST CHORDALITY OF GRAPHS, TEST ACYCLICITY OF HYPERGRAPHS, AND SELECTIVELY REDUCE ACYCLIC HYPERGRAPHS*

ROBERT E. TARJAN† AND MIHALIS YANNAKAKIS†

Abstract. Chordal graphs arise naturally in the study of Gaussian elimination on sparse symmetric matrices; acyclic hypergraphs arise in the study of relational data bases. Rose, Tarjan and Lueker [SIAM J. Comput., 5 (1976), pp. 266–283] have given a linear-time algorithm to test whether a graph is chordal, which Yannakakis has modified to test whether a hypergraph is acyclic. Here we develop a simplified linear-time test for graph chordality and hypergraph acyclicity. The test uses a new kind of graph (and hypergraph) search, which we call *maximum cardinality search*. A variant of the method gives a way to selectively reduce acyclic hypergraphs, which is needed for evaluating queries in acyclic relational data bases.

Key words. graph algorithm, acyclic data base scheme, sparse Gaussian elimination, graph search, hypergraph

1. Introduction. We shall use more-or-less standard terminology from the theory of graphs and hypergraphs [3], some of which we review here. A *hypergraph* $H=(V, E)$ consists of a set of *vertices* V and a set of *edges* E; each edge is a subset of V. A *graph* is a hypergraph all of whose edges have size two. The *graph* $G(H)$ of a hypergraph H is the graph whose vertices are those of H and whose edges are the vertex pairs $\{v, w\}$ such that v and w are in a common edge of H. Two vertices of a graph G are *adjacent* if they are contained in an edge. A *path* in G is a sequence of distinct vertices $v_0, v_1, \cdots, v_k$ such that v_i and v_{i+1} are adjacent for $0 \leqq i < k$. A *cycle* is a path $v_0, v_1, \cdots, v_k$ such that $k \geqq 2$ and v_0 and v_k are adjacent. Vertices v_i and $v_{(i+1) \bmod (k+1)}$ for $0 \leqq i \leqq k$ are *consecutive* on the cycle. A *clique* of G is a set of pairwise adjacent vertices. A hypergraph H is *conformal* if every clique of $G(H)$ is contained in an edge of H. A graph G is *chordal* if every cycle of length at least four has a chord, i.e., an edge joining two nonconsecutive vertices on the cycle. A hypergraph H is *acyclic* if H is conformal and $G(H)$ is chordal.

Chordal graphs arise in the study of Gaussian elmination on sparse symmetric matrices [12]. Acyclic hypergraphs arise in the study of relational data base schemes [1], [7], [21]; they are powerful enough to capture most real-world situations but simple enough to have many desirable properties [1], [2], [9], [18]. Rose, Tarjan and Lueker [15] have given an $O(n+m)$-time[1] algorithm (henceforth called the RTL algorithm) to test whether a graph is chordal. Yannakakis [19] has extended the algorithm to the problem of testing whether a hypergraph is acyclic. In this paper we propose a simplified version of the RTL algorithm that can be used for testing both chordality of graphs and acyclicity of hypergraphs. In § 2 we develop the algorithm as it applies to graph chordality testing. In § 3 we modify the algorithm for hypergraph acyclicity testing. Besides leading to a method simpler than the RTL test, our analysis provides additional insight into the structure of chordal graphs and acyclic hypergraphs. In § 4 we use this insight to develop a simple linear-time algorithm for selectively reducing acyclic hypergraphs, a problem that arises in evaluating queries in acyclic relational data bases.

* Received by the editors October 7, 1982, and in revised form May 23, 1983.

† Bell Laboratories, Murray Hill, New Jersey 07974.

[1] We shall use n to denote the number of vertices and m to denote the total size of the edges in a hypergraph.

2. Testing chordality of graphs. In this section we shall freely use results of [14], [15]. A discussion of chordal graphs and their importance in Gaussian elimination can be found in [14]. Let $G=(V, E)$ be a graph. Let $\alpha: V \leftrightarrow \{1, 2, \cdots, n\}$ be a numbering defining a total ordering of the vertices of G. We shall use the notation $v <_\alpha w$ to mean $\alpha(v) < \alpha(w)$. The *fill-in* produced by this ordering is the set

$$F(\alpha)=\{\{v, w\} \mid \{v, w\} \notin E \text{ and there is a path from } v \text{ to } w \text{ containing only } v, w, \text{ and vertices ordered before both } v \text{ and } w\}.$$

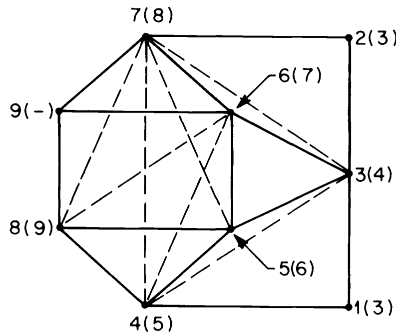


FIG. 1. *A graph numbered by maximum cardinality search and the corresponding elimination graph. Original edges are solid; fill-in edges are dashed. Followers of vertices (to be defined below) are in parentheses.*

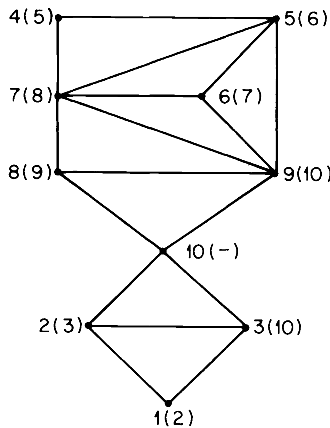


FIG. 2. *A chordal graph numbered by maximum cardinality search. Ordering is zero fill-in. Followers of vertices are in parentheses.*

The *elimination graph* of G with ordering α is $G(\alpha)=(V, E \cup F(\alpha))$. (See Fig. 1.) Note that $G(\alpha)$ is a subgraph of the transitive closure of G. If $F(\alpha)=\emptyset$, α is a *zero fill-in ordering* of G. (See Fig. 2.)

LEMMA 1 [15]. *A pair $\{v, w\}$ is in $E \cup F(\alpha)$ if and only if either $\{v, w\} \in E$ or there is a vertex u such that $\{u, v\}$, $\{u, w\} \in E \cup F(\alpha)$ and u is ordered before both v and w.*

LEMMA 2 [15]. *An ordering α is zero fill-in if and only if for all distinct $\{u, v\}$, $\{u, w\}$ in E such that u is ordered before both v and w, $\{v, w\} \in E$.*

LEMMA 3 [15]. *Any ordering α is a zero fill-in ordering of the corresponding graph $G(\alpha)$.*

THEOREM 1 [14]. *A graph G is chordal if and only if it has a zero fill-in ordering.*

Remark. Lemmas 1 and 2 and Theorem 1 are implicit in the work of Dirac [6] and Fulkerson and Gross [8] on chordal graphs, although they did not consider the notion of fill-in.

The RTL chordality-testing algorithm consists of two steps:

Step 1. Compute an ordering α of G that is zero fill-in if and only if G is chordal.
Step 2. Compute the fill-in produced by α. G is chordal if and only if $F(\alpha)=\emptyset$.

The algorithm we shall describe consists of simplified methods for carrying out steps one and two. Step 1 of the RTL method consists of numbering the vertices from n to 1 in decreasing order using *lexicographic search*, defined as follows: For each unnumbered vertex v, maintain a list of the numbers of the numbered vertices adjacent to v, with the numbers in each list arranged in decreasing order. As the next vertex to number, select the vertex whose list is lexicographically greatest, breaking ties arbitrarily. Although somewhat complicated, lexicographic search can be implemented to run in $O(n+m)$-time.

We shall derive a sufficient condition for an ordering of a chordal graph to be zero fill-in. This condition holds not only for lexicographic search but also for a simpler kind of search that we call *maximum cardinality search*: Number the vertices from n to 1 in decreasing order. As the next vertex to number, select the vertex adjacent to the largest number of previously numbered vertices, breaking ties arbitrarily. (See Figs. 1 and 2.)

LEMMA 4. *Let $G=(V, E)$ be a chordal graph and let α be an ordering of G. If α has the following property, then α is zero fill-in*:

(P) *If $u<_\alpha v<_\alpha w$, $\{u, w\}\in E$, and $\{v, w\}\notin E$, then there is a vertex x such that $v<_\alpha x$, $\{v, x\}\in E$, and $\{u, x\}\notin E$.*

Proof. Suppose α has property P. Let $v_0, v_1, \cdots, v_k$ be an unchorded path for which $\alpha(v_k)$ is maximum, such that $k\geqq 2$ and

(Q) For some i in the interval[2] $[1, k-1]$, the following inequalities hold:

$$v_0>_\alpha v_k>_\alpha v_1>_\alpha v_2>\cdots>_\alpha v_i \quad \text{and} \quad v_i<_\alpha v_{i+1}<\cdots<_\alpha v_k.$$

(A path is *unchorded* if any two nonconsecutive vertices are nonadjacent.) We shall derive a contradiction, thus showing that no unchorded path has property Q.

Since v_0 and v_1 are adjacent but not v_0 and v_k, there is by property P a vertex x such that $v_k<_\alpha x$ and v_k but not v_1 is adjacent to x. Let $j>1$ be minimum such that v_j is adjacent to x. Since G is chordal, x is not adjacent to v_0, for otherwise $v_0, v_1, \cdots, v_j$, x would be an unchorded cycle. Thus if $v_0>_\alpha x$, the path $v_0, v_1, \cdots v_j, x$ has property Q; if $x>_\alpha v_0$, the path $x, v_j, v_{j-1}, \cdots, v_0$ has property Q. Either case contradicts the choice of $v_0, v_1, \cdots, v_k$ as the path with $\alpha(v_k)$ maximum having property Q. Therefore no path has property Q.

Now suppose $\{u, v\}$ and $\{u, w\}$ are distinct edges such that u is ordered before both v and w by α. If v and w were not adjacent, either v, u, w or w, u, v would have property Q. Thus v and w must be adjacent. Since this holds for any such $\{u, v\}$ and $\{u, w\}$, it follows from Lemma 2 that α is zero fill-in. □

THEOREM 2. *Any ordering α generated by maximum cardinality search has Property* P *and thus is zero fill-in if G is chordal.*

Proof. Let α be an ordering generated by maximum cardinality search. Suppose $u<_\alpha v<_\alpha w$ and w is adjacent to u but not to v. When v is numbered, v must be

[2] Throughout this paper we shall use the notation $[i_1, i_2]$ to denote the set of integers $\{i \mid i_1\leqq i\leqq i_2\}$.

adjacent to at least as many numbered vertices as u. Thus, since u but not v is adjacent to w, v is adjacent to some other numbered vertex x not adjacent to u. Since this holds for all such u, v, w, ordering α has Property P. □

Remark. It is easy to show that any ordering generated by lexicographic search also has Property P.

We can implement maximum cardinality search as follows. We maintain an array of sets *set*(i) for $0 \leqq i \leqq m-1$. We store in *set* (i) all unnumbered vertices adjacent to exactly i numbered vertices. Initially *set* (0) contains all the vertices. We maintain the largest index j such that *set* (j) is nonempty. To carry out a step of the search, we remove a vertex v from *set* (j) and number it. For each unnumbered vertex w adjacent to v, we move w from the set containing it, say *set* (i), to *set* $(i+1)$. Then we add one to j and while *set* (j) is empty repeatedly decrement j. If we represent each set by a doubly linked list of vertices (to facilitate deletion) and maintain for each vertex the index of the set containing it, the search requires $O(n+m)$ time. (Since j is incremented n times and is never less than -1, the total time to manipulate j is $O(n)$.)

The following program written in a variant of Dijkstra's guarded command language [5], implements maximum cardinality search. For any unnumbered vertex v, *size* (v) is the number of numbered vertices adjacent to v. If v is a numbered vertex, we define *size* (v) to be -1.

MAXIMUM CARDINALITY SEARCH.

```
local j, v;
for i ∈ [0, n−1] → set (i) := ∅ rof;
for v ∈ vertices → size (v) := 0; add v to set (0) rof;
i := n; j := 0;
do i ≧ 1 →
    v := delete any from set (j);
    α(v) := i; α^{-1}(i) := v; size (v) := −1;
    for {v, w} ∈ E such that size (w) ≧ 0 →
        delete w from set (size (w));
        size (w) := size (w) + 1;
        add w to set (size (w))
    rof;
    i := i − 1;
    j := j + 1;
    do j ≧ 0 and set (j) = ∅ → j := j − 1 od
od;
```

Let us turn to Step 2 of the chordality test. We shall describe an algorithm that computes the fill-in produced by an arbitrary numbering α of an arbitrary graph G in $O(n+m')$ time, where m' is the number of edges in $G(\alpha)$. The algorithm, a simplification of the RTL method, was discovered by Greg Whitten [17], who presented it at the SIAM Symposium on Sparse Matrix Computations, held in Knoxville, Tennessee in 1978, but never published it, even in the conference proceedings.

Let G be a graph, let α be an ordering of G, and let $G(\alpha) = (V, E \cup F(\alpha))$ be the elimination graph of G with ordering α. For any vertex v, let $f(v)$, the *follower* of v, be the vertex of smallest number that is both adjacent to v in $G(\alpha)$ and has number larger than that of v. (See Figs. 1 and 2.) Note that a vertex need not have a follower. We define $f^i(v)$ for $i \geqq 0$ by $f^0(v) = v$, $f^{i+1}(v) = f(f^i(v))$.

LEMMA 5. *If* $\{v, w\} \in E \cup F(\alpha)$ *with* $v <_\alpha w$, *then* $f^i(v) = w$ *for some* $i \geqq 1$.

Proof. We use induction on $\alpha(v)$ from n to 1. If $f(v) \neq w$, then $v <_\alpha f(v) <_\alpha w$ by the definition of f. By Lemma 1, $\{f(v), w\} \in E \cup F(\alpha)$. By the induction hypothesis $f^i(f(v)) = w$ for some $i \geqq 1$, i.e., $f^{i+1}(v) = w$. □

THEOREM 3. *A pair $\{v, w\}$ with $v <_\alpha w$ is in $E \cup F(\alpha)$ if and only if there is a vertex x such that $\{x, w\} \in E$ and $f^i(x) = v$ for some $i \geqq 0$.*

Proof. Let $\{v, w\}$ be a pair with $v <_\alpha w$. Suppose there is a vertex x such that $\{x, w\} \in E$ and $f^i(x) = v$ for some $i \geqq 0$. Then $x \leqq_\alpha v$ by the definition of f. Since α is a zero-fill-in ordering for $G(\alpha)$ by Lemma 3, $\{v, w\} \in E \cup F(\alpha)$ by the definition of fill-in.

Conversely, suppose $\{v, w\} \in E \cup F(\alpha)$. We prove by induction on $\alpha(v)$ that there is a vertex x such that $\{x, w\} \in E$ and $f^i(x) = v$ for some $i \geqq 0$. If $\{v, w\} \in E$, $x = v$ and $i = 0$ satisfy the theorem. Otherwise, by Lemma 1 there is a vertex u such that $\{u, v\}$, $\{u, w\} \in E \cup F(\alpha)$ and $u <_\alpha v$. By the induction hypothesis there is a vertex x such that $\{x, w\} \in E$ and $f^i(x) = u$ for some $i \geqq 0$. By Lemma 4, $f^j(u) = v$ for some $j \geqq 0$. Thus x and $i + j$ satisfy the theorem. □

Theorem 3 leads to a fast algorithm for computing fill-in. We process the vertices in order from the vertex numbered 1 to the vertex numbered n. When processing vertex w, we compute the set $A(w)$ of all vertices v such that $\{v, w\} \in E \cup F(\alpha)$ and $v <_\alpha w$; we also find all vertices v such that $f(v)$ should be w, and define $f(v) = w$. To compute $A(w)$ we initialize it to contain all vertices v such that $\{v, w\} \in E$ and $v <_\alpha w$. Then we repeat the following step until it no longer applies: Select a vertex $v \in A(w)$ such that $f(v)$ has been computed (i.e., $f(v) <_\alpha w$) and $f(v) \notin A(w)$, and add $f(v)$ to $A(w)$. After constructing $A(w)$, define $f(v)$ to be w for all vertices $v \in A(w)$ such that $f(v)$ has not yet been defined.

We shall implement a variant of this algorithm to run in $O(n + m')$ time. We use two arrays, f and *index*. When a vertex v with $\alpha(v) = i$ is processed, we initialize $f(v)$ to be v and *index* (v) to be i. The first time we process a higher numbered vertex w adjacent to v, we define $f(v)$ to be w. Every time we process a higher numbered vertex w adjacent to v, we define *index* $(v) = \alpha(w)$. Thus *index* (v) is always the maximum number in the set $\{v\} \cup \{w | \{v, w\} \in E$ and w has been processed$\}$. This idea of an index array is due to Gustavson [11]. To process a vertex w, we repeat the following step for each vertex v such that $\{v, w\} \in E$ and $v <_\alpha w$:

General step. Initialize x to v. While *index* $(x) < \alpha(w)$, set *index* $(x) = \alpha(w)$, add $\{x, w\}$ to $E \cup F(\alpha)$, replace x by $f(x)$, and repeat. When *index* $(x) = \alpha(w)$, if $f(x) = x$ set $f(x) = w$.

The following program implements this method:

```
FILL-IN COMPUTATION.
    local v, w, x;
    for i ∈ [1, n] →
        w := α⁻¹(i); f(w) := w; index (w) := i
        for {v, w} ∈ E such that α(v) < i →
            x := v;
            do index (x) < i →
                index (x) := i;
                add {x, w} to E ∪ F(α);
                x := f(x)
            od;
            if f(x) = x → f(x) := w fi
        rof
    rof;
```

This fill-in algorithm can be used not only in Step 2 of the chordality test, but also in the symbolic factorization step of Gaussian elimination on sparse symmetric matrices [16]. If we only want to test for zero fill-in, as is the case in Step 2 of the chordality test, we can restate the algorithm as follows. Compute $f(v)$ for every vertex v. For every $\{v, w\} \in E$ such that $v <_\alpha w$, verify that either $\{f(v), w\} \in E$ or $f(v) = w$. The following program performs this test:

```
TEST FOR ZERO FILL-IN.
    for i ∈ [1, n] →
        w := α⁻¹(i); f(w) := w; index (w) := i;
        for v such that {v, w} ∈ E and α(v) < i →
            index (v) := i; if f(v) = v → f(v) := w fi
        rof;
        for v such that {v, w} ∈ E and α(v) < i →
            if index (f(v)) < i → reject fi
        rof;
    rof;
    accept;
```

Since the zero-fill-in test terminates as soon as it detects a fill-in edge, it runs in $O(n+m)$ time, rather than in the $O(n+m')$ time needed to actually compute the fill-in.

This completes our implementation of Steps 1 and 2 and gives us a simple $O(n+m)$-time chordality test.

Remark. In both the fill-in computation and the zero-fill-in test we can replace the array *index* by a bit array, say *test*, such that *test* $(v)=$ **true** when vertex w is processed if and only if *index* $(v) = \alpha(w)$. This saves space but requires an extra pass after each vertex is processed, to reset *test* (v) to **false** for each v such that $\{v, w\} \in E \cup F(\alpha)$ and $v <_\alpha w$.

3. Testing acyclicity of hypergraphs. Let $H = (V, E)$ be a hypergraph. We shall assume without loss of generality that every edge is nonempty and every vertex is contained in at least one edge. H is acyclic if and only if either of the following equivalent conditions holds [1]:

(1) All the vertices of H can be deleted by repeatedly applying the following two operations:
  (i) delete a vertex that occurs in only one edge;
  (ii) delete an edge that is contained in another edge.

(2) There is a forest F (called the *join forest*) with the edges of H as vertices, such that for every vertex v of H, the subgraph of F induced by those vertices (edges of H) that contain v is connected.

Condition (1) leads directly to an algorithm for testing the acyclicity of a hypergraph [10], [20]. Another algorithm based on condition (2) appears in [4]. Both of these algorithms run in time quadratic in the size of the hypergraph. Yannakakis [19] has given a linear-time algorithm based on the definition of acyclicity, using techniques from the RTL algorithm. We shall use the techniques of § 2 to obtain a simplified linear-time test.

Our algorithm for testing hypergraph acyclicity consists of two steps, analogous to those of the graph chordality test:

Step 1. Compute an ordering α of $G(H)$ that is guaranteed to be zero fill-in if H is acyclic.

Step 2. Check that α is zero-fill-in and that H is conformal.

Since $G(H)$ is chordal if H is acyclic, we could carry out Step 1 by applying maximum cardinality search to $G(H)$. However, $G(H)$ may have size quadratic in the size of H; consider for example the case of a hypergraph with only a single edge, containing all the vertices. Therefore we shall use the following variant of maximum cardinality search, which operates directly on H: Number the vertices from n to 1 in decreasing order. As the next vertex to number, select any unnumbered vertex in an edge of H containing as many numbered vertices as possible, breaking ties arbitrarily. We call this method *maximum cardinality search on hypergraphs.*

THEOREM 4. *Suppose H is acyclic. Then any numbering α generated by a maximum cardinality search of H can also be generated by a maximum cardinality search of $G(H)$, and is thus zero-fill-in on $G(H)$.*

Proof. Consider applying maximum cardinality search in parallel to H and $G(H)$. We must show that each time we choose a vertex v to number in H, we can choose the same vertex to number in $G(H)$. Suppose i vertices have been numbered identically in H and $G(H)$, for some $i \geqq 0$. For any vertex v, let $B(v)$ be the set of numbered vertices adjacent to v.

We first prove that if w is any vertex that can be numbered next in $G(H)$, then there is some edge R of H such that $B(w)$ is exactly the set of numbered vertices in R. Let w be such a vertex, i.e., suppose $|B(w)|$ is maximum among unnumbered vertices. Since $G(H)$ is chordal and maximum cardinality search generates zero-fill-in numberings on chordal graphs, every pair of vertices in $B(w)$ must be adjacent in $G(H)$. Thus $\{w\} \cup B(w)$ is a clique in $G(H)$. Since H is conformal there is some edge R of H such that $\{w\} \cup B(w) \subseteq R$. Furthermore R cannot contain any numbered vertices not in $B(w)$, since all vertices in R are adjacent to w in $G(H)$. This means that $B(w)$ is exactly the set of numbered vertices in R.

Now let w and R be as above and suppose v can be numbered next in H; that is, v is an unnumbered vertex in an edge S of H containing as many numbered vertices as possible. Then S contains at least as many numbered vertices as R, and since every numbered vertex in S is adjacent to v in $G(H)$, the choice of w means that $|B(v)| = |B(w)|$. Thus v can be numbered next in $G(H)$. □

During a maximum cardinality search of a hypergraph, we call an edge *exhausted* if all vertices contained in it are numbered and *nonexhausted* otherwise. If R is a nonexhausted edge containing as many numbered vertices as possible and we number a vertex in R, then if R is still nonexhausted it still contains as many numbered vertices as possible. Thus after selecting a nonexhausted edge having as many numbered vertices as possible, we can number all its unnumbered vertices consecutively before selecting another edge. This restricted form of maximum cardinality search facilitates testing $G(H)$ for chordality and H for conformity. The following program implements this method. (See Figs. 3 and 4.) In addition to numbering the vertices, the program performs the following computations: It numbers the selected edges from 1 to k in order of their selection; if S is the ith edge selected, then $S = R(i)$ and $\beta(S) = i$. It extends this numbering to the vertices by defining $\beta(v) = \min\{\beta(R) | R$ is selected and $v \in R\}$. Note that $v <_\beta w$ implies $v >_\alpha w$. Finally, for each edge S it computes $\gamma(S)$, defined to be $\max\{\beta(v) | v \in S\}$ if S is not among the selected edges and $\max\{\beta(v) | v \in S$ and $\beta(v) < \beta(S)\}$ if S is among the selected edges. (If $\beta(v) = \beta(S)$ for all $v \in S$, $\gamma(S)$ is undefined.) As an aid to the computation, the program maintains *size* (S) for each edge S, which is the count of numbered vertices in S if S is nonexhausted and minus one if S is exhausted. For $i \in [0, n-1]$, *set* (i) is the set of nonexhausted edges containing i numbered vertices. Index j is the maximum i such that *set* (i) is nonempty; index k counts the number of numbered edges.

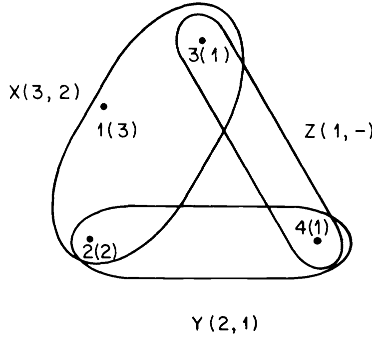


FIG. 3. *Restricted maximum cardinality search of a hypergraph. Numbers on vertices are α-numbers; numbers in parentheses are β-numbers. The first number associated with an edge is its β-number, if any; the second number is its γ-number. Edge Y fails the acyclicity test, since $\gamma(X)=\beta(Y)$ but $X \cap \{v \mid \beta(v)<\beta(Y)\} = \{3\} \not\subseteq Y$.*

RESTRICTED MAXIMUM CARDINALITY SEARCH ON HYPERGRAPHS.

local i, j, k, S;
for $i \in [0, n-1] \rightarrow$ *set* $(i) := \varnothing$ **rof**;
for $S \in E \rightarrow$ *size* $(S) := 0$; $\gamma(S) :=$ *undefined*; **add** S **to** *set* (0) **rof**;
$i := n+1$; $j := k := 0$;
do $j \geqq 0 \rightarrow$
  $S :=$ **delete any from** *set* (j);
  $\beta(S) := k := k+1$; $R(k) := S$; *size* $(S) := -1$;
  for $v \in S$ **such that** v is unnumbered $\rightarrow$
    $\alpha(v) := i := i+1$; $\beta(v) := k$
    for $S \in E$ **such that** $v \in S$ **and** *size* $(S) \geqq 0 \rightarrow$
      $\gamma(S) := k$;
      delete S **from** *set* (*size* (S));
      size $(S) :=$ *size* $(S)+1$;
      if *size* $(S) < |S| \rightarrow$ **add** S **to** *set* (*size* (S))
      $\square$ *size* $(S) = |S| \rightarrow$ *size* $(S) := -1$
      fi
    rof
  rof;
  $j := j+1$;
  do $j \geqq 0$ and *set* $(j) = \varnothing \rightarrow j := j-1$ **od**
od;

Remark. The assignment "$\alpha(v) := i := i-1$" in this program is a sequential assignment that computes $i-1$ and assigns its value to i and then to $\alpha(v)$.

As it happens, we can test the acyclicity of H using only β and γ; α is unnecessary and its computation can be dropped from the program. The following theorem gives our acyclicity test (see Figs. 3 and 4):

THEOREM 5. *H is acyclic if and only if for each $i \in [1, k]$ and each edge S such that $\gamma(S)=i$, $S \cap \{v \mid \beta(v)<i\} \subseteq R(i)$. (Since $\beta(v)=i$ implies $v \in R(i)$, this condition is equivalent to $S \cap \{v \mid \beta(v) \leqq i\} \subseteq R(i)$.)*

Proof. Suppose H is acyclic. Then α is a zero fill-in ordering of $G(H)$. Consider any $i \in [1, k]$. Suppose $\gamma(S)=i$. Then there is a vertex $u \in S$ such that $\beta(u)=i$. Vertex

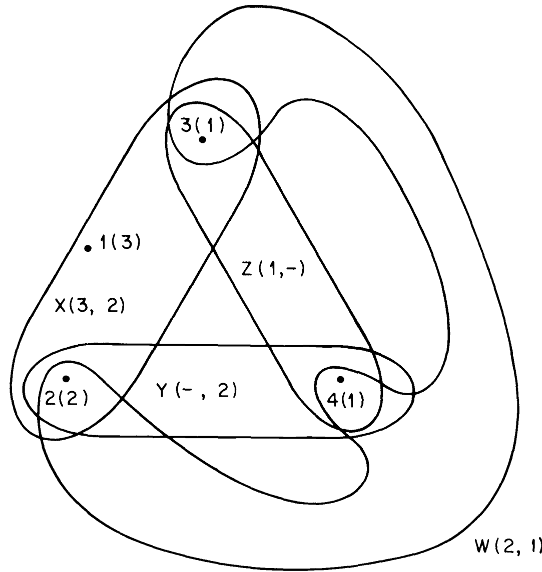


FIG. 4. *Restricted maximum cardinality search of an acyclic hypergraph. Notation is as in Fig. 3. All edges pass the acyclicity test. Note that edge Y is never selected.*

u and its adjacent larger numbered vertices (with respect to α) form a clique in $G(H)$, which since H is conformal must be contained in an edge of E, say T. Since $u \in S \cap R(i)$, $S \cap \{v|\beta(v) < i\} \subseteq S \cap \{v|\alpha(v) \geqq \alpha(u)\} \subseteq T$ and $R(i) \cap \{v|\alpha(v) \geqq \alpha(u)\} \subseteq T$. When u is numbered $R(i)$ contains at least as many numbered vertices as T, which means $R(i) \cap \{v|\alpha(v) \geqq \alpha(u)\} = T \cap \{v|\alpha(v) \geqq \alpha(u)\}$, and $S \cap \{v|\beta(v) < i\} \subseteq R(i)$.

To prove the converse, we use acyclicity condition (1). Suppose that for each $i \in [1, k]$ and each edge S such that $\gamma(S) = i$, $S \cap \{v|\beta(v) < i\} \subseteq R(i)$. We delete vertices and edges from H by processing the sets $R(i)$ from $i = k$ to $i = 1$. To process $R(i)$, we delete every set S such that $\gamma(S) = i$ and every vertex v such that $\beta(v) = i$. This deletion method maintains the following invariants: Just before a set $R(i)$ is processed, every remaining vertex v has $\beta(v) \leqq i$. Every remaining set S such that $\gamma(S) = i$ thus satisfies $S = S \cap \{v|\beta(v) \leqq i\} \subseteq R(i)$, and by rule (ii) S can be deleted when $R(i)$ is processed. Once all such sets are deleted, a vertex v with $\beta(v) = i$ is contained only in $R(i)$ and can be deleted by rule (i). (A set $S \neq R(i)$ containing v has $\gamma(S) \geqq i$; if $\gamma(S) > i$, S was deleted previously.) Thus we can delete all the vertices of H, and H is acyclic. □

We obtain a linear-time algorithm from Theorem 5 by performing together all the set inclusion tests involving a given $R(i)$.

ACYCLICITY TEST.
for $i \in [1, k] \rightarrow$
  for $S \in E$ **such that** $\gamma(S) = i \rightarrow$
    if $S \cap \{v|\beta(v) < i\} \not\subseteq R(i) \rightarrow$ *reject* **fi**
  rof
rof;
accept;

The following program fills in the details of this method. During testing of a set $R(i)$, *index* $(v)=i$ if $v \in R(i)$; *index* $(v)<i$ if $v \notin R(i)$.

```
for v ∈ V → index (v) := 0 rof;
for i ∈ [1, k] →
  for v ∈ R(i) → index (v) := i rof;
  for S ∈ E such that γ(S) = i →
    for v ∈ S → if β(v) < i and index (v) < i → reject fi rof
  rof
rof;
accept;
```

This gives an $O(n+m)$-time acyclicity test.

Remark. As in the fill-in computation, we can replace the array index by a bit array if we are willing to reset it after processing each $R(i)$.

If the hypergraph is acyclic we can construct a join forest for it (acyclicity condition (2)) from the parameters γ. Beeri et al. [2] show that if a hypergraph H passes the acyclicity test (1) then a join forest F for H can be constructed as follows: The vertices of F are the edges of H; if an edge R was deleted by operation (ii) of the acyclicity test (1) because it was contained in some other edge S, then F has an edge $\{R, S\}$. (If there were several edges containing R when R was deleted, then we arbitrarily pick one such edge S.) From the proof of Theorem 5 is follows that the forest F with the edges of H as vertices and with edges $\{S, R(\gamma(S))\}$ for all S with $\gamma(S)$ defined is a join forest for the acyclic hypergraph H.

4. Selectively reducing an acyclic hypergraph. We conclude this paper by considering the following problem. Suppose we are given a hypergraph H and a set of marked vertices. We wish to *selectively reduce* H by repeating the following two operations until neither is applicable:

(i) delete an unmarked vertex that occurs in only one edge;

(ii) delete an edge that is contained in another edge.

Selective reduction is necessary in computing queries in acyclic relational data bases. A relational data base is a collection of relations; each relation is a table over some set of attributes. A relational data base can be modeled by a hypergraph whose vertices are the attributes and whose edges are the relations (see [1], [21] for more information). Suppose now that we want to compute the relationship among the attributes in a given set X. If the hypergraph is acyclic this is done as follows: First, we mark the vertices of X and selectively reduce the hypergraph H. After the reduction we are left with a collection of partial edges (i.e. subsets of some of the original edges) $Y_1, \cdots, Y_k$. (It is easy to see that the selective reduction process has the Church–Rosser property; that is, the final collection of partial edges is the same, regardless of the order in which the operations (i), (ii) are applied.) For each Y_i we find all the edges of H that contain Y_i. We project the relations corresponding to such edges onto the attributes Y_i and take the union of the resulting relations, giving a single relation for each Y_i. Then we take the join of the relations obtained for $Y_1, \cdots, Y_k$ (see [13], [18]).

As an example of selective reduction, consider a hypothetical data base for storing information about research papers. Figure 5 illustrates the relations, which correspond to the schemes $R_1=\{AU, AF\}$, $R_2=\{PT, AU\}$, $R_3=\{PT, Y, JN, VN\}$, $R_4=\{PT, Y, CN\}$, $R_5=\{CN, CL, Y\}$, $R_6=\{PT, S\}$, where the abbreviations are as indicated

(a)

AU = AUTHOR
AF = AFFILIATION
CN = CONFERENCE NAME
JN = JOURNAL NAME
CL = CONFERENCE LOCATION
VN = JOURNAL VOLUME, PAGE NUMBERS
Y = PUBLICATION YEAR
PT = PAPER TITLE
S = SUBJECTS

(b)

AU	AF
FRED	IBM
SUE	BELL

PT	AU
NICE RESULTS	FRED
NICER RESULTS	SUE

PT	Y	JN	VN
NICE RESULTS	1982	SICOMP	10, 820-830

PT	Y	CN
NICER RESULTS	1983	STOC

CN	CL	Y
STOC	BOSTON	1983

PT	S
NICE RESULTS	SORTING
NICER RESULTS	SORTING

FIG. 5. *A relational data base for research papers.* (a) *Abbreviations of attributes.* (b) *Relations. Only the entries for two papers are shown.*

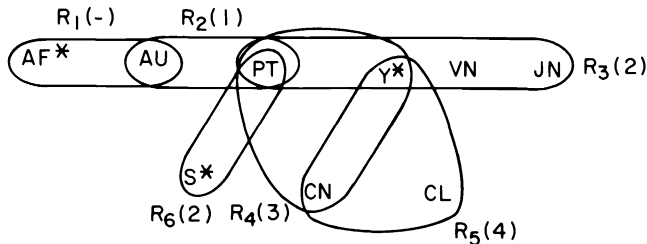


FIG. 6. *Hypergraph corresponding to relational schemes in Fig. 5. Relations are indexed in β-order of a restricted maximum cardinality search. Numbers in parentheses are γ-numbers. Applying the selective reduction algorithm, we delete VN, JN and CL (which are unmarked and in only one edge), then R_5 (now contained in R_4), then CN, then R_3, leaving R_1, R_2, R_6 and $\{PT, Y\}$.*

in Fig. 5. Figure 6 shows the hypergraph corresponding to this relation scheme. Suppose we wish to know whether any Bell Laboratories authors published any papers on graph theory in 1983. To answer this query, we must compute the relationship among the attributes in $X = \{AF, S, Y\}$, select those tuples with $AF =$ "Bell Laboratories," $S =$ "graph theory," and $Y =$ "1983," and check whether the result is empty.

After carrying out the appropriate selective reduction, the edges remaining are $R_1 = \{AU, AF\}$, $R_2 = \{PT, AU\}$, $R_6 = \{PT, S\}$, and $\{PT, Y\}$, the last of which is contained in two original edges, R_3 and R_4. Thus we join R_1, R_2, and R_6 with the union of projections of R_3 and R_4 on $\{PT, Y\}$.

Kuper describes a nonlinear algorithm for the selective reduction of an acyclic hypergraph [12]. We shall present a simple linear-time algorithm. In order to solve this problem, we first consider a related but simpler problem: given an acyclic

hypergraph, determine which of its edges are maximal (in the set-theoretic sense). The edges that are not maximal can always be deleted from H in any selective reduction of H, by operation (ii). It is easy to see that if H is acyclic, all its maximal edges are selected during a maximum cardinality search. (Thus any nonselected edge can be deleted during the selective reduction process.) However, not all selected edges need be maximal. We can ensure that only maximal edges are selected by breaking ties according to cardinality (largest edge preferred). However, there seems to be no easy way to incorporate this into the algorithm while preserving the $O(n+m)$ time bound. The following theorem gives an alternative characterization of maximal edges. If R is a selected edge, let R' be the set $\{v \in R | \beta(v) < \beta(R)\}$. (Note that $\gamma(R) = \max\{\beta(v) | v \in R'\}$.)

THEOREM 6. *Suppose we carry out a restricted maximum cardinality search on an acyclic hypergraph H, selecting edges* $R(1), R(2), \cdots, R(k)$. *Then the maximal edges of H are exactly the selected edges* $R(i)$ *such that* $i = k$ *or* $|R(i)| > |R'(i+1)|$.

Proof. Suppose S is a nonselected edge. Let $\gamma(S) = i$. Then $S = S \cap \{v | \beta(v) \leqq i\} \subseteq R(i)$ by Theorem 5, and S is not maximal.

Suppose $R(i)$ for some $i < k$ is maximal. If $\gamma(R(i+1)) = i$, then $R'(i+1) \subseteq R(i)$ by Theorem 5, which means $|R'(i+1)| < |R(i)|$, since $R'(i+1) = R(i)$ would contradict the maximality of $R(i)$. If $\gamma(R(i+1)) \neq i$, then $|R'(i+1)| = |R(i+1) \cap \{v | \beta(v) < i\}| \leqq |R'(i)| < |R(i)|$ by the definition of maximum cardinality search.

Conversely, suppose $R(i)$ is not maximal. Then there is some maximal edge containing $R(i)$, which must be $R(j)$ for some $j > i$. Consider $R(i+1)$. Just before $R(i)$ is selected, $R(i+1)$ contains at most $|R'(i)|$ numbered vertices; just after $R(i)$ is selected, $R(i+1)$ contains at least as many numbered vertices as $R(j)$, and hence at least $|R(i)|$. Since exactly $|R(i) - R'(i)|$ new vertices are numbered when $R(i)$ is selected, $R(i+1)$ contains every vertex in $R(i) - R'(i)$, and $|R'(i+1)| = |R(i)|$. □

We can compute the maximal edges of an acyclic hypergraph in $O(n+m)$ time by carrying out a restricted maximum cardinality search and applying the test in Theorem 6 to each selected edge. Suppose we have done this. Let $R(1)$, $R(2), \cdots, R(k)$ be the list of selected edges according to the restricted maximum cardinality search on the acyclic hypergraph H, and suppose that $R(i)$ is not maximal. By the proof of Theorem 6, just before $R(i)$ is selected, both $R(i)$ and $R(i+1)$ contain the same number of numbered vertices, and $R(i+1)$ contains all unnumbered vertices in $R(i)$. Thus we could have selected $R(i+1)$ in place of $R(i)$ during the search. After selecting $R(i+1)$, $R(i)$ would be exhausted and not eligible for later selection, and the rest of the edges $R(i+2), \cdots, R(k)$ could be chosen in the same order as before. Thus $R(1), \cdots, R(i-1), R(i+1), \cdots, R(k)$ is also a valid selection order (according to maximum cardinality search) for the hypergraph. This means that if we delete from the list of selected edges all nonmaximal edges we are left with a list that corresponds to a valid selection order. The selective reduction process can always begin by deleting all edges not in this list, since each is contained in some edge in the list.

Redefine $R(1), \cdots, R(k)$ to be the remaining (selected) edges, all incomparable, and suppose that some vertex v occurs in only one edge $R(i)$. Consider what happens if we delete v. The remaining hypergraph H' is also acyclic [2]. If the new $R(i)$ (without v) contains only vertices that appear in earlier sets $R(j)$ (i.e. with $j < i$), then $R(i)$ is exhausted when its turn comes, and therefore cannot be selected. However, since v occurs only in $R(i)$ no other edge is affected and therefore $R(1), \cdots, R(i-1)$, $R(i+1), \cdots, R(k)$ is a valid selection order for H'. Since $R(i)$ is not selected and H' is acyclic we know from the previous section that the new $R(i)$ (without v) is contained in some other edge $R(j)$.

If the new $R(i)$ contains some vertex that does not appear in any earlier $R(j)$, then the same order $R(1), \cdots, R(i), \cdots, R(k)$ continues being a valid selection order for H'. From Theorem 6 we know that all these edges are incomparable, unless $|R(i)| = |R'(i+1)|$, in which case $R(i) \subseteq R(i+1)$ and $R(i)$ can be dropped from the list of selected edges without violating the validity of the selection order.

Our discussion suggests the following algorithm for the selective reduction of an acyclic hypergraph. (See Fig. 6.)

Step 1. Carry out a restricted maximum cardinality search of the hypergraph. Apply Theorem 6 to discard the selected edges that are not maximal. Let $R(1), \cdots, R(k)$ be the list of remaining edges, and let $|R'(i)|$ be as defined before Theorem 6. For each unmarked vertex v, compute the number of remaining (selected) edges in which v appears.

Step 2. Repeat the following operation until it does not apply:

Delete any unmarked vertex v that occurs in exactly one edge $R(i)$; decrement $|R(i)|$. If $|R(i)| = |R'(i)|$ or $|R(i)| = |R'(i+1)|$ then do the following: Delete $R(i)$. For each vertex $v \in R(i)$, decrement the count of edges in which v appears. Decrease $|R'(i+1)|$ by $|R(i)| - |R'(i)|$. For $j \geqq i$, replace $R(j)$ by $R(j+1)$.

The selected edges that remain after the execution of these steps are exactly those that cannot be deleted by the selective reduction process. If we maintain a list of the unmarked vertices occurring in exactly one edge and maintain the set of remaining edges $R(1)$, $R(2) \cdots$ as a doubly linked list, we can carry out this computation in $O(n+m)$ time. We leave the implementation of this algorithm as an exercise. The correctness of the method follows from our previous discussion.

Note that in the data base application, Step 1 will be executed only once (when the data base is set up), and only Step 2 will be executed to answer a query. Let $Y_1, \cdots, Y_k$ be the remaining sets of vertices when the algorithm terminates. We have to find now for each Y_i the edges of the original hypergraph H that contain Y_i. Let W be the set of remaining vertices, i.e. the union of the Y_i's. Let $\hat{H}$ be the hypergraph with set of vertices W and an edge $\hat{S} = S \cap W$ for each edge S of H. Then $\hat{H}$ is acyclic [2]. (Strictly speaking, $\hat{H}$ is a multihypergraph; i.e., it might have edges that consist of exactly the same vertices. However, everything we have said about hypergraphs also holds for multihypergraphs.) A valid selection order for $\hat{H}$ according to maximum cardinality search is $Y_1, \cdots, Y_k$. For a vertex v in W, let $\hat{\beta}(v) = \min\{i | v \in Y_i\}$, and for an edge S of H let $\hat{\gamma}(S) = \max\{\hat{\beta}(v) | v \in \hat{S}\}$. From the last section we know that $\hat{S} = S \cap W \subseteq Y_{\hat{\gamma}(S)}$ for each edge S of H. If S contains some Y_i then this can be only $Y_{\hat{\gamma}(S)}$ (since all the Y_j's are incomparable), and this is true if and only if $|S \cap W| = |Y_{\hat{\gamma}(S)}|$. Therefore, all we have to do is compute $\hat{\beta}(v)$ for each $v \in W$, compute $\hat{\gamma}(S)$ for each edge S of H, and compare $|S \cap W|$ to $|Y_{\hat{\gamma}(S)}|$.

REFERENCES

[1] C. BEERI, R. FAGIN, D. MAIER, A. MENDELZON, J. D. ULLMAN AND M. YANNAKAKIS, *Properties of acyclic database schemes*, in Proc. 13th Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1981, pp. 355–362.

[2] C. BEERI, R. FAGIN, D. MAIER AND M. YANNAKAKIS, *On the desirability of acyclic database schemes*, J. Assoc. Comput., 30 (1983), pp 479–513.

[3] C. BERGE, *Graphs and Hypergraphs*, North-Holland, Amsterdam, 1973.

[4] P. A. BERNSTEIN AND N. GOODMAN, *The power of natural semijoins*, this Journal, 10 (1981), pp. 751–771.

[5] E. W. DIJKSTRA, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[6] G. A. DIRAC, *On rigid circuit graphs*, Abh. Math. Sem. Univ. Hamburg, 25 (1961), pp 71–76.
[7] R. FAGIN, A. O. MENDELZON AND J. D. ULLMAN, *A simplified universal relation assumption and its properties*, ACM Trans. Database Systems, 7 (1982), 343–360.
[8] D. R. FULKERSON AND O. A. GROSS, *Incidence matrices and interval graphs*, Pacific J. Math., 15 (1965), pp. 835–855.
[9] M. GOODMAN AND O. SHMUELI, *Syntactic characterizations of tree database schemes*, J. Assoc. Comput., Mach., 30 (1983), 767–786.
[10] M. H. GRAHAM, *On the universal relation*, Technical Report, Univ. of Toronto, Toronto, Ontario, Canada, 1979.
[11] F. G. GUSTAVSON, *Some basic techniques for solving sparse systems of linear equations*, in Sparse Matrices and Their Applications, D. S. Rose and R. A. Willoughby, eds., Plenum Press, New York, 1972, pp. 41–52.
[12] G. KUPER, *An algorithm for reducing acyclic hypergraphs*, unpublished manuscript, Stanford Univ., Stanford, CA, 1982.
[13] D. MAIER AND J. D. ULLMAN, *Connections in acyclic hypergraphs*, in Proc. ACM Symposium on Principles of Database Systems, Association for Computing Machinery, New York, 1982, pp. 34–39.
[14] D. J. ROSE, *Triangulated graphs and the elimination process*, J. Math. Anal. Appl., 32 (1970), pp. 597–609.
[15] D. J. ROSE, R. E. TARJAN AND G. S. LUEKER, *Algorithmic aspects of vertex elimination on graphs*, this Journal, 5 (1976), pp. 266–283.
[16] R. E. TARJAN, *Graph theory and Gaussian elimination*, in Sparse Matrix Computations, J. R. Bunch and D. J. Rose, eds., Academic Press, New York, 1976, pp. 3–22.
[17] G. WHITTEN, private communication, 1978.
[18] M. YANNAKAKIS, *Algorithms for acyclic database schemes*, Proc. International Conference on Very Large Data Bases, 1981, pp. 82–94.
[19] M. YANNAKAKIS, *A linear-time algorithm for recognizing acyclic hypergraphs*, unpublished manuscript, 1982.
[20] C. T. YU AND M. Z. OZSOYOGLU, *An algorithm for tree-query membership of a distributed query*, in Proc. 1979 IEEE COMPSAC, Institute of Electrical and Electronic Engineers, New York, 1979, pp. 306–312.
[21] C. ZANIOLO, *Analysis and design of relational schemata for database systems*, Ph.D. thesis, Univ. of California at Los Angeles, Los Angeles, CA, 1976.

SIAM J. COMPUT.
Vol. 13, No. 3, August 1984
© 1984 Society for Industrial and Applied Mathematics
008

EFFICIENT PARALLEL ALGORITHMS FOR A CLASS OF GRAPH THEORETIC PROBLEMS*

YUNG H. TSIN†‡ AND FRANCIS Y. CHIN†

Abstract. In this paper, we present efficient parallel algorithms for the following graph problems: finding the lowest common ancestors for vertex pairs of a directed tree; finding all fundamental cycles, a directed spanning forest, all bridges, all bridge-connected components, all separation vertices, all biconnected components, and testing the biconnectivity of an undirected graph. All these algorithms achieve the $O(\lg^2 n)$ time bound, with the first two algorithms using $n\lceil n/\lg n\rceil$ processors and the remaining algorithms using $n\lceil n/\lg^2 n\rceil$ processors. In all cases, our algorithms are better than the previously known algorithms and in most cases reduce the number of processors used by a factor of $n \lg n$. Moreover, our algorithms are optimal with respect to the time-processor product for dense graphs, with the exception of the first two algorithms.

The machine model we use is the PRAM which is a SIMD model allowing simultaneous reads but not simultaneous writes to the same memory location.

Key words. parallel computation, analysis of algorithms, graph algorithms, directed spanning forests, lowest common ancestors, fundamental cycles, bridges, bridge-connected components, separation vertices, biconnected components, SIMD machines, PRAM

1. Introduction. The design of efficient parallel algorithms for graph problems has been investigated by many people [2], [3], [4], [5], [7], [8], [12], [15], [16], [17]. In particular, Chin, Lam and Chen [3], [4] designed parallel algorithms for several graph problems in which the processor-time products achieve the lower bounds for the corresponding sequential algorithms for dense graphs. In this paper, we present efficient parallel algorithms for other graph problems in which the processor-time products differ from the lower bounds for their sequential counterparts for dense graphs by at most a factor of $\lg n$.[1]

We are interested in the following graph problems: finding the lowest common ancestors for $q(1 \leqq q \leqq n^2)$ vertex pairs of a directed tree; finding a complete set of fundamental cycles, a directed spanning forest, all bridges, all bridge-connected components, all biconnected components, all separation vertices and testing the biconnectivity of an undirected graph. This class of problems has also been studied by Savage [15] and Savage and Ja'Ja' [16]. They designed parallel algorithms for these problems and achieved an $O(\lg^2 n)$ time bound with the processor-time products being $O(n^2 \lg^2 n)$ for the directed spanning tree problem and being $O(n^3)$ or $O(n^2(\lg n)^m)$, where $m \geqq 3$, for the remaining problems. In this paper, we present parallel algorithms for the same class of problems. Our algorithm for the lowest common ancestors problem takes $O(\lceil q/nK\rceil \cdot \lg n + n/K)$ time with $nK (K>0)$ processors. The algorithm for the fundamental cycles problem takes $O(\lceil |E|/nK\rceil \cdot \lg n + n/K + \lg^2 n)$ time with $nK (K \geqq 1)$ processors, where E is the edge set of the undirected graphs. The algorithms for the remaining problems all take $O(n/K + \lg^2 n)$ time with $nK (K \geqq 1)$ processors. In particular, an $O(\lg^2 n)$ time bound can be achieved with $K = \lceil n/\lg n\rceil$ for the first two problems and with $K = \lceil n/\lg^2 n\rceil$ for the remaining problems. As the processor-time products of our algorithms are at most $O(n^2 \lg n)$, for $1 \leqq K \leqq \lceil n/\lg n\rceil$, our algorithms are better than the previously known results in all cases, and in most cases

* Received by the editors April 5, 1982 and in final revised form June 1, 1983. This research was supported by the Natural Science and Engineering Research Council of Canada under grant NSERC-A4319.

† Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1.

‡ Present address: Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada A1C 5S7.

[1] Throughout this paper, we use $\lg n$ to denote $\lceil \log_2 n\rceil$.

use less processors by a factor of $n \lg n$. Except for the algorithms for the first two problems, the processor-time products of our algorithms are $O(n^2)$, which is optimal for dense graphs.

The computation model we use is the single-instruction stream multiple-data stream (SIMD) model. We assume that all processors have access to a common memory, and that simultaneous reads from the same location are allowed but simultaneous writes on the same location are prohibited. This model is called PRAM in [6].

In describing our parallel algorithms, we use the instruction introduced by Preparata and Vuillemin [11]. Specifically, parallel operations are controlled by

for all $i{:}P(i)$ **pardo** instructions **dopar**;

where $P(i)$ is a predicate of i.

2. Definitions and notation. A *graph* $G(V, E)$ consists of a finite nonempty set V of vertices and a set E of pairs of vertices called edges. If the edges are unordered pairs, then G is *undirected*; otherwise G is *directed*. Without loss of generality, we assume $V=\{1, 2, \cdots, n\}$ throughout this paper. If for every two vertices u, v in V, there is a path in G joining u and v, then G is *connected*. Each connected maximal subgraph of G is called a *component* of G. An *adjacency matrix* M of G is a $n\times n$ Boolean matrix such that $M[u, v]=1$ if and only if $(u, v)\in E$. A *tree* is a connected undirected graph with no cycles in it. Let $T(V', E')$ be a directed graph, T is said to have a *root* r, if $r\in V'$ and every vertex $v\in V'$ is reachable from r via a directed path. If the underlying undirected graph of T is a tree, then T is a *directed tree*. If, moreover, the underlying graph of T is a subgraph of a connected undirected graph $G(V, E)$ and $V'=V$, then T is a *directed spanning tree* in G. A *directed forest* is a graph whose connected components are directed trees. If T is a directed forest such that each directed tree in T is a directed spanning tree of a component of an undirected graph G and vice versa, then T is called a *directed spanning forest* of G. If the edges of T are all reversed, the resulting graph is called an *inverted spanning forest* of G. *Inverted spanning trees*, *inverted trees*, *inverted forests*, etc. are defined similarly. Throughout this paper, we denote the "undirected" path from vertex a to vertex b in a (directed) tree by $[a *\!\!\to b]$, and by $[a *\!\!\to b)$ if vertex b is to be excluded. If the path consists of at least one edge, then the "$*$" is removed from the notation.

An inverted tree T is called an *ordered tree* if the sons of every vertex in T are ordered. If v is the ith son of a vertex in T, then the *rank* of v is i.

Let $T(V', E')$ be a directed tree, and u, $v\in V'$, the *lowest common ancestor* (LCA (u, v)) of u and v in T is the vertex $w\in V'$ such that w is a common ancestor of u and v, and any other common ancestor of u and v in T is also an ancestor of w in T. If T is a spanning tree of a connected, undirected graph G, let (u, v) be an edge in $G-T$, then the cycle in G consisting of the paths $[u *\!\!\to \text{LCA}(u, v)]$, $[\text{LCA}(u, v) *\!\!\to v]$ and the edge (v, u) is a *fundamental cycle* in G. Let $e\in E$, e is a *bridge* in G if and only if e is not on any cycle in G. Let B be the set of bridges in G, every connected component of the graph $G'(V, E-B)$ is a *bridge-connected component* of G. Let $a\in V$, if there exist $u, v\in V$ such that u, v, a are all distinct and such that every path connecting u and v in G passes through a, then a is called a *separation vertex* of G. A graph is *biconnected* if it contains no separation vertex. Every maximal biconnected subgraph of G is called a *biconnected component* of G. To test the *biconnectivity* of G, is to test if G is biconnected.

3. Two useful lemmas. In this paper, we will frequently use the following two lemmas in analyzing the time and processor complexities.

LEMMA 3.1. *Given n elements* $\{a_0, a_1, \cdots, a_{n-1}\}$, *let f be a function to be applied to every element. If computing* $f(a_i)$ *takes t time units and* $K(\geqq 1)$ *processors are provided, then* $f(a_i)$, $0 \leqq i \leqq n-1$, *can be computed in* $(\lceil n/K \rceil * t)$ *parallel time units.*

LEMMA 3.2 [3], [4]. *Given n elements* $\{a_0, a_1, \cdots, a_{n-1}\}$ *and K processors,* $A(n) = a_0 * a_1 * a_2 * \cdots * a_{n-1}$ *can be computed in T parallel time units where* $*$ *is any associative binary operator and*

$$T = \begin{cases} \lceil n/K \rceil - 1 + \lg K & \text{if } \lfloor n/2 \rfloor > K, \\ \lg n & \text{if } \lfloor n/2 \rfloor \leqq K. \end{cases}$$

4. Finding all paths from the vertices to the roots in an inverted forest. In this section, we present a method for constructing an array, denoted by $\mathbf{F}^+$, in which each row contains a path from a vertex to a root in an inverted forest. The array will be very useful in the design of parallel algorithms presented in the following sections.

Let $T(V', E')$ be an inverted forest with $|V'| = n$, without loss of generality, we assume $V' = \{1, 2, \cdots, n\}$. Let $\{T_j\}$ be the set of all inverted trees in T and $\{r_j\}$ be the set of all their roots.

DEFINITION. $F: V' \to V'$ is a function such that

$$F(i) = \text{the father of the vertex } i \text{ in } T \quad \text{for } i \notin \{r_j\},$$

$$F(r) = r \quad \forall r \in \{r_j\}.$$

The function F can be represented by a directed graph F which can be constructed from T by adding a self-loop at each root r_j in T.

From the function F, we define F^k, $k \geqq 0$, as follows:

DEFINITION. $F^k: V' \to V'$, $k \geqq 0$, is a function such that

$$F^0(i) = i \quad \forall i \in V',$$

$$F^k(i) = F(F^{k-1}(i)) \quad \forall i \in V', \quad k > 0.$$

If i is a vertex in T_j, $F^k(i)$ is the kth ancestor of i in T_j or r_j.

DEFINITION. For each $i \in V'$, if i is in T_j, for some j, then

$$\text{depth}(i) = \min\{k \mid F^k(i) = r_j \text{ and } 0 \leqq k \leqq n-1\}.$$

The concepts $F^k(i)$, $k \geqq 0$ and depth (i), $1 \leqq i \leqq n$, were first introduced by Savage in [15]. She showed that given the function F of a directed forest T (T could be a directed forest or an inverted forest), $F^k(i)$, $0 \leqq k \leqq n-1$, and depth (i), $1 \leqq i \leqq n$, can be computed in $O(\lg n)$ time with n^2 processors and $n\lceil n/\lg n \rceil$ processors respectively. In the following, we will show in Theorem 4.1 that $F^k(i)$, $0 \leqq k \leqq n-1$, $1 \leqq i \leqq n$, can indeed be computed in $O(\lg n)$ time with $n\lceil n/\lg n \rceil$ processors or in $O(\lg^2 n)$ time with $n\lceil n/\lg^2 n \rceil$ processors, and then depth (i) can be computed in $O(\lg n)$ additional time with n processors.

THEOREM 4.2. (i) *Given the function F of a directed or an inverted forest T,* $F^k(i)$, $i \in V'$, $0 \leqq k \leqq n-1$ *can be computed in* $O(n/K + \lg n)$ *time with nK* $(K > 0)$ *processors.*

(ii) *Given* $F^k(i)$, $0 \leqq k \leqq n-1$, $1 \leqq i \leqq n$, *and* $nK (K > 0)$ *processors, depth* (i), $1 \leqq i \leqq n$ *can be computed in* $O(\lg(n/K))$ *time if* $K \geqq 1$ *or in* $O(\lceil 1/K \rceil \lg n)$ *time if* $0 < K < 1$.

Proof. To compute F^k, for all $0 \leqq k \leqq n-1$, we proceed in two steps:

1. **for** i: $1 \leqq i \leqq n$ **pardo** $F^0(i) := i$; $F^1(i) := F(i)$ **dopar**;
2. **for** $t := 0$ **to** $\lg(n-1) - 1$ **do**
   for $s: 1 \leqq s \leqq 2^t$, $i: 1 \leqq i \leqq n$ **pardo**
   $F^{2^t+s}(i) := F^{2^t}(F^s(i))$
   dopar;

If nK processors are given, it is clear that step 1 can be computed in $O(\lceil 1/K \rceil)$ time (Lemma 3.1). Step 2 can be computed in

$$\sum_{t=0}^{\lg(n-1)-1} (\lceil 2^t/K \rceil) = \lg K + \sum_{t=\lg K}^{\lg(n-1)-1} (\lceil 2^t/K \rceil)$$

$$< \lg K + 1/K \sum_{t=\lg K}^{\lg(n-1)-1} 2^t + \lg(n-1) - \lg K$$

$$= O(n/K + \lg n) \text{ parallel time units.}$$

Once $F^k(i)$, $1 \leqq i \leqq n$, $0 \leqq k \leqq n-1$, are computed, depth (i), $1 \leqq i \leqq n$, can be found by performing a binary search on the ordered sequence $F^0(i), F^1(i), \cdots, F^{n-1}(i)$, for each i, searching for the left-most occurrence of r_j using $F^{n-1}(i)(=r_j)$ as the key. This takes a total of $\lceil 1/K \rceil \lg n$ time units if $0 < K < 1$. For $K \geqq 1$, the search is performed in the following way: divide the sequence into $\lceil n/K \rceil$ segments, assign one processor to each segment and perform simultaneously a binary search on each segment. After this step, every processor compares the element it finds with the preceding and succeeding elements in the sequence. There is exactly one processor which does not have all the three elements distinct or identical and this processor locates the left-most occurrence of r_j. This takes a total of $\lg \lceil n/K \rceil + 3$ parallel time units. □

The actual computations of $F^k(i)$, $1 \leqq i \leqq n$, $0 \leqq k \leqq n-1$, and depth (i), $1 \leqq i \leqq n$, are performed in an array $\mathbf{F}^+$ in which $\mathbf{F}^+[i, k]$ contains $F^k(i)$. After the computations are finished, each row of $\mathbf{F}^+$ is right shifted so that all the r_j's except the left-most one are eliminated. As a consequence, the right-most column of the array contains only the roots from $\{r_j\}$. Furthermore, for each vertex i, all occurrences of i appear only in column $(n-1)-$depth (i). For each row i, a number, $n+i$, acting as an undefined value, is inserted into the first $(n-1)-$depth (i) entries. These adjustments are done for convenience and not out of necessity and they take $O(n/K)$ time with $nK(K>0)$ processors (Lemma 3.1). The adjusted array, $\mathbf{F}^+$, of a directed tree is depicted in Fig. 4.1. Note that the ith row in $\mathbf{F}^+$ contains the path from vertex i to a root in T.

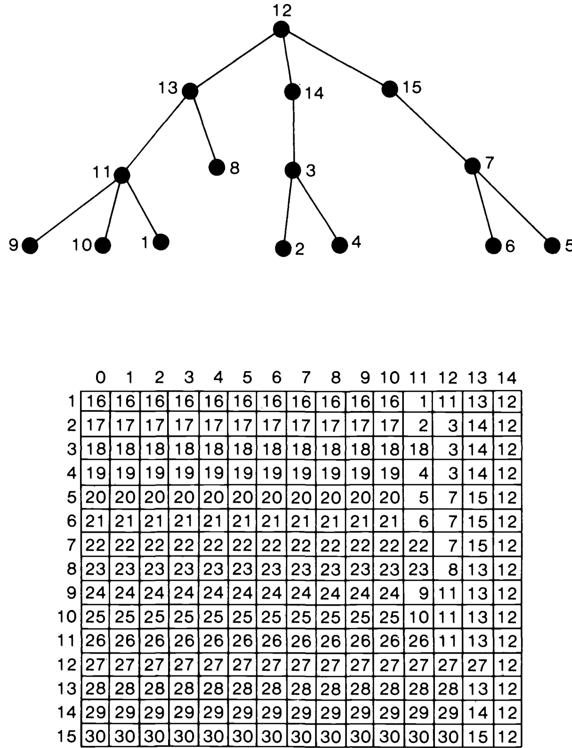
5. Finding a directed spanning forest in an undirected graph. In this section, we present an efficient parallel algorithm for finding a directed spanning forest in an undirected graph $G(V, E)$. In view of the fact that it is the inverted spanning forest of G which is useful in the design of other parallel algorithms in the following sections, the algorithm presented below actually constructs an inverted spanning forest. Nevertheless, converting an inverted spanning forest into a directed spanning forest is straightforward. This algorithm will serve as the backbone of the other algorithms presented in the following sections.

This algorithm is based on the algorithm for finding an undirected spanning forest presented in [3] and the array $\mathbf{F}^+$ presented in the last section. The latter is used to assign a direction to each edge in the undirected spanning forest generated by the former.[2]

We first give a general description for the strategy used in our algorithm. In the course of running the algorithm for finding an undirected spanning forest [3], a number of 1-tree-loop's [7][3] are generated. Each of the 1-tree-loop's is a directed graph whose vertices are supervertices generated during the previous iteration (a *supervertex* is a

[2] We assume the reader is familiar with the undirected spanning forest algorithm. For those who are not, we refer them to reference [3].

[3] A 1-tree-loop is a directed graph in which every vertex has outdegree 1 and in which there is exactly one cycle and the length of the cycle is 2.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	16	16	16	16	16	16	16	16	16	16	16	1	11	13	12
2	17	17	17	17	17	17	17	17	17	17	17	2	3	14	12
3	18	18	18	18	18	18	18	18	18	18	18	18	3	14	12
4	19	19	19	19	19	19	19	19	19	19	19	4	3	14	12
5	20	20	20	20	20	20	20	20	20	20	20	5	7	15	12
6	21	21	21	21	21	21	21	21	21	21	21	6	7	15	12
7	22	22	22	22	22	22	22	22	22	22	22	22	7	15	12
8	23	23	23	23	23	23	23	23	23	23	23	23	8	13	12
9	24	24	24	24	24	24	24	24	24	24	24	9	11	13	12
10	25	25	25	25	25	25	25	25	25	25	25	10	11	13	12
11	26	26	26	26	26	26	26	26	26	26	26	26	11	13	12
12	27	27	27	27	27	27	27	27	27	27	27	27	27	27	12
13	28	28	28	28	28	28	28	28	28	28	28	28	28	13	12
14	29	29	29	29	29	29	29	29	29	29	29	29	29	14	12
15	30	30	30	30	30	30	30	30	30	30	30	30	30	15	12

FIG. 4.1. *A directed tree and its array* $\mathbf{F}^+$. *Note that since* $n = 15$, *any number greater than* 15 *serves as an undefined value in the array.*

vertex in G or a 1-tree-loop). The edges of these 1-tree-loop's will be included in the undirected spanning forest and all these edges are directed edges, whose directions are ignored by the algorithm in [3]. If the only loop in a 1-tree-loop is destroyed by eliminating the out-going edge from the smallest-numbered-vertex, the resulting graph is an inverted tree. As a result, when the loops of all the 1-tree-loop's are destroyed in this way, the resulting graph (built by embedding the modified (acyclic) 1-tree-loop's created during one iteration into the modified (acyclic) 1-tree-loop's created during the following iteration) may well be an inverted spanning forest. Unfortunately, this is not the case in general, because some vertices may end up with two fathers. This situation is depicted in Fig. 5.1, where a directed edge (a, b) is selected during iteration $j+1$ to connect two supervertices S_1 and S_2 created during iteration j. The two graphs resulting from the two supervertices are inverted trees. However, since a is not the root r_1 of S_1, a will have two fathers after S_1 and S_2 have been included into a single supervertex. Therefore, the graph $S_1 \cup S_2$ is not an inverted tree, by definition, unless the directions of all the edges on the path from a to r_1 are reversed. The same situation occurs in $S_2 \cup S_3$ when the directed edge (c, d) is selected to connect S_2 and S_3. To overcome this difficulty, we have to reverse the directions of all edges on the path from a to r_1 and those on the path from c to r_2. The array $\mathbf{F}^+$, described in § 4, contains the path from any vertex to a root in an inverted forest T; hence we can generate the array $\mathbf{F}^+$ covering both S_1 and S_2. By retrieving the ath row and the cth row of $\mathbf{F}^+$, we can identify the set of all edges whose directions are to be reversed in S_1 and S_2 respectively.

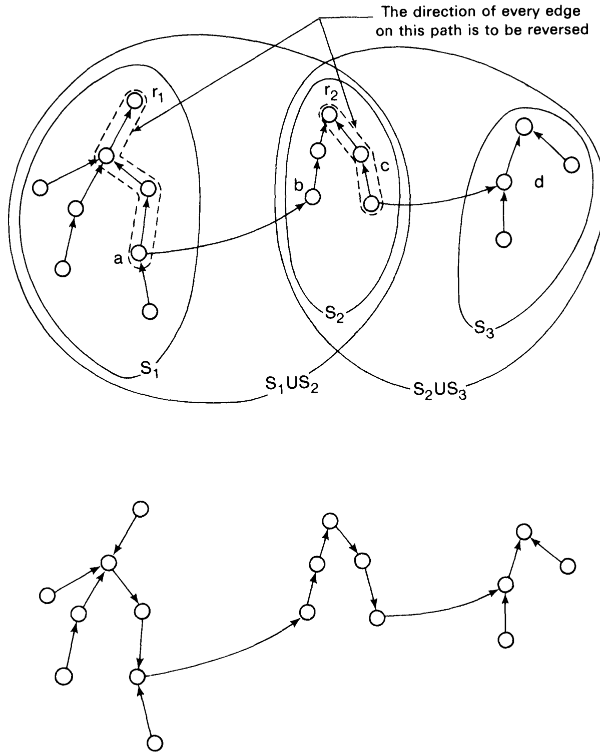


FIG. 5.1.

Our algorithm runs in two stages.

ALGORITHM DSF.

Stage 1 (∗ The first stage is basically a modified version of the algorithm for finding an undirected spanning forest. We refer the reader to reference [3] for the details.∗)

Execute the algorithm for finding an undirected spanning tree; during each iteration j, $1 \leqq j \leqq \lg n$, record the following information:

a. Convert the forest of all 1-tree-loops generated during this iteration into a forest of inverted trees by eliminating the edge from the smallest-numbered-vertex of each 1-tree-loop and store the forest in a vector F_j. (∗ Note: This vector acts as the function F defined in § 4.∗)

b. Record the "actual" edges in G establishing the connection specified in F_j. (∗ Note: The edges recorded in F_j are *pseudo* edges which connect "supervertices". They do not exist in G. However, for each pseudo edge, there exists a corresponding actual edge in G.*)

c. The vector $D[1 \,..\, n]$ generated during this iteration is stored as D_j. (∗ Note: $D_j[v]$ is the supervertex containing vertex v when iteration j is completed.*)

Stage 2

1. Generate $\mathbf{F}_j^+$'s from F_j, $1 \leqq j \leqq \lg n$.
2. (∗ Adjust the directions of the edges, starting from those recorded during iteration $\lg n$, gradually down to those recorded during interation 1. ∗)

$R' := \{v \in V | D_{\lg n}[v] = v\}$;
(* Note: In the following **for** loop, R' contains the tails of those actual edges in G which connect two supervertices in the inverted trees generated during iteration i, where $j < i \leqq \lg n$. It includes all those vertices which have two or more fathers in the directed graph formed upon the inverted trees. *)
for $j := \lg n$ **downto** 1 **do**
begin
i) For every $r' \in R'$,
reverse the direction of every "pseudo" edge lying on the path from the supervertex $D_j[r']$ to the root of the inverted tree, in F_j, containing $D_j[r']$;
ii) Output all the "actual edges" in G corresponding to the pseudo edges in F_j;
iii) $R' = R' \cup \{v \in V | v$ is the tail of an "actual" edge output in step ii)$\}$
end;

A complete example is given in Fig. 5.2 and a detailed implementation using the method described above is given in the Appendix.

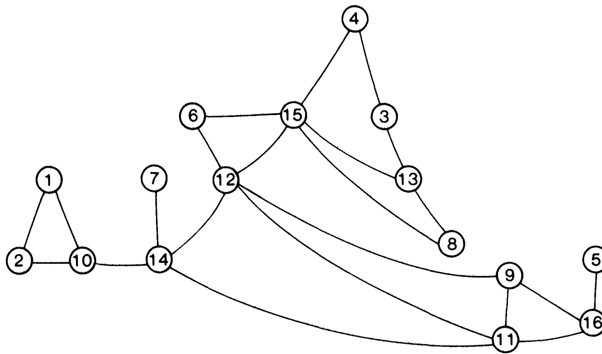


FIG. 5.2(i) $G(V, E)$.

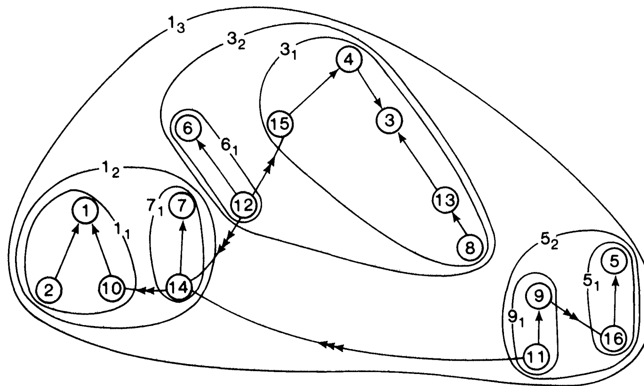


FIG. 5.2(ii). *A potential inverted spanning tree of G.* → *a directed edge selected during the first iteration*; →→ *a directed edge selected during the second iteration*; →→→ *a directed edge selected during the third iteration.*

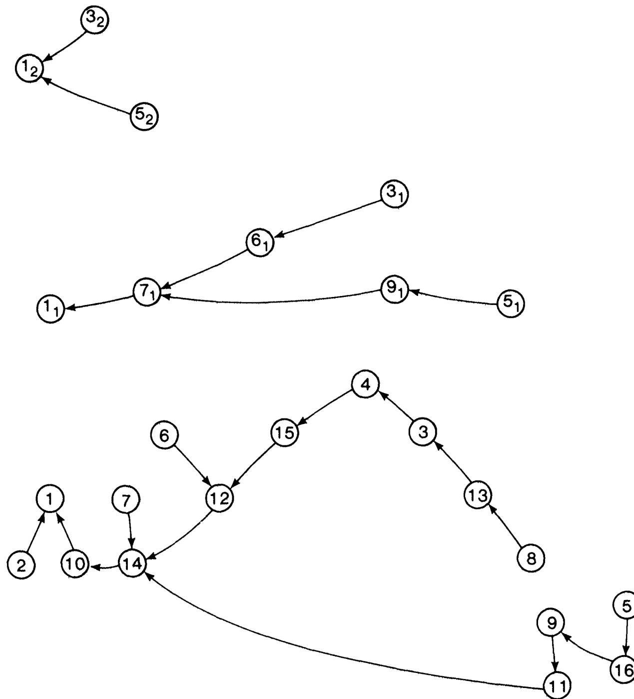


FIG 5.2(iii). *An inverted spanning tree of G.*

THEOREM 5.1. *Algorithm* DSF *correctly generates an inverted spanning forest for an undirected graph.*

Proof. (Backward induction.) In stage 1, an inverted forest F_j is correctly generated during each iteration j, $1 \leqq j \leqq \lg n$[3]. In stage 2, supposing that after processing F_i, $j \leqq i \leqq \lg n$, an inverted forest F'_j is created. Clearly, F'_j and F_j must have the same vertex set V_j. When processing F_{j-1}, it should be clear that there exists a one-to-one correspondence between the vertices in V_j and the inverted trees in F_{j-1}. This implies that no two instances of r' in R' will belong to the same inverted tree in F_{j-1}. As a result, after step i, each inverted tree in F_{j-1} is effectively modified so as to root at the supervertex $D_{j-1}[r']$. These modified inverted trees are then embedded into the inverted forest F'_j in step 2ii), the resulting directed graph F'_{j-1} is clearly an inverted forest. But $F'_{\lg n} = F_{\lg n}$ is an inverted forest initially, therefore, by induction, F'_1 must be an inverted forest and hence an inverted spanning forest for G. □

THEOREM 5.2. *Finding an inverted spanning forest takes $O(n/K + \lg^2 n)$ time with nK $(K \geqq 1)$ processors.*

Proof. Stage 1 takes $O(n/K + \lg^2 n)$ time with $nK(K \geqq 1)$ processors [3]. Since the total number of edges in the inverted forest is at most

$$\sum_{j=1}^{\lg n} \lfloor n/2^{j-1} \rfloor < 2n,$$

the creation of $\mathbf{F}_j^+$, $1 \leqq j \leqq \lg n$, in step 1 of stage 2 can be done in $O(n/K + \lg n)$ time with $nK(K \geqq 1)$ processors. Steps 2ii) and iii) each takes $O(1)$ time for each iteration.

Since the size of $\mathbf{F}_j^+$, $1 \leqq j \leqq \lg n$, is $\lceil n/2^{j-1} \rceil \times \lceil n/2^{j-1} \rceil$, step 2i) requires

$$\sum_{j=1}^{\lg n} \lceil \lceil n/2^{j-1} \rceil^2 / nK \rceil < \sum_{j=1}^{\lg n} \lceil n/2^{j-1} \rceil^2 / nK + \lg n$$
$$= O(n/K + \lg n) \quad \text{time for } \lg n \text{ iterations.}$$

Hence the theorem. □

Note that the processor-time product is $O(n^2)$, when $1 \leqq K \leqq \lceil n/\lg^2 n \rceil$, the algorithm is thus optimal for dense graphs.

6. Finding the lowest common ancestors of q vertex pairs in a directed tree. Let $T(V', E')$ be a directed tree and $V' = \{1, 2, \cdots, n\}$. We shall make use of the array $\mathbf{F}^+$ to design a parallel algorithm for finding the lowest common ancestors of q vertex pairs in T. Let a and b be a vertex pair, if c is their lowest common ancestor, then row a and row b of $\mathbf{F}^+$ will have identical contents between column $(n-1) - \text{depth}[c]$ and column $n-1$, inclusive, and will have different contents in the other columns. As a result, to determine c, we can perform a binary search on row a and row b simultaneously in the following way: if the two entries being examined in row a and row b (in the same column, of course) are different, the search is continued on the right-half, otherwise it is continued on the left-half. It takes $\lg n + 1$ time units to find c with one processor. In general, we have:

THEOREM 6.1. *Given q vertex pairs, $1 \leqq q \leqq n^2$, finding the lowest common ancestors for these vertex pairs takes $O(\lceil q/nK \rceil \cdot \lg n + n/K)$ time if $nK\,(K > 0)$ processors are available.*

Proof. Finding the lowest common ancestors of the q vertex pairs takes $\lceil q/nK \rceil \cdot \lg n$ time units, if $nK \leqq q \leqq n^2$ (Lemma 3.1) or $\lg n + 1$ time units, if $nK > q$. Constructing the array $\mathbf{F}^+$ takes $O(n/K + \lg n)$ time, thus finding the lowest common ancestors of $q\,(1 \leqq q \leqq n^2)$ vertex pairs, takes $O(\lceil q/nK \rceil \cdot \lg n + n/K)$ time with nK $(K > 0)$ processors. □

In particular, when $K = n$ and $\lceil n/\lg n \rceil$, this algorithm takes $O(\lg n)$ and $O(\lg^2 n)$ time, respectively.

7. Finding all fundamental cycles of a connected, undirected graph. Without loss of generality, we assume that the undirected graph $G(V, E)$ is connected from this section onwards.

It is known that a set of fundamental cycles of a connected, undirected graph $G(V, E)$ can be determined from a spanning tree $T(V, E')$ of G [14]. Specifically, if (a, b) is an edge in $G - T$, then (a, b) together with the paths $[b \ast\!\!\rightarrow \text{LCA}\,(a, b)]$ and $[\text{LCA}\,(a, b) \ast\!\!\rightarrow a]$ form a fundamental cycle.

Based on the above observation, we can find a set of fundamental cycles of G as follows: First, an inverted spanning tree T of G is found, using the algorithm presented in § 5, which takes $O(n/K + \lg^2 n)$ time with $nK\,(K \geqq 1)$ processors. The lowest common ancestor algorithm is then called to determine the lowest common ancestor for every pair of vertices (a, b) in $G - T$. The algorithm returns the ordered pair $(\mathbf{LCA}^+, \mathbf{F}^+)$ and the vector depth, where $\mathbf{LCA}^+[a, b]$ contains the lowest common ancestor of (a, b). A vector $\mathbf{P}^+$ is then created such that $\mathbf{P}^+[v]$ contains the value $(n-1) - \text{depth}\,[v]$, which is the column number of v in $\mathbf{F}^+$. Hence, for each (a, b) in $G - T$, the path from column $\mathbf{P}^+[a]$ to column $\mathbf{P}^+[\mathbf{LCA}^+[a, b]]$ in row a and the path from column $P^+[b]$ to column $\mathbf{P}^+[\mathbf{LCA}^+[a, b]]$ in row b of $\mathbf{F}^+$ and the edge (a, b) determine a fundamental cycle in G.

The correctness of the algorithm is easily verified. Since the number of vertex pairs $q = |E - E'|$, the algorithm obviously takes $O(\lceil |E|/nK \rceil \cdot \lg n + n/K + \lg^2 n)$ time with nK ($K \geqq 1$) processors. In particular, the $O(\lg^2 n)$ time bound is achieved with $K = n/\lg n$. Note that the output of the algorithm is stored in an $O(n^2)$ compact data structure, which consists of the triple $(\mathbf{P}^+, \mathbf{LCA}^+, \mathbf{F}^+)$.

8. Finding the HLCA(u)'s. The algorithms we present in the following sections rely heavily on the function, HLCA(u), $\forall u \in V$, (note: The prefix H stands for highest), which is defined as follows.

DEFINITION. Let $G(V, E)$ be an undirected graph, $T(V, E')$ be its inverted spanning tree and $u \in V$. HLCA $(u) =$ LCA (u, v), where $(u, v) \in E - E' \cup \{(u, u)\}$ and depth (LCA $(u, v)) \leqq$ depth (LCA (u, v')), $\forall (u, v') \in E - E' \cup \{(u, u)\}$.

Figure 8.1 gives an illustration of HLCA (u). The solid lines and circles represent the edges and vertices of an inverted spanning tree of an undirected graph. The dotted lines represent the edges in the graph $G - T$ emanating from a particular vertex u. To compute HLCA (u), $\forall u \in V$, we may first use the lowest common ancestor algorithm to find LCA (u, v), $\forall (u, v) \in E - E' \cup \{(u, u)\}$ and then apply Lemma 3.2 to find HLCA (u), $\forall u \in V$. However, in doing so, we will require $O(\lceil |E - E'|/nK \rceil \cdot \lg n + n/K)$ time if $nK (K > 0)$ processors are available. In this section, we show a way of finding HLCA (u), $\forall u \in V$ in $O(n/K + \lg n \lg \lg n)$ time with nK ($K \geqq 1$) processors. This method allows us to design optimal parallel algorithms for the graph theoretic problems discussed in the following sections.

The method is based on the preorder numbering [9] of the vertices in an ordered spanning tree $T(V, E')$ of G. We denote the preorder number of a vertex v by pre (v).

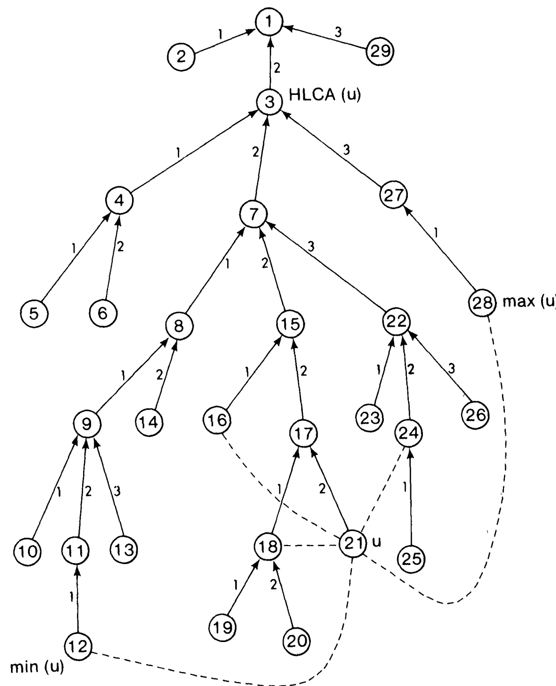


FIG. 8.1.

DEFINITION. *Let* $u, v \in V$, $u \preceq v$ iff u is an ancestor of v, $u \prec v$ iff u is a proper ancestor of v.

LEMMA 8.1. *Let* $u, v \in V$, $v \preceq u$ *iff* $\text{pre}(v) \leqq \text{pre}(u) < \text{pre}(v) + nd(v)$, *where* $nd(v)$ *is the number of descendants of* v.

Proof. Immediate from the definition of preorder traversal. □

LEMMA 8.2. *Let* $(u, v), (u, w) \in E - E'$;

(i) *if* $\text{pre}(v) < \text{pre}(w) < \text{pre}(u)$,
then $\text{depth}(\text{LCA}(u, v)) \leqq \text{depth}(\text{LCA}(u, w))$;

(ii) *if* $\text{pre}(v) > \text{pre}(w) > \text{pre}(u)$,
then $\text{depth}(\text{LCA}(u, v)) \leqq \text{depth}(\text{LCA}(u, w))$.

Proof. (i) By Lemma 8.1, $\text{pre}(\text{LCA}(u, v)) \leqq \text{pre}(v)$ and $\text{pre}(u) < \text{pre}(\text{LCA}(u, v)) + nd(\text{LCA}(u, v))$. Therefore $\text{pre}(\text{LCA}(u, v)) < \text{pre}(w) < \text{pre}(\text{LCA}(u, v)) + nd(\text{LCA}(u, v))$. By Lemma 8.1, $\text{LCA}(u, v) \preceq w$. Hence, $\text{depth}(\text{LCA}(u, v)) \leqq \text{depth}(\text{LCA}(u, w))$. Part (ii) can be proved similarly. □

Lemma 8.2 points out that we can reduce the problem of finding $\text{HLCA}(u)$ to that of finding the lowest common ancestor of two particular vertices in $\{v \mid (u, v) \in E - E'\} \cup \{u\}$.

DEFINITION. Let $u \in V$, $W = \{v \mid (u, v) \in E - E'\} \cup \{u\}$.

$$\text{pmax}(u) = v, \quad \text{where } v \in W \text{ and } \text{pre}(v) \geqq \text{pre}(w), \quad \forall w \in W;$$

$$\text{pmin}(u) = v, \quad \text{where } v \in W \text{ and } \text{pre}(v) \leqq \text{pre}(w), \quad \forall w \in W.$$

COROLLARY 8.3. $\text{HLCA}(u) = (\min \preceq)\{\text{LCA}(u, \text{pmin}(u)), \text{LCA}(u, \text{pmax}(u))\}$.

Proof. Immediate from Lemma 8.2. □

COROLLARY 8.4. $\text{HLCA}(u) = \text{LCA}(\text{pmin}(u), \text{pmax}(u))$.

Proof. From Corollary 8.3, $\text{HLCA}(u) \preceq \text{pmin}(u)$ and $\text{HLCA}(u) \preceq \text{pmax}(u)$. Thus, $\text{HLCA}(u) \preceq \text{LCA}(\text{pmin}(u), \text{pmax}(u))$. By definition, $\text{pre}(\text{pmin}(u)) \leqq \text{pre}(u) \leqq \text{pre}(\text{pmax}(u))$. This implies $\text{pre}(\text{LCA}(\text{pmin}(u), \text{pmax}(u))) \leqq \text{pre}(u) < \text{pre}(\text{LCA}(\text{pmin}(u), \text{pmax}(u))) + nd(\text{LCA}(\text{pmin}(u), \text{pmax}(u)))$. By Lemma 8.1, $\text{LCA}(\text{pmin}(u), \text{pmax}(u)) \preceq u$. Therefore $\text{LCA}(\text{pmin}(u), \text{pmax}(u)) \preceq \text{LCA}(u, \text{pmin}(u))$ and $\text{LCA}(\text{pmin}(u), \text{pmax}(u)) \preceq \text{LCA}(u, \text{pmax}(u))$. By Corollary 8.3, $\text{LCA}(\text{pmin}(u), \text{pmax}(u)) \preceq \text{HLCA}(u)$. □

LEMMA 8.5. *Let* $T(V, E')$ *be a directed tree whose vertices have been labelled in preorder. Then finding* $\text{HLCA}(u)$, $\forall u \in V$, *can be done in* $O(n/K + \lg n)$ *time with* $nK (K \geqq 1)$ *processors.*

Proof. To compute $\text{pmax}(u)$ and $\text{pmin}(u)$, $\forall u \in V$, we need $O(n/K + \lg K)$ time with nK $(K \geqq 1)$ processors (Lemma 3.2), and to find $\text{HLCA}(u)$, $\forall u \in V$, we need to find the lowest common ancestors of the n $(\text{pmin}(u), \text{pmax}(u))$ pairs. This takes $O(n/K + \lg n)$ time with $nK (K > 0)$ processors (Theorem 6.1). □

Figure 8.1. gives an illustration of the above lemmas and corollaries. The numbers in the circles are the preorder numbers of the vertices. For instance, the preorder number of u is 21. For convenience sake, we name each vertex by its preorder number. It can be easily checked that $\text{depth}(\text{LCA}(u, 12)) < \min(\text{depth}(\text{LCA}(u, 18)), \text{depth}(\text{LCA}(u, 16)))$, and that $\text{depth}(\text{LCA}(u, 28)) < \text{depth}(\text{LCA}(u, 24))$. Furthermore, $\text{pmin}(u) = 12$, $\text{pmax}(u) = 28$, and $\text{LCA}(12, 28) = 3$, which is clearly $\text{HLCA}(u)$.

The crucial step in computing $\text{HLCA}(u)$, $\forall u \in V$, is to determine the preorder numbers efficiently. The usual way of numbering the vertices of a tree in preorder is to traverse the tree. However, this will result in an $O(n)$ time algorithm, which is undesirable. In the following lemma, we show that we can carry out preorder numbering in parallel without traversing the tree.

LEMMA 8.6. *Let $T(V, E')$ be an ordered tree* [9]. *For each $v \in V$,*

$$\text{pre}(v) = \sum_{u \in \text{ANC}(v)} \sum_{w \in \text{EBRO}(u)} nd(w) + na(v)$$

$$= \sum_{u \in \text{ANC}(v)-\{r\}} nds(F(u), \text{rank}(u)-1) + 1 + \text{depth}(v),$$

where

ANC (v) *is the set of all ancestors of* v;
EBRO(u) *is the set of all elder brothers of* u;
$nd(w)$ *is the number of descendants of* w;
$na(v)$ *is the number of ancestors of* v.
$nds(v, j)$ *is the total number of descendants of the first* j *sons of* v; *and*
rank (v)*is the rank of* v, *i.e., the position of* v *among all its brothers.*

Proof. Trivial. □

Let us consider the inverted spanning tree given in Fig. 8.1. again. Consider the vertex u, pre $(u) = 21$, the ancestors of u are the vertices 21, 17, 15, 7, 3 and 1. The number of descendants of the elder brothers of each of these vertices except the root are 3, 1, 7, 3 and 1, respectively. These numbers sum up to 15. The number of ancestors of u is 6, this gives rise to a total sum of 21, which is the preorder number of u.

Using Lemma 8.6, we want to show that the preorder numbers pre (v), $\forall v \in V$ can be determined in $O(n/K + \lg n \lg\lg n)$ time with $nK (K \geqq 1)$ processors. Assuming that an inverted tree T represented by an array $T[1 \ldots 2, 1 \ldots n]$ such that $\{(T[1, i], T[2, i]) | 1 \leqq i \leqq n\} = E'$ is given (we assume $T[2, r] = 0$ for the root r).

ALGORITHM Preorder.
1. Compute the array $\mathbf{F}^+$ and the vector *depth* for T.
2. Order the sons of every vertex in T, i.e., compute rank (v), $\forall v \in V$.
3. Find $nds(v, j)$, $\forall v \in V$, $1 \leqq j \leqq n(v)$, where $n(v)$ is the number of sons of v.
4. Compute pre (v), $\forall v \in V$.

LEMMA 8.7. *Algorithm Preorder takes $O(n/K + \lg n \lg\lg n)$ time with nK $(K \geqq 1)$ processors.*

Proof. Step 1 can be done in $O(n/K + \lg n)$ time (Theorem 4.2). In step 2, the ordered pairs $\{(T[2, i], T[1, i]) | 1 \leqq i \leqq n\}$ are sorted. This can be done in $O(\lg n \lg\lg n)$ time with n processor [1]. (* In fact, for $K \geqq \lg n$, we can sort n elements in $O(\lg n)$ time [1]. However, the $O(\lg n \lg\lg n)$ time suffices for our purposes here *). Assuming that the sorted T is stored in $T'[1 \ldots 2, 1 \ldots n]$, then T' is divided into segments such that in each segment, the first row contains the same vertex v in every entry, and the second row contains the set of all sons of v in T. The relative position of vertex i in the second row of the segment in which i resides, is the rank of i, i.e., rank (i).

In step 3, $nd(v)$, $\forall v \in V$, are first computed by scanning the $((n-1) - \text{depth}(v))$th column of $\mathbf{F}^+$ and counting the number of occurrences of v. By Lemma 3.2, this takes $O(n/K + \lg K)$ time. After this, $nds(v, j)$, $\forall v \in V$, $1 \leqq j \leqq n(v)$, are computed using the following formula

$$nds(v, j) = \sum_{1 \leqq i \leqq j} nd(s_i), \qquad 1 \leqq j \leqq n(v).$$

It has been shown in [10], that the partial sums $\sum_{1 \leqq i \leqq j} a_i$, $1 \leqq j \leqq n$, can be computed in $O(\lg n)$ time if n processors are given. Since for each vertex v, v has $n(v)$ sons,

the time needed to compute $nds(v, j)$, $1 \leqq j \leqq n(v)$, is $O(\lg(n(v)))$ if $n(v)$ processors are assigned to v. (This is possible if we make use of the sorted array T'). As a result, all these partial sums, $nds(v, j)$, $1 \leqq j \leqq n(v)$, $\forall v \in V$, can be computed in parallel in $\max_{v \in V} \{O(\lg(n(v)))\} = O(\lg n)$ time with $\sum_{v \in V} n(v) = n-1$ processors.

Finally, in step 4, pre (v), $\forall v \in V$ is computed using the formula given in Lemma 8.6. We assume $nds(v, 0) = 0$, $\forall v \in V$. Note that ANC (v) is available in the vth row of $\mathbf{F}^+$ starting from column $(n-1) - \text{depth}(v)$ to column $(n-1)$, and $na(v)$ equals depth $(v)+1$. By Lemma 3.2, this takes $O(n/K + \lg K)$ time.

Summing up, pre (v), $\forall v \in V$ can be determined in $O(n/K + \lg n \lg\lg n)$ time with $nK(K \geqq 1)$ processors □

THEOREM 8.8. *Computing* HLCA (u), $\forall u \in V$ *can be done in* $O(n/K + \lg n \lg\lg n)$ *time with* nK $(K \geqq 1)$ *processors.*

Proof. Lemmas 8.5, 8.7. □

9. Finding all bridges in a connected, undirected graph. In this section, we present an optimal parallel algorithm for finding all bridges in a connected, undirected graph. The correctness of the algorithm is based on the following theorems.

LEMMA 9.1. *Let $G(V, E)$ be a connected, undirected graph. If $e \in E$ is a bridge of G, then e is contained in every inverted spanning tree of G.*

Proof. Trivial. □

Due to this lemma, the number of edges to be examined is greatly reduced from $O(n^2)$ to $O(n)$.

LEMMA 9.2. *e is not a bridge if and only if e is on a fundamental cycle.*

Proof. Trivial. □

THEOREM 9.3. *Let $T(V, E')$ be an inverted spanning tree of a connected, undirected graph G, and $e = (a, b) \in E'$. Then (a, b) is a bridge of G if and only if for each descendant i of a, there does not exist (i, j) in $G - T$ such that* depth (LCA $[i, j]$) < depth (a).

Proof. Let $e = (a, b) \in E'$ be a bridge in G. If there exists (i, j) in $G - T$ such that i is a descendant of a in T and depth (LCA $[i, j]$) < depth (a), then the path $[i \rightarrow j \ast\!\!\rightarrow \text{LCA}[i, j] \ast\!\!\rightarrow b \rightarrow a \ast\!\!\rightarrow i]$ is a cycle containing e. This leads to a contradiction by Lemma 9.2.

Conversely, if e is not a bridge, then by Lemma 9.2, e is on a fundamental cycle C, i.e., there exists (i, j) in $G - T$ such that

$$C: [i \rightarrow j \ast\!\!\rightarrow \text{LCA}[i, j] \ast\!\!\rightarrow i].$$

$e \neq (i, j)$ because e is not in $G - T$. As a result, e is either on the path $[j \rightarrow \text{LCA}[i, j]]$ or on the path $[\text{LCA}[i, j] \rightarrow i]$, implying depth $(j) \geqq$ depth $(a) >$ depth $(b) \geqq$ depth (LCA $[i, j]$) or depth $(i) \geqq$ depth $(a) >$ depth $(b) \geqq$ depth (LCA $[i, j]$). Hence in either case there exists (i, j) in $G - T$ such that i is a descendent of a and depth (LCA $[i, j]$) < depth (a). □

ALGORITHM Bridges

1. Construct an inverted spanning tree $T(V, E')$ for $G(V, E)$.
2. Compute HLCA (u), $\forall u \in V$.
3. Compute $\alpha(u)$, $\forall u \in V$, where

$$\alpha(u) = \min \{\text{depth}(\text{HLCA}(w)) \mid u \leq w\}.$$

4. For each $(u, F(u)) \in E'$, check if depth $(u) \leqq \alpha(u)$. (*$(u, F(u))$ is a bridge iff depth $(u) \leqq \alpha(u)$*)

The correctness of the algorithm is supported by Theorem 9.3.

THEOREM 9.4. *Algorithm Bridges runs in* $O(n/K + \lg^2 n)$ *time with* $nK (K \geqq 1)$ *processors.*

Proof. With nK ($K \geqq 1$) processors, step 1 takes $O(n/K + \lg^2 n)$ time (Theorem 5.2). Step 2 takes $O(n/K + \lg n \lg\lg n)$ time (Theorem 8.8). Steps 3 and 4 take $O(n/K + \lg K)$ time (Lemma 3.1 and 3.2). Hence, Algorithm Bridges runs in $O(n/K + \lg^2 n)$ time with $nK (K \geqq 1)$ processors. □

10. The bridge-connected components of a connected, undirected graph. Once the bridges of a connected, undirected graph are determined, its bridge-connected components can be determined. Specifically, we eliminate all the bridges in G and then use Algorithm MOD.CONNECT [3], [4] to find the connected components of the resulting graph. Each of the connected components thus found is a bridge-connected component of G.

The algorithm obviously runs in $O(n/K + \lg^2 n)$ time with $nK (K \geqq 1)$ processors.

11. Finding all biconnected components in a connected, undirected graph. In this section, we present an optimal parallel algorithm for finding all biconnected components of a connected, undirected graph $G(V, E)$. Since a biconnected component can be completely determined by its vertex set, it suffices to find the vertex sets of all the biconnected components of G.

DEFINITION. Let $T(V, E')$ be an inverted spanning tree of $G(V, E)$. Let $e_1 = (a, F(a))$, $e_2 = (b, F(b)) \in E'$. $e_1 \Delta e_2$ iff

(i) e_2 is on $[a * \rightarrow \text{HLCA}(a)]$ or e_1 is on $[b * \rightarrow \text{HCLA}(b)]$; or

(ii) $(a, b) \in E - E'$ and neither $a \leq b$ nor $b \leq a$ in T.

ALGORITHM Biconnect.

1. Find an inverted spanning tree $T(V, E')$ of $G(V, E)$.
2. Compute HLCA (v) $\forall v \in V$.
3. Construct an undirected graph $G''(E', E'')$ such that $(e_1, e_2) \in E''$ iff $e_1 \Delta e_2$.
4. Find the connected components $\{B_j\}$ of G''. (* Note: Every connected component of G'' uniquely determines the vertex set of a biconnected component in G and vice versa. *)

LEMMA 11.1. (i) *For each edge* $(a, b) \in E$ *there exists a unique biconnected component in G containing the edge.*

(ii) *All edges in the same cycle in G belong to the same biconnected component in G.*

From the definition, if $e_1 \Delta e_2$ then e_1 and e_2 belong to the same fundamental cycle. It is easily shown that if $e_1 \Delta e_2$ and $e_2 \Delta e_3$, then e_1 and e_3 belong to the same cycle in G. This is easily generalized to:

LEMMA 11.2. *If* $e_1 \Delta e_2, e_2 \Delta e_3, \cdots, e_{l-1} \Delta e_l$, *then there exists a cycle in G containing both* e_1 *and* e_l.

THEOREM 11.3. e *and* e' *belong to the same connected component in* G'' *if and only if* e *and* e' *belong to the same biconnected component in G.*

Proof. Let e and e' belong to the same connected component in G''. Then there exists a path: $e, e_1, \cdots, e_l, e'$ in G''. This implies that $e \Delta e_1, e_1 \Delta e_2, \cdots, e_l \Delta e'$. By Lemma 11.2, e, e' belong to the same cycle in G. By Lemma 11.1 (ii), e and e' belong to the same biconnected component in G.

Let e and e' belong to the same biconnected component in G. Then there exists a simple cycle C containing e and e' in G. Let $\mathscr{C}$ be the set of fundamental cycles such that $C = \cup_+ \mathscr{C}$ ($\cup_+$ stands for the mod-two sum). It is easily shown that there exists a subset $\{C_i\}_{1 \leqq i \leqq l}$ of $\mathscr{C}$ such that $e \in C_1$, $e' \in C_l$ and e_i is a common edge of C_i and C_{i+1},

$1 \leqq i < l$. Let (a_i, b_i) be the edge in $G-T$ determining C_i, $1 \leqq i \leqq l$. Let $e(a_i)$, $e(b_i)$ be the edges in T such that $e(a_i) = (a_i, F(a_i))$ and $e(b_i) = (b_i, F(b_i))$; then in each C_i, we have: (i) $e(a_i) \Delta e(b_i)$ and ($e_{i-1} \Delta e(a_i)$ or $e_{i-1} \Delta e(b_i)$) and ($e_i \Delta e(a_i)$ or $e_i \Delta e(b_i)$); or (ii) $e_{i-1} \Delta\ e(a_i)$ and $e_i \Delta e(a_i)$; or (iii) $e_{i-1} \Delta e(b_i)$ and $e_i \Delta e(b_i)$. In any of the above cases, there is a path from e_{i-1} to e_i in G''. In particular, there is a path from e to e_1 and a path from e_{l-1} to e' in G''. Joining all these paths together, we have a path from e to e' in G''. Hence, e and e' belong to the same connected component in G''. □

LEMMA 11.4. *Algorithm Biconnect runs in* $O(n/K + \lg^2 n)$ *time with* $nK (K \geqq 1)$ *processors.*

Proof. With $nK (K \geqq 1)$ processors available, step 1 takes $O(n/K + \lg^2 n)$ time (Theorem 5.2). Step 2 takes $O(n/K + \lg n \lg\lg n)$ time (Theorem 8.8). Step 3 can be carried out as follows: Construct an adjacency matrix M'' for G''. For every $e \in E'$, $M''[e, e']$ and $M''[e', e]$ are set to 1 if and only if (i) e' is on the path $[a \ast\!\!\rightarrow \text{HLCA}(a)]$ or (ii) (a, b) is in $G-T$ and neither $a \leqq b$ nor $b \leqq a$ in T, where $e = (a, F(a))$ and $e' = (b, F(b))$. Due to $|E'| = O(n)$ and the availability of $\mathbf{F}^+$, testing the above conditions takes $O(n/K)$ time with $nK (K \geqq 1)$ processors (Lemma 3.1). Step 4 takes $O(n/K + \lg^2 n)$ time [3], [4]. Hence, Algorithm Biconnect takes $O(n/K + \lg^2 n)$ time with $nK (K \geqq 1)$ processors. □

12. Finding all separation vertices and determining the biconnectivity of a connected, undirected graph. It is easily verified that if a is not the root r of T, then a is a separation vertex of G if and only if a is the root of $T \cap B_j$ for some j where B_j is a biconnected component of G and that r is a separation vertex if and only if r is the root of at least two distinct $T \cap B'$, $T \cap B''$. As a result, the algorithm for finding the biconnected components can be used to determine the set of all separation vertices of G as follows.

THEOREM 12.1. *The set of separation vertices can be found in* $O(n/K + \lg^2 n)$ *time with* $nK (K \geqq 1)$ *processors.*

Proof. First, the set of all biconnected components is determined. This takes $O(n/K + \lg^2 n)$ time with $nK (K \geqq 1)$ processors (Theorem 11.4). Next, the head of each $e \in E'$, head(e), is determined. This obviously takes $O(1)$ time with nK processors. Then the set of all head(e)'s are divided into groups such that those e's belonging to the same biconnected component have their head(e)'s grouped together. This involves sorting and takes $O(\lg n \lg\lg n)$ time with n processors [1]. Finally, the head(e) with the smallest depth in each group is selected, these head(e)'s form the set of separation vertices. r is included in the set if and only if r is selected from two or more groups. This takes $O(n/K + \lg K)$ time with nK processors (Lemma 3.2). □

To determine the biconnectivity of a connected, undirected graph G, we can check the numbers of separation vertices it has. Clearly, G is biconnected if and only if there are no separation vertices. This takes $O(n/K + \lg^2 n)$ time with $nK (K \geqq 1)$ processors.

For completeness, we would like to point out that the algorithm for finding all biconnected components can be used to determine the set of all bridges as well. This is based on the fact that an edge e of G is a bridge if and only if e is a biconnected component of G.

13. Conclusions. The parallel algorithms presented in this paper are optimal for dense graphs except for the problem of finding the lowest common ancestor of vertex pairs in a directed tree and the problem of finding all fundamental cycles in an undirected graph. If an optimal algorithm for finding the lowest common ancestors running in $O(n+q)/nK$ time with $nK (K \geqq 1)$ processors is found, then the algorithm

for finding the fundamental cycles presented in this paper is also improved without any modification. Moreover, this achievement will provide us with an alternate efficient way to compute HLCA (v), $\forall v \in V$ which is crucial in the design of optimal parallel algorithms for the last five problems.

The optimality of our parallel algorithms may suggest that optimal sequential algorithms can be derived from them. As a matter of fact, it has been shown that $O(|V|+|E|)$ time and space sequential algorithms for finding the bridges and biconnected components can be derived [18].

Of all the algorithms presented in this paper, only the algorithm for the lowest common ancestor problem achieves the $O(\lg n)$ time bound. It is therefore intriguing to consider whether there exist $O(\lg n)$ time algorithms for the remaining problems on our SIMD model. No one has yet proven that the $O(\lg^2 n)$ time is a lower bound and this time bound seems unlikely to be surmounted. The difficulty seems to arise from the model we use. In fact, Shiloach and Vishkin [17] have conjectured that the $O(\lg^2 n)$ time bound cannot be breached with a polynomial number of processors on our SIMD model. Recently, Reif managed to design $O(\lg n)$ time probabilistic algorithms for this class of problems [13]. His probabilistic algorithms can be converted into $O(\lg n)$ time parallel algorithms. The resulting algorithms are, however, *nonuniform* in the sense that a different program is needed for each n. Another problem of immediate interest is whether there exist parallel algorithms which are optimal for both dense and sparse graphs. Specifically, they achieve the $O(\lg^2 n)$ time bound using $\lceil m/\lg^2 n \rceil$ processors where m is the number of edges of the given graph.

Finally, we shall point out that although we assume nK, the number of processors available, satisfies the condition $K \geqq 1$ throughout this paper, it is not difficult to extend our results to cases where $0<K<1$ if Brent's theorem [19] is used.

Appendix.

ALGORITHM **DSF** (*To find an inverted spanning forest in an undirected graph *)

Stage 1

```
{Variable declarations}
M; array[1..n, 1..n] of 0..1;
FR+: array[1..2n-1, 0..n-1] of 1..n lg n;
depth: array[1..2n-1] of 0..n-1;
PTR: array[1..n lg n] of 1..2n-1;
DV: array[0..lg n, 1..n] of 1..n;
rootv: array[1..2n-1] of 1..n;
B: array[1..2, 1..n, 1..n] of 1..n;
flag: array[1..n] of 0..1;
D, C: array[1..n] of 1..n;
phase: 1..lg n; startpt: 1..2n-1;
Step 1: {initialization}
  for all i: 1≦i≦n pardo
                DV[0, i] := D[i] := i; flag[i] := 0
                dopar;
  for all i: 1≦i≦n lg n pardo PTR[i] := 0 dopar;
  for all i: 1≦i≦2n-1 pardo
                FR+[i, 0] := FR+[i, 1] := 0;
                rootv[i] := 0
                dopar;
```

```
    for all i, j; 1 ≦ i, j ≦ n pardo
                    B[1, i, j] := i; B[2, i, j] := j
                    dopar;
    phase := 0; startpt := 0;
repeat
Step 2(a):
    {Pack all defined rows in each segment together}
    S := {i|flag[i] := 0};
        {Set pointers in array PTR. second is a function extracting the second
         portion of a variable formed by the function concatenation in the
         preceding step.}
    temp := second(sort({concat(flag[i], i)|1 ≦ i ≦ n}));
    PTR[phase * n + 1 . . (phase + 1) * n] := second(sort({concat(temp[i],
    startpt + i)|1 ≦ i ≦ |S|} ∪ {concat(temp[i], 0)||S| < i ≦ n}));
    startpt := startpt + n/2 ** phase;
Step 2(b):
    for all i ∈ S pardo
                j0 := min {j|M[i, j] = 1, j ∈ S}
                    if none then j0 := i;
                C[i] := j0;
                FR+[PTR[phase * n + i], 0] := phase * n + i;
                FR+[PTR[phase * n + i], 1] := phase * n + j0
            dopar;
Step 3(a):
    {Check to see if the set S can be reduced any further;
    if not, then terminate execution}
    if (for all i ∈ S, C[i] = i) then exit;
Step 3(b):
    for all i ∈ S pardo if C[i] = i then flag[i] := 1 dopar;
Step 4:
    for all i ∈ S pardo D[i] := C[i] dopar;
Step 5:
    for j := 1 step 1 until lg n do
    for all i ∈ S pardo C[i] := C[C[i]] dopar;
Step 6(a):
    for all i ∈ S pardo D[i] := min {C[i], D[C[i]]} dopar;
Step 6(b):
    for all i: 1 ≦ i ≦ n pardo D[i] := D[D[i]] dopar;
Step 6(c): {Record the array D[i], 1 ≦ i ≦ n}
    for all i: 1 ≦ i ≦ n pardo
                    if i ∈ S
                    then DV[phase + 1, i] := D[i]
                    else DV[phase + 1, i] := D[DV[phase, i]]
                dopar;
Step 6(d): {Convert the edge from the smallest-numbered vertex of each
1-tree-loop to a self-loop}
    for all i: D[i] = i
      pardo
        FR+[PTR[phase * n + i], 1] := FR+[PTR[phase * n + i], 0]
      dopar;
```

```
Step 7(a):
  for all i ∈ S pardo
    for all j ∈ S: j = D[j] pardo
      Choose any j0 ∈ S such that D[j0] = j and M[i, j0] = 1
        if none then j0 := j;
      M[i, j] := M[i, j0];
      B[1, i, j] := B[1, i, j0];
      B[2, i, j] := B[2, i, j0]
    dopar
  dopar;
Step 7(b):
  for all j ∈ S: j = D[j] pardo
    for all i ∈ S: i = D[i] pardo
      Choose any i0 ∈ S such that D[i0] = i and M[i0, j] = 1 if none then i0 := i;
      M[i, j] := M[i0, j];
      B[1, i, j] := B[1, i0, j];
      B[2, i, j] := B[2, i0, j]
    dopar
  dopar;
Step 7(c):
  for all i ∈ S pardo M[i, i] := 0 dopar;
Step 8:
  for all i ∈ S pardo if D[i] ≠ i then flag[i] := 1 dopar;
phase := phase + 1;
until (phase ≧ lg n);
Stage 2
Step 1: {Evaluate the array FR+}
  Compute FR+ and depth [i] for 1 ≦ i ≦ 2n − 1.
Step 2:
  phase := phase − 1;
  {Note that at this point, each vertex k left in S is the root of a in-tree recorded
   in the "last" segment}
  for all k: k ∈ S pardo
    rootv[PTR[phase*n + k]] := k
  dopar;
  repeat
    for all i: (phase * n + 1 ≦ i ≦ (phase + 1) * n
                and PTR[i] ≠ 0
                and FR+[PTR[i], (n − 1) − depth [i]]
                 ≠ FR+[PTR[i], (n − 1) − depth [i] + 1]);
                {not self-loop}
    pardo {Output all the edges except the one emanating
              from the new root first}
      {Denoting FR+[PTR[i], (n − 1) − depth [i]] mod n
       and FR+[PTR[i], (n − 1) − depth [i] + 1] mod n by
       v0[i] and v1[i] respectively}
      if rootv[PTR[i]] = 0 then
        begin
        T[1, B[1, v0[i], v1!i]] := B[1, v0[i], v1[i]];
        T[2, B[1, v0[i], v1[i]]] := B[2, v0[i], v1[i]];
        end;
```

```
{Define the roots for the next segment};
if phase > 0
then rootv[PTR[DV[phase - 1, B[1, v0[i], v1[i]]] +
   (phase - 1) * n]] := B[1, v0[i], v1[i]];
{Reverse the edges if necessary}
if rootv[PTR[i]] ≠ 0
then for all j: ((n - 1) - depth [i] ≦ j < (n - 1))
   pardo{Denoting FR+[PTR[i], j] mod n and FR+[
   PTR[i], j + 1] mod n by v0[j] and v1[j] respectively}
   T[1, B[2, v0[j], v1[j]]] := B[2, v0[j], v1[j]];
   T[2, B[2, v0[j], v1[j]]] := B[1, v0[j], v1[j]];
   {Redefine the roots as well}
   if phase > 0 then
   begin
     rootv[PTR[DV[phase - 1, B[1, v0[j], v1[j]]]
     + (phase - 1) * n]] := 0;
     rootv[PTR[DV[phase - 1, B[2, v0[j], v1[j]]]
     + (phase - 1) * n]] := B[2, v0[j], v1[j]]
   end
   dopar
 dopar;
{Pass the roots defined in the current and previous segments to the next
 segment}
for all i: (phase * n + 1 ≦ i ≦ (phase + 1) * n
       and PTR[i] and rootv[PTR[i]] ≠ 0)
  pardo
    rootv[PTR[DV[phase - 1, rootv[PTR[i]]] + (phase
    -1) * n]] := rootv[PTR[i]]
  dopar;
phase := phase - 1;
until (phase < 0);
```

REFERENCES

[1] A. BORODIN AND J. E. HOPCROPT, *Routing, merging and sorting on parallel models of computation*, Proc. 14th ACM Symposium on Theory of Computing, San Francisco, April 1982, pp. 338–344.

[2] A. K. CHANDRA, *Maximal parallelism in matrix multiplication*, IBM Rept., RC 6193, 1975.

[3] F. Y. CHIN, J. LAM AND I-NGO CHEN, *Efficient parallel algorithms for some graph problems*, Comm. ACM, 25 (1982), pp. 659–665.

[4] ———, *Optimal parallel algorithms for the connected component problem*, IEEE Proc. Intel. Conference on Parallel Processing, 1981, pp. 170–175.

[5] D. M. ECKSTEIN AND D. A. ALTON, *Parallel graph processing using depth-first search*, Conferences on Theoretic Computer Science Univ. Waterloo, 1977, pp. 21–29.

[6] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, Proc. 10th Symposium on Theory of Computing, San Diego, CA, 1978, pp. 114–118.

[7] D. S. HIRSCHBERG, A. K. CHANDRA AND D. V. SARWATE, *Computing connected components on parallel computers*, Comm. ACM, 22 (1979), pp. 461–464.

[8] J. JA' JA' AND J. SIMON, *Parallel algorithms in graph theory: planarity testing*, this Journal, (1982), pp. 314–328.

[9] D. KNUTH, *The Art of Computer Programming, Vol.* 1., 2nd ed., Addison-Wesley, Reading, MA, 1973.

[10] P. KOGGE AND H. STONE, *A parallel algorithm for the efficient solution of a general class of recurrence equations*, IEEE Trans. Comput., C-22 (1973), pp. 786–792.

[11] F. P. PREPARATA AND J. VUILLEMIN, *The cube-connected cycles: A versatile network for parallel computation*, Comm. ACM, 24 (1981), pp. 300–309.

[12] E. REGHBATI AND D. G. CORNEIL, *Parallel computations in graph theory*, this Journal, 7 (1978), pp. 230–237.

[13] J. REIF, *Symmetric complementation*, Proc. 14th ACM Symposium on Theory of Computing, San Francisco, April 1982, pp. 201–214.

[14] E. M. REINGOLD, J. NIEVERGELT AND N. DEO, *Combinatorial Algorithms: Theory and Practice*, Prentice Hall, Englewood Cliffs, NJ, 1977.

[15] C. D. SAVAGE, *Parallel algorithms for graph theoretic problems*, Ph.D. dissertation, R-784, Dept. Mathematics Univ. Illinois, Urbana, 1977.

[16] C. D. SAVAGE AND J. JA' JA', *Fast, efficient parallel algorithms for some graph problems*, this Journal, (1981), pp. 682–691.

[17] Y. SHILOACH AND U. VISHKIN, *An $O(\log n)$ Parallel Connectivity Algorithm*, J. Algorithms, 3 (1982), pp. 57–67.

[18] Y. H. TSIN, *A generalization of Tarjan's depth first search algorithm for the biconnectivity problem*, Tech. Rept. TR82-2, Univ. Alberta, Alberta, April 1982.

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

SIAM J. COMPUT.
Vol. 13, No. 3, August 1984

© 1984 Society for Industrial and Applied Mathematics
009

INSERTION AND COMPACTION ALGORITHMS IN SEQUENTIALLY ALLOCATED STORAGE*

B. S. BAKER† AND E. G. COFFMAN, JR.†

Abstract. Among the more difficult combinatorial problems in computer science are those occurring in dynamic storage allocation. We investigate the handling of memory conflicts occurring when a request for a block of storage is received, but no one region of available space is large enough due to fragmentation of storage. For arbitrary block sizes, we show that it is NP-hard to find a minimal cost reallocation of memory that will allow insertion of a new block. Therefore, we focus on the "bay restaurant" model studied by Robson, in which requests are for only one or two units of memory. We give a polynomial time algorithm for minimal cost insertions of batches of blocks and for minimal cost memory compaction. We show that the cost of inserting p blocks one at a time, i.e. on-line, is no worse than

$$\lfloor 2(\tfrac{3}{2})^{\lceil \log p \rceil} - 1 \rfloor$$

times the minimal cost of inserting the blocks in one batch, i.e. off-line. For p a power of two, this bound is tight.

Key words. dynamic storage allocation, storage compaction, packing algorithms

1. Introduction. Among the more difficult combinatorial problems in computer science are those concerning algorithms used in dynamic storage allocation. The limited literature on the analysis of combinatorial models of sequentially allocated storage has focused primarily on the bay restaurant (or lunch counter) model [4], [5] and the buddy system [2]. (See [3] for a full discussion.) The characteristic question in this research has asked for the minimum size of memory sufficient to guarantee that requests for storage can always be honored without moving blocks already allocated, assuming that the total instantaneous demand never exceeds a certain level.

In this paper, we also examine the bay restaurant model in which requests are for only 1 or 2 units of storage, but we orient our analysis to the familiar problem of insertions of blocks into a fragmented, sequentially allocated memory, and to the related memory compaction problem. Thus, our emphasis will be on the effective handling of memory conflicts occurring when a request for a block of storage is received, the cumulative available space is sufficiently large for the block, but no one region of available space is large enough. In this case, the goal is to minimize the cost of moving blocks to make a contiguous region available for the new block.

In § 2, we describe the model. In § 3, we show that for arbitrary block sizes, the problem of minimizing the cost of moving blocks in order to insert a new block is NP-complete. For the bay restaurant model, however, we give a polynomial time algorithm that inserts blocks at the smallest possible cost. In § 4, we adapt the results for insertions to handle compaction in the bay restaurant model. In § 5, we investigate the relationship between on-line and off-line insertions. In particular, we show that using the algorithm of § 3 for the bay restaurant model, the cost of inserting p blocks one at a time, i.e. on-line, is no worse than

$$\lfloor 2(\tfrac{3}{2})^{\lceil \log p \rceil} - 1 \rfloor$$

times the miniml cost of inserting the blocks in one batch, i.e. off-line. For p a power of two, this bound is tight.

---

* Received by the editors May 28, 1982, and in final revised form May 31, 1983.
† Bell Laboratories, Murray Hill, New Jersey 07974.

2. The model. Let the memory be represented by the set $M=\{1, 2, \cdots, m\}$ of positions or locations. At any given point in time a collection of blocks $S=\{B_1, B_2, \cdots, B_n\}$ with corresponding integer sizes $\{b_1, b_2, \cdots, b_n\}$, $1 \leqq b_i \leqq m$, $1 \leqq i \leqq n$, will be stored or allocated positions in M in the sense that

(i) $\sum_{i=1}^{n} b_i \leqq m$ (memory is sufficiently large),

(ii) for each i, B_i will be assigned consecutive positions represented by some interval $[j, j+b_i-1] \subseteq [1, m]$ (blocks are sequentially allocated), and

(iii) the intervals assigned to different blocks are disjoint.

The set of assigned intervals will determine a particular *allocation* of S. A *hole* in some allocation of S is simply any interval disjoint from those assigned to the B_i's, and not a subset of any larger such interval.

The *insertion problem* for a set of blocks $S'=\{B_{n+1}, \cdots, B_{n+p}\}$ not in S refers to the operation of producing a valid allocation A' of $S \cup S'$ from a given allocation A of S, assuming that

$$\sum_{i=1}^{n+p} b_i \leqq m.$$

The *compaction problem* refers to the production of a new allocation A' of S from a given allocation A of S, such that A' has exactly one hole located in the rightmost position of M.

The basic operation in an insertion or compaction algorithm will be a *move.* A *move* is either

(1) an operation taking an allocation A of S into an allocation of A' of S, where the interval assigned to exactly one block in S differs in A and A', or

(2) the operation of inserting a new block into a hole of an allocation A, without changing the position of other blocks.

In (1), the cost of the move is the size of the block moved; in (2), it is the size of the block inserted. Implicitly, if a move takes a block from $[j, k]$ to $[j', k']$, then those positions in $[j', k',]$ not in $[j, k]$ must be contained in some hole.

We shall be concerned only with algorithms that describe sequences of moves yielding desired allocations; the cost of such a sequence is simply the sum of the costs of its moves. Other models could be conceived, assuming, for example, that there were buffer storage not necessarily disjoint from M. However, our model is consistent with the assumptions that M is the entirety of primary storage, its use is homogeneous over all positions, and no addition temporary storage is used.

3. The insertion problem. The problem of finding a minimum cost sequence of moves for the insertion problem is easily seen to be NP-hard. For example, let $\{a_1, a_2, \cdots, a_{3n}\}$ be an instance of the NP-complete 3-partition problem [3]: Can $3n$ integers $a_1, a_2, \cdots, a_{3n}$ summing to ns, with $s/4 < a_i < s/2$ for $1 \leqq i \leqq 3n$, be partitioned into n sets of three elements, such that the sum of the elements in each set is s? As illustrated in Fig. 1, each such instance can be embedded in an insertion problem. Specifically, the 3-partition question for $\{a_1, \cdots, a_{3n}\}$ can be answered affirmatively if and only if the insertion of a piece of size $w=ns$ into the allocation of Fig. 1 incurs the least possible cost $w+\sum_{i=1}^{3n} a_i = 2w$, or equivalently, if the compaction of this allocation incurs cost w. Thus, we have the following result:

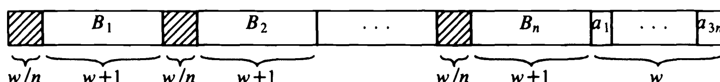


FIG. 1. *Illustrating complexity.*

PROPOSITION 1. *Given an allocation A and integers w and k, it is* NP-*complete to determine whether a block of size w can be inserted into A using cost* $\leqq k$, *or whether compaction of A can be performed using cost* $\leqq k$.

As with similar combinatorial problems, it is important to determine those special cases for which efficient optimization rules can be found for insertion and compaction. This paper concentrates on the fundamental case of the bay restaurant problem, where the block sizes are restricted to two values, s and $2s$. For the analysis of the case of interest, when m is an even multiple of s, there is no loss of generality in the convenient assumption $s=1$. Blocks of size 2 will be called *pairs*, and those of size 1, *unit blocks*, or simply *units*.

The move sequences generated by our insertion and compaction algorithms are based on the notion of *window*. In a given allocation, the interval $[j, k]$, $k=j+2r+1$, $r \geqq 0$, is a window if position j is unoccupied or contains a unit block, position k is unoccupied or contains a unit block, and there are r contiguous pairs in positions $j+1$ to $k-1$. The cost of a window is defined as the sum of the sizes of the blocks in the window. An example is shown in Fig. 2.

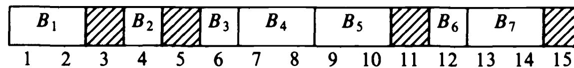


FIG. 2. *Illustration of windows for* $m=15$. *The windows are* $[3,4]$, $[4,5]$, $[5,6]$, $[6,11]$, $[11,12]$, $[12,15]$.

Note that the positions in an initial or final sequence occupied contiguously by pairs are not in any window. The intervals corresponding to maximal such sequences (e.g. positions 1 and 2 in Fig. 2) will be called *boundary intervals.* Also, adjacent windows overlap in exactly one position, but every pair of adjacent positions falls wholly within at most one window (althouth one or both positions may be in each of two windows).

An algorithm for inserting a single pair B_{n+1} will be called a *window algorithm* if it inserts B_{n+1} into some window w, after executing a move sequence satisfying:

(1) Unit blocks in w are moved into holes not in w

(2) In a sequence of $r \geqq 0$ moves the remaining r blocks (all are pairs) are moved one position left or right, creating a hole of size 2.

Note that a sequence satisfying (2) is generally not unique. For example, insertion of a pair in a window such as $[6, 11]$ of Fig. 2 can be made by moving B_3 to position 5 and shifting both B_4 and B_5 one position right or one position left, or by moving B_3 and shifting B_4 left and B_5 right one position. All possibilities lead to the same cost.

The notion of window algorithm can be extended in a natural way to insertions of sets of blocks. An algorithm that inserts a set S' of p blocks into an allocation A_0 is called a window algorithm if it inserts the $p' \leqq p$ pairs of S' into a minimum cost set W of p' disjoint windows in A_0 by move sequences satisfying (1) and (2), and the remaining unit blocks into holes contained in windows not in W.

As we shall see shortly, for $p=1$, a window algorithm that selects a minimum cost window for inserting a single pair implements a minimum cost insertion. Although it may be enticing, it is erroneous to assume that iteratively selecting minimum cost windows is optimal for $p>1$. An example is shown in Fig. 3. This example also shows that such an algorithm does not necessarily satisfy our definition of a window algorithm, which requires that each of the p windows selected be a window in the *original* allocation A_0. The basic result concerning window algorithms is as follows:

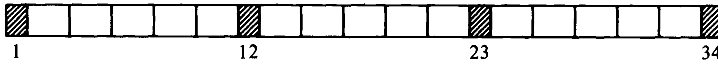


FIG. 3. *Illustrating window insertions. A minimum cost window in* A_0 *is* [12, 23]. *Inserting a pair into* [12, 23] *gives an allocation* A' *with one window* [1, 34] (*which is not a window in* A_0). *Inserting next a pair into* [1, 34] *gives a total cost that exceeds by* 22 *the cost of inserting into window* [1, 12] *followed by inserting into window* [23, 34].

THEOREM 1. *Let block sizes be limited to* 1 *and* 2. *For inserting any set of blocks into a given but arbitrary allocation, there exists a window algorithm achieving minimum cost.*

Proof. Since units can always be inserted at cost 1, there can be no advantage in inserting any unit of a given set before any pairs. Thus, from every optimal algorithm we can construct another that inserts units only after the pairs have been inserted. Consequently, we need only prove the theorem under the assumption that $p \geqq 1$ pairs are to be inserted.

Consider an arbitrary move sequence $\underset{\sim}{x}$, inserting the pairs $S' = \{B_{n+1}, \cdots, B_{n+p}\}$, $p \geqq 1$, into an allocation A_0 of $S = \{B_1, \cdots, B_n\}$ to produce the final allocation A_p. Call $[j, k]$ an α-interval if it is a window or boundary interval of A_0. We say that a block B is moved out of α-interval $[j, k]$ by $\underset{\sim}{x}$ if B is in $[j, k]$ in A_0 but not A_p; conversely, B is moved into $[j, k]$ if it is in $[j, k]$ in A_p but not A_0. We now represent the blocks moved as a result of the insertions of blocks in S' by the following p block sequences.

Each of the sequences begins with a different one of the p pairs in S'. Suppose we have constructed the first $i-1 \geqq 0$ sequences and the first $j \geqq 1$ elements of the ith sequence, $B_1(i), \cdots, B_j(i)$, where $B_1(i)$ is the ith pair selected from S'. If the α-interval in A_0 containing $B_j(i)$ contains at least one pair moved out by $\underset{\sim}{x}$ but not already selected as an element of the first i sequences, one such pair is selected as $B_{j+1}(i)$ and the process is repeated for this new pair. Otherwise, $B_j(i)$ terminates the ith sequence, and if $i < p$, the $(i+1)$st sequence is begun with the next pair from S'.

According to the above process, if a pair in the α-interval $[j, k]$ terminates the ith sequence, it is because all $t \geqq 0$ pairs moved out of $[j, k]$ by $\underset{\sim}{x}$ have already appeared in the first i sequences; and these t pairs were selected because t other pairs in the first i sequences were moved into $[j, k]$ by $\underset{\sim}{x}$. Thus, $[j, k]$ must have at least one more pair in A_p than in A_0, and hence cannot be a boundary interval in A_0. It follows that all α-intervals containing pairs terminating a sequence must be disjoint windows in A_0. If an α-interval $[j, k]$ contained the pairs terminating two different sequences, then $[j, k]$ would contain at least two more pairs in A_p than in A_0. Consequently, there are exactly p disjoint α-intervals containing the pairs that terminate the sequences.

It remains to observe that if $[j, k]$ terminates a sequence, it is in a window in A_0 with $r \geqq 0$ pairs starting in positions $j+1, j+3, \cdots, k-2$, whereas the $r+1$ pairs in $[j, k]$ in A_p start in positions $j, j+2, \cdots, k-1$. Hence, any units in positions j or k, and all pairs in $[j+1, k-1]$ in A_0 must be moved at least one position by $\underset{\sim}{x}$. It follows that an optimal algorithm can do no better than to insert p pairs into a minimum cost set of p disjoint windows of A_0, with each insertion satisfying the properties of a window algorithm. □

Finding a minimum cost set of p windows. Armed with Theorem 1, we now consider briefly the problem of efficiently finding a minimum cost set of p disjoint windows in a given allocation. This problem can be solved in time $O(p^2)$ and space $O(p \log p)$ using the following dynamic programming approach. Let $\underset{\sim}{s} = w_1 w_2 \cdots w_k$ be any

sequence of not necessarily adjacent windows in the same order as in A_0. We want to compute the cost of inserting a set of pairs into $\underset{\sim}{s}$ from the cost of inserting some of the pairs into $w_1 w_2 \cdots w_{\lfloor k/2 \rfloor}$ and the remainder into $w_{\lfloor k/2 \rfloor+1} \cdots w_k$. Taking into account the fact that pairs cannot be inserted into adjacent windows, let $c(\underset{\sim}{s}, r, x_1, x_2)$ denote the minimum cost of inserting r pairs into $\underset{\sim}{s}$, where $0 \leqq r \leqq k$, and x_1 and x_2 in $\{0, 1\}$ are control variables such that x_1, respectively x_2, is equal to one if and only if a pair is inserted into w_1, respectively w_k. These control variables are used to ensure that two adjacent windows are never selected in any of the partial solutions. For $k > 1$, $c(\underset{\sim}{s}, r, x_1, x_2)$ is the minimum of $c((w_1, \cdots, w_{\lfloor k/2 \rfloor}), j, x_1, z_1) + c((w_{\lfloor k/2 \rfloor+1}, \cdots, w_k), r-j, z_2, x_2)$ over $0 \leqq j \leqq \lfloor k/2 \rfloor$ and z_1, z_2 such that at least one of z_1 and z_2 is 0 if $w_{\lfloor k/2 \rfloor}$ and $w_{\lfloor k/2 \rfloor+1}$ are adjacent. A table containing $c(\underset{\sim}{s}, r, x_1, x_2)$ and the values of j, z_1, and z_2 producing this value, for $0 \leqq k$ and x_1, x_2 in $\{0, 1\}$, can be computed recursively from the tables for $w_1, w_2, \cdots, w_{\lfloor k/2 \rfloor}$ and $w_{\lfloor k/2 \rfloor+1}, \cdots, w_k$ in time $O(k^2)$ and space $O(k \log k)$.

Now, the p disjoint windows of minimal cost must be selected from the $3p-2$ windows of minimal cost, since each of the first $p-1$ windows selected eliminates at most two adjacent windows from consideration. Therefore, we may apply the above algorithm to the sequence of $3p-2$ windows of minimal cost, and the entire computation will require $O(p^2)$ time and $O(p \log p)$ space. The set of p disjoint windows of minimal cost can be recovered in time $O(p)$ from the values of j, z_1, and z_2 in the table.

4. The compaction problem. Insertion algorithms can be applied directly to the problem of memory compaction for an arbitrary allocation A, i.e., a minimum cost placement of blocks into the interval $[1, b]$, where

$$b = \sum_{i=1}^{n} b_i$$

and $\{b_i\}_{i=1}^n$ is the collection of block sizes in A. If there is no block occupying an interval containing $[b, b+1]$, a minimum cost compaction of A is simply a minimum cost insertion of the blocks in $[b+1, m]$ into the suballocation of A restricted to $[1, b]$. In our bay restaurant model, the case of a pair in $[b, b+1]$ is easy to handle. If there is at least one window contained in $[1, b-1]$, an optimum insertion sequence can be started with the pair in $[b, b+1]$. Otherwise, we must have exactly one unoccupied position in $[1, b]$, and all blocks to the right of this hole can be moved left one position for an optimum compaction.

5. Off-line vs. on-line insertions. Compaction is a natural application of off-line insertions, where batches of insertions are known in advance. However, a typical assumption in dynamic storage allocation is that insertions are made on-line and one at a time, with no knowledge of future insertions. Under these circumstances, one may consider adopting the obvious least-cost-window (LCW) algorithm: simply insert the next block into a window of the current allocation having least cost. By Theorem 1, this is obviously locally optimal. However, as the next result shows and as illustrated in Fig. 3, this algorithm can perform quite badly in the worst case, as compared to an algorithm that is able to batch insertions.

Concentrating as before only on the insertion of pairs, let OPT (A, p) denote the cost of inserting p pairs into a minimum cost collection of p disjoint windows in A, and let LCW (A, p) denote the cost of iteratively inserting p pairs into A by the LCW algorithm.

THEOREM 2. *For all A and* $p \geqq 1$, *we have*[1]

$$\frac{\mathrm{LCW}(A,p)}{\mathrm{OPT}(A,p)} \leqq \left[2\left(\frac{3}{2}\right)^{\lceil \log p \rceil} - 1 \right]. \tag{1}$$

Moreover, for p a power of 2 *no smaller bound is possible.*

Proof. The result is trivially true for $p = 1$, so let us assume $p > 1$. We proceed by contradiction: Let (A, p) be a counterexample to (1) and suppose that it is smallest in the sense that there exists no other (A', p'), $p' < p$, violating (1).

Let $\underset{\sim}{w} = w_1 w_2 \cdots w_p$ denote the sequence of disjoint windows used by OPT, and let $\underset{\sim}{v} = v_1 v_2 \cdots v_p$ denote the sequence of windows used by LCW, in the order that the insertions are made. We have

$$\mathrm{LCW}\,(A,p) = \sum_{i=1}^{p} c(v_i) + 2p, \qquad \mathrm{OPT}\,(A,p) = \sum_{i=1}^{p} c(w_i) + 2p.$$

We say that a window v *touches* window w if $v = w$ or if v and w are adjacent. Let $\underset{\sim}{x} = x_1 x_2 \cdots x_t$ be that subsequence of $\underset{\sim}{v}$ such that for each i, $1 \leqq i \leqq t$, x_i touches one or two windows in $\underset{\sim}{w}$ not already touched by $x_1 x_2 \cdots x_{i-1}$. Observe that every window in $\underset{\sim}{w}$ must be touched by at least one window in $\underset{\sim}{x}$; otherwise there would be a window in $\underset{\sim}{w}$ disjoint from and having a cost at least that of any window in $\underset{\sim}{v}$. It is easily seen that in this case $(A', p-1)$ would be a smaller counterexample to the theorem, where A' is obtained from A by inserting a pair in w.

CLAIM 1. *There is a subsequence* $\underset{\sim}{y}$ *of* $\underset{\sim}{v}$ *inserting* $\lfloor p/2 \rfloor$ *pairs such that*

(a) *the total cost of* $\underset{\sim}{y}$ *is at most* $\mathrm{OPT}(A,p)/2$, *and*

(b) *for any* y_j *in* $\underset{\sim}{y}$ *and* v_i *in* $\underset{\sim}{v}$, *if* v_i *is a subinterval of* y_j *then* v_i *is also in* $\underset{\sim}{y}$.

Proof. First, we construct a subsequence $\underset{\sim}{z}$ of $\underset{\sim}{v}$ that satisfies (a) but not necessarily (b). Let $x_{i_1}, \cdots, x_{i_r}$ be those x_i's touching two windows in $\underset{\sim}{w}$, each for their first time, and let $x_{i_{r+1}}, \cdots x_{i_t}$ be those touching just one for the first time. Without loss of generality, we assume for convenience that $c(x_{i_{r+1}}) \leqq c(x_{i_{r+2}}) \leqq \cdots \leqq c(x_{i_t})$, where $c(x_{i_j})$ denotes the cost of x_{i_j}. For $\underset{\sim}{z}$ we select $z_j = x_{i_j}$, $1 \leqq j \leqq r + \lfloor (t-r)/2 \rfloor = \lfloor (r+t)/2 \rfloor$. Each of the z_j, $j \leqq r$, has at most the cost of each of the two windows in $\underset{\sim}{w}$ that it touches. For $j > r$, z_j has at most the cost of the window in $\underset{\sim}{w}$ that x_{i_j} touches for the first time, and, because of the assumed ordering, it has at most the cost of the window touched by $x_{i_{j'}}$, $j' = j + \lfloor (t-r)/2 \rfloor$, for the first time. It follows routinely that

$$\sum_{j=1}^{\lfloor (r+t)/2 \rfloor} z_j \leqq \frac{\mathrm{OPT}\,(A,p)}{2}.$$

But, since $2r + (t-r) = r + t = p$, we have $\lfloor (r+t)/2 \rfloor = \lfloor p/2 \rfloor$.

Next, we obtain $\underset{\sim}{y}$ satisfying both (a) and (b) by replacing some of the z_j as follows. Examine the windows $v_1, v_2, \cdots, v_p$ in that order. If v_j is not in the sequence, say $\underset{\sim}{z}'$, produced after processing $v_1, \cdots, v_{j-1}$, but it is a subinterval of some z in $\underset{\sim}{z}'$, then replace one such z in $\underset{\sim}{z}'$ by v_j. Note that the total cost of the new sequence has at most the cost of $\underset{\sim}{z}'$.

Let y be the final sequence obtained after examining v_p. Clearly, (a) remains satisfied by $\underset{\sim}{y}$. To verify that (b) is also satisfied, let v_i be a subinterval of v_j and suppose v_j is in $\underset{\sim}{y}$. Clearly, $j > i$ and v_j is examined after v_i. If v_j replaces a window when it is examined, there must have been a window containing v_j in the sequences existing at the times the windows v_k, $k \leqq j$, were examined. Thus, whether or not v_j is in the current sequence when v_i is examined, if v_i is not already in the sequence, it will

[1] All logarithms are base 2.

replace a window that is. Since v_i can contain no interval v_k, $k > i$, v_i cannot be replaced subsequently. Thus, v_i must be in $\underline{y}$. Therefore (b) is satisfied by $\underline{y}$. □

Now, order $\underline{y}$ so that the intervals in $\underline{y}$ are in the same order as in $\underline{v}$. Reorder $\underline{v}$ so that $\underline{y}$ is the initial sequence into which insertions are made, and the remaining intervals are in the same order as before. By property (b) of Claim 1, each interval is a window at the time an insertion is to be made into it in the new sequence $\underline{v}'$. Let A' be the allocation after the insertions of $\underline{y}$ are made according to $\underline{v}'$. It is easily seen that the sequence remaining in $\underline{v}'$ after $\underline{y}$ is an LCW sequence inserting $\lceil p/2 \rceil = p - \lfloor p/2 \rfloor$ pairs into A'. Thus, we can write

$$\mathrm{LCW}(A, p) = \sum_{i=1}^{\lfloor p/2 \rfloor} c(y_i) + 2\lfloor p/2 \rfloor + \mathrm{LCW}(A', \lceil p/2 \rceil),$$

and, since (A, p) is a smallest counterexample,

$$\mathrm{LCW}(A, p) \leqq \sum_{i=1}^{\lfloor p/2 \rfloor} c(y_i) + 2\lfloor p/2 \rfloor + \left[2\left(\frac{3}{2}\right)^{\lceil \log \lceil p/2 \rceil \rceil} - 1 \right] \mathrm{OPT}(A', \lfloor p/2 \rfloor).$$

Using Claim 1 and the construction of $\underline{y}$, we have

$$\mathrm{LCW}(A, p) \leqq \tfrac{1}{2}\,\mathrm{OPT}(A, p) + \left[2(\tfrac{3}{2})^{\lceil \log p \rceil} - 1\right] \mathrm{OPT}(A', \lceil p/2 \rceil), \tag{2}$$

where we have also used $\lceil \log \lceil p/2 \rceil \rceil = \lceil \log p \rceil - 1$ for $p > 1$.

The next major step is given by:

CLAIM 2. *For each* $p \geqq 2$

$$\mathrm{OPT}(A', \lceil p/2 \rceil) \leqq (\tfrac{3}{2})\,\mathrm{OPT}(A, p).$$

Proof. Consider those intervals in A' of maximum length such that each position in such an interval is an interval of $\underline{y}$ or $\underline{w}$. In characterizing the structure of these disjoint intervals, which we shall call maximal $\underline{y}$, $\underline{w}$-intervals, we recall the following facts: The intervals of $\underline{w}$ are disjoint and two intervals of $\underline{y}$ are either disjoint or one is a subinterval of the other. Moreover, if an interval w in $\underline{w}$ and an interval y in $\underline{y}$ overlap in more than one position, then w must be a subinterval of y. Thus, each maximal $\underline{y}$, $\underline{w}$-interval corresponds to a unique, minimum-length sequence of intervals of one of the following three types:

1. $y_{i_1} w_{i_1} y_{i_2} \cdots y_{i_{k-1}} w_{i_{k-1}} y_{i_k}, \quad k \geqq 1,$
2. $y_{i_1} w_{i_1} y_{i_2} \cdots y_{i_k} w_{i_k}$ or $w_{i_1} y_{i_1} w_{i_2} \cdots w_{i_k} y_{i_k}, \quad k \geqq 1,$
3. $w_{i_1} y_{i_1} w_{i_2} \cdots w_{i_{k-1}} y_{i_{k-1}} w_{i_k}, \quad k \geqq 1,$

where y_{i_j} is in $\underline{y}$, and w_{i_j} is in $\underline{w}$ but not $\underline{y}$, $1 \leqq j \leqq k$. Let S denote the set of these sequences describing all of the maximal $\underline{y}$, $\underline{w}$-intervals in A'. Since each such sequence has minimum length, no y_{i_j} or w_{i_j} in such a sequence can be a proper subinterval of any interval in $\underline{y}$. Note that every interval in $\underline{y}$ and $\underline{w}$ is in exactly one maximal $\underline{y}$, $\underline{w}$-interval, and an interval in $\underline{y}$ or $\underline{w}$ is in a sequence of S if and only if it is not properly contained in any interval of $\underline{y}$.

Now suppose there are at least $\lceil p/2 \rceil$ sequences of type 3 in S. Since such sequences begin and end with intervals in $\underline{w}$, they must correspond to maximal $\underline{y}$, $\underline{w}$-intervals in A' that are in fact windows in A'. Clearly, they are available for the insertion of the $\lceil p/2 \rceil$ pairs remaining after the insertions of $\underline{y}$ have been made. Bounding the cumulative cost of $\lceil p/2 \rceil$ insertions into these windows we have

$$\mathrm{OPT}(A', \lceil p/2 \rceil) \leqq \sum_{i=1}^{\lfloor p/2 \rfloor} c(y_i) + \sum_{i=1}^{p} c(w_i) + 2\lceil p/2 \rceil.$$

Use of Claim 1 and the construction of $\underset{\sim}{y}$ then establishes Claim 2.

It remains to prove that in fact there must be at least $\lceil p/2 \rceil$ sequences in S of type 3. Our approach will be to show that the maximal $\underset{\sim}{y}$, $\underset{\sim}{w}$-intervals described by sequences in S of types 1 and 2 all contain at least as many intervals of $\underset{\sim}{y}$ as intervals of $\underset{\sim}{w}$, and those described by sequences of type 3 contain at most one more interval in $\underset{\sim}{w}$ than $\underset{\sim}{y}$. The result then follows directly from the fact that there are $\lceil p/2 \rceil$ more intervals in $\underset{\sim}{w}$ than in $\underset{\sim}{y}$,

First, consider any interval y in $\underset{\sim}{y}$ containing $r \geqq 1$ intervals in $\underset{\sim}{w}$. Before LCW can insert a pair into y it must have inserted enough pairs into subintervals of y to turn y into a window. Consequently, either

(a) y has initial and final subintervals coinciding with windows in $\underset{\sim}{w}$, and at least $r-1$ proper subintervals of y are in $\underset{\sim}{y}$, or

(b) there is no initial interval of y coinciding with a window in $\underset{\sim}{w}$, or there is no final interval of y coinciding with a window in $\underset{\sim}{w}$, and at least r proper subintervals of y are in $\underset{\sim}{y}$.

Now let $\underset{\sim}{s}$ be a type 1 sequence in S. By definition there are more intervals of $\underset{\sim}{y}$ in $\underset{\sim}{s}$ than intervals of $\underset{\sim}{w}$. Thus, if no y in $\underset{\sim}{y}$ appearing in $\underset{\sim}{s}$ contains an interval of $\underset{\sim}{w}$ then the interval corresponding to $\underset{\sim}{s}$ contains at least as many intervals of $\underset{\sim}{y}$ as of $\underset{\sim}{w}$.

So suppose there is a y in $\underset{\sim}{y}$ and $\underset{\sim}{s}$ that contains at least one interval in $\underset{\sim}{w}$. If there is at least one interval of $\underset{\sim}{w}$ in $\underset{\sim}{s}$, y must be adjacent to at least one such interval. Since no two intervals of $\underset{\sim}{w}$ can be adjacent, y must satisfy (b) and hence it *properly* contains at least as many intervals in $\underset{\sim}{y}$ as in $\underset{\sim}{w}$. Once again, it follows that the number of intervals in $\underset{\sim}{y}$ contained in the interval represented by $\underset{\sim}{s}$ must be at least the number of intervals in $\underset{\sim}{w}$ that are contained in the interval. If there is no interval of $\underset{\sim}{w}$ in $\underset{\sim}{s}$, then by definition $\underset{\sim}{s}$ must consist solely of the interval y in $\underset{\sim}{y}$. Since y is an LCW window, it cannot begin or end with a subinterval that is an interval in $\underset{\sim}{y}$. Hence, since we are assuming that y has at least one interval in $\underset{\sim}{w}$, y must satisfy (a), and the number of intervals of $\underset{\sim}{y}$ properly contained in y cannot be more than one less than the number of intervals in $\underset{\sim}{w}$ properly contained in y. Thus, counting y itself along with the intervals of $\underset{\sim}{y}$ that it contains, we have once again that the interval represented by $\underset{\sim}{s} = y$ contains at least as many intervals of $\underset{\sim}{y}$ as of $\underset{\sim}{w}$.

Next, suppose $\underset{\sim}{s}$ is a type 2 sequence in S. By definition it has as many intervals in $\underset{\sim}{y}$ as in $\underset{\sim}{w}$. Since each such type 2 sequence must have at least one interval in $\underset{\sim}{w}$, each y in $\underset{\sim}{y}$ and $\underset{\sim}{s}$ must satisfy (b), if it contains at least one interval of $\underset{\sim}{w}$. Thus, using exactly the same arguments as before, we see that the interval represented by $\underset{\sim}{s}$ contains at least as many intervals of $\underset{\sim}{y}$ as of $\underset{\sim}{w}$.

Our final observation is that if $\underset{\sim}{s}$ is a type-3 sequence, it must have at least one interval in $\underset{\sim}{w}$. Therefore, any interval in $\underset{\sim}{y}$ and $\underset{\sim}{s}$ containing at least one interval of $\underset{\sim}{w}$ must satisfy (b). Since a type-3 sequence has one more interval in $\underset{\sim}{w}$ than $\underset{\sim}{y}$, it follows that type-3 sequences correspond to intervals containing at most one more interval of $\underset{\sim}{w}$ than $\underset{\sim}{y}$. This completes the proof of Claim 2. □

Using Claim 2 in (2), we get

$$\mathrm{LCW}(A, p) \leqq \tfrac{1}{2}\,\mathrm{OPT}(A, p) + \tfrac{3}{2}\,\mathrm{OPT}(A, p) \lfloor 2(\tfrac{3}{2})^{\lceil \log p \rceil - 1} - 1 \rfloor$$

and, hence, the result we have been seeking:

$$\mathrm{LCW}(A, p) \leqq \lfloor 2(\tfrac{3}{2})^{\lceil \log p \rceil} - 1 \rfloor\, \mathrm{OPT}(A, p).$$

Our last step shows that (1) is asymptotically achievable for any p that is a positive power of 2. Let $V(A, p)$ be the cost incurred by LCW in moving just those blocks originally in A, disregarding the cost of moving blocks inserted subsequently. Define

the allocation $A_1(n)$ having no units and exactly 3 consecutive windows, each with n pairs. ($A_1(5)$ is shown in Fig. 3.) By having LCW choose the middle window first, we see that $V(A_1(n), 2) = 8n$, whereas $\mathrm{OPT}(A_1(n), 2) = 4n+4$ by choosing the outer windows. Inductively, as illustrated in Fig. 4, define the allocation $A_{k+1}(n) = A_k(n)z_kA_k(n)$, using the obvious notation, where z_k is a sequence of 3^kn pairs. If LCW always selects the middle of three adjacent windows of equal cost, it inserts 2^{k+1} pairs into $A_{k+1}(n)$ by first filling up both copies of $A_k(n)$ (except for the extreme positions) by inserting the first $2^{k+1}-2$ pairs, then filling up the window containing z_k by inserting another pair, and finally inserting the last pair into the sole remaining window, which is forced by the above insertions to include all of memory. If we define

$$V_k = \frac{V(A_k(n), 2^k)}{2n}, \qquad k \geqq 1,$$

it is routine to verify that

$$V_{k+1} = 2V_k + 2 \cdot 3^k, \qquad k = 1, 2, \cdots.$$

Solving this recurrence, we get

$$V_k = 2 \cdot 3^k - 2^k, \qquad k \geqq 1.$$

An optimization rule can insert 2^k pairs into 2^k disjoint windows, each containing n pairs. Hence,

$$\frac{\mathrm{LCW}(A_k(n), 2^k)}{\mathrm{OPT}(A_k(n), 2^k)} \geqq \frac{V(A_k(n), 2^k)}{\mathrm{OPT}(A_k(n), 2^k)} = \frac{2n[2(3^k) - 2^k]}{2(n+1)2^k}.$$

By choosing n sufficiently large, this ratio can be made as close as desired to $2(\frac{3}{2})^k - 1$. This completes the proof of Theorem 2. □

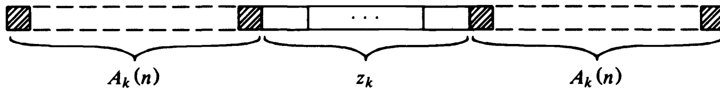


FIG. 4. *Illustrating $A_{k+1}(n)$ in the worst case example of Theorem 2.*

6. Discussion. In this paper we have addressed the complexity problem of insertions and compactions within linear, arbitrarily fragmented storage. From an engineering point of view of the entire question of dynamic storage allocation, one must also consider schemes such as the buddy system for governing the evolution of allocations that facilitate insertions and/or compactions. For example, in a pragmatic approach to the bay restaurant model of two block sizes, it is natural to consider an allocation policy that assigns pairs so that they begin in only the odd locations; insertions may now require the moving of one or two units, even when two consecutive unoccupied locations exist, but pairs will never have to be moved to make room for a new block. Such a scheme extends in the obvious way to the assumption of block sizes in the set $\{1, 2, 4, 8, 16, \cdots, m\}$, assuming m is a power of 2. It can be shown that inserting a block of size n in such a system can be done in an amount of time proportional to $n \log n$. The authors intend to present this along with other characterizations of the model in a subsequent paper.

There are many other interesting open problems connected with the model of this paper. There are the obvious ones of generalizing possible block sizes to $\{1, k\}$, $\{1, 2, 3\}$, or $\{1, 2, 4\}$, etc., which apparently entail a significantly greater difficulty. Possibly there might be polynomial time approximation algorithms which would handle

arbitrary block sizes. It would be interesting to know how bad on-line insertions can be compared to batched insertions for arbitrary block sizes. The definition of compaction might be changed to require only that a single hole be created somewhere in memory, rather than at the right end; it appears that finding a minimum cost compaction with this revised definition becomes significantly harder. One might also consider modifications to the model. For example, the cost function could be changed to reflect sequentially accessed storage (e.g. disks) rather than randomly accessed storage. Another possibility would be to allow additional temporary storage outside the region of memory being allocated. In our model, we assumed that there was no temporary storage available except to store the batch of blocks to be inserted. However, the proof of Theorem 1 analyzed only the positions of blocks in the initial and final allocations. Therefore, even if algorithms were allowed to store blocks temporarily in external storage at no cost, there would still be an optimal window algorithm which made no use of external storage. We conclude that external storage is not helpful for blocks of size 1 and 2. It might of course be helpful for handling blocks of arbitrary size.

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] K. C. KNOWLTON, *A fast storage allocator*, Comm. ACM, 8 (1965), pp. 623–625.

[3] D. E. KNUTH, *Fundamental Algorithms*, Vol. I, 2nd ed., Addison-Wesley, Reading, MA, 1973.

[4] J. M. ROBSON, *An estimate of the store size necessary for dynamic storage allocation*, J. Assoc. Comput. Mach., 18 (1971), 416–423.

[5] ———, *Bounds for some functions concerning dynamic storage allocation*, J. Assoc. Comput. Mach., 21 (1974), 491–499.

SIAM J. COMPUT.
Vol. 13, No. 3, August 1984

© 1984 Society for Industrial and Applied Mathematics
010

MOVEMENT PROBLEMS FOR 2-DIMENSIONAL LINKAGES*

JOHN HOPCROFT†, DEBORAH JOSEPH‡ AND SUE WHITESIDES§

Abstract. This paper is motivated by questions concerning the planning of motion in robotics. In particular, it is concerned with the motion of planar linkages from the complexity point of view. There are two main results. First, a planar linkage can be constrained to stay inside a bounded region whose boundary consists of straight lines by the addition of a polynomial number of new links. Second, the question of whether a planar linkage in some initial configuration can be moved so that a designated joint reaches a given point in the plane is PSPACE-hard.

Key words. robotics, manipulators, mechanical arms, algorithms, polynomial time, PSPACE-hard

1. Introduction. This paper is concerned with the motion of linkages from the computational complexity point of view. The research was motivated by earlier work in robotics, particularly that of Lozano-Perez and Wesley [LW-79], Lozano-Perez [L-80], Reif [R-79] and Schwartz and Sharir [S-81], [S-82]. There are two natural ways in which linkage movement problems arise in robotics. First, a linkage can model a robot arm. A frequently encountered model consists of a sequence of links connected together consecutively at movable joints. Second, linkages can also model hinged objects being moved by an arm or other type of manipulator. In both cases, it is essential to plan collision-avoiding paths of motion, as the manipulator and the object it is moving are generally required to lie within regions whose boundaries are determined by walls and the presence of other objects in the work space.

A *linkage* is a collection of rigid rods called *links* (see Fig. 1.1). The endpoints of various links are connected by joints, each joint connecting two or more links. The

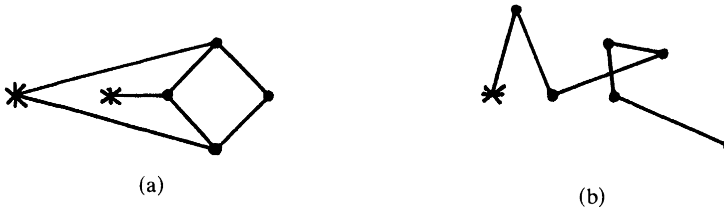


FIG. 1.1. (a) *A planar linkage.* (b) *An arm in the plane.*

links are free to rotate about the joints. In a *planar* linkage, links are allowed to cross over one another, and the linkage may be fastened to the plane so that the locations of certain joints are fixed (the fixed joints are indicated by $*$ in the figures).

In a physical realization of a planar linkage, each link could move in a separate plane parallel to the ground. If links were joined together or to the ground by pins, then a link in one plane might collide with a pin joining links in two other planes. However, it is not difficult to design simple devices that function like pins but that do not interfere with the motions of the linkage. Thus the mathematical model in which

* Received by the editors November 19, 1982, and in revised form June 17, 1983. This research was supported in part by Office of Naval Research under contract N00014-76-C-0018, in part by the National Science Foundation under grant MCS81-01220, and the National Science Foundation under a Postdoctoral Fellowship and in part by a Dartmouth College Junior Faculty Fellowship.

† Department of Computer Science, Cornell University, Ithaca, New York 14853.
‡ Department of Computer Science, University of Wisconsin, Madison, Wisconsin 53706.
§ School of Computer Science, McGill University, Montreal, Quebec, Canada H3A 2K6.

links cross over one another and in which the locations of some joints are fixed can be physically realized.

An *arm* is a simple type of linkage consisting of a sequence of links joined together consecutively with the location of one end fixed.

Suppose that an arm is required to stay inside some given region R of the plane. It is a natural question to ask whether new links can be adjoined to the arm in such a way that the original links in the arm automatically stay inside R. The new links may move outside R, and some of the links may have an endpoint fixed in the plane. The key requirement is that no motion of the arm inside R be prevented by the addition of the new links. Figure 1.2 shows how this can be done if R is a circular region.

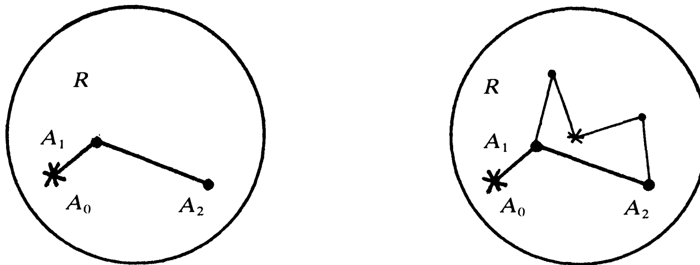


FIG. 1.2. *An arm confined to a circular region. Connecting joints A_1 and A_2 to the center of the circle by two-link "elbows" keeps the arm in R.*

The reason that we are interested in reductions of this sort is that the motions of the new linkage can be studied without reference to the region R. The first main result of our paper is that for any compact connected, but not necessarily simply connected, region R whose boundary consists of a finite set of straight line segments and any linkage L positioned within R, there is a reduction of the type we have just described. What is more the number of new links that must be added is bounded by a polynomial in the number of original links and the number of sides of R. Also the lengths and placements of the new links are easy to compute.

Our second result is that the reachability question for planar linkages is PSPACE-hard. In other words, given an initial configuration of an arbitrary planar linkage L, a joint J in that linkage and a point p in the plane, the question of whether L can be moved so that J reaches p is PSPACE-hard.

The main technique used throughout the paper is to build complex linkages by connecting together simpler special purpose linkages. Some of these simpler linkages date from the 19th century and are described in § 2. These include Peaucellier's straight line motion device, which is a linkage containing a joint whose locus is *exactly* a straight line segment, and linkages that translate and rotate vectors and multiply distances.

Section 3 contains an easy demonstration that a linkage required to move inside a closed bounded *convex* polygonal region R can be embedded in a more complex linkage that enforces the boundary constraint for the original linkage. The extension of this result to a linkage L constrained to move inside a nonconvex bounded region R with straight line boundaries appears in § 4. We obtain this result by triangulating the complement of R in its convex hull H, designing a linkage that contains a joint whose locus is a triangle, and then using this device to build a linkage that can keep a link entirely outside a triangle. By keeping each link of L outside each triangle in $H-R$ while requiring each joint of L to remain inside H, we keep L inside R. We do this in such a way that the motion of L is not restricted in any other way.

Section 5 contains our other main result, that the reachability question for planar linkages is PSPACE-hard. We obtain this result by designing a linkage that can simulate a linear bounded automaton (LBA). The result should be compared to Reif's result [R-79] that in 3-dimensional space, the reachability problem is PSPACE-hard even for a simple, hinged, tree-like linkage required to move in a nonconvex region.

2. Simple linkages.

2.1. Overview. This section describes planar linkages that perform certain tasks. After a discussion in § 2.2 of Peaucellier's straight line motion linkage, we show in § 2.3 how to use this device to build linkages that can translate and rotate vectors. Then in § 2.4 we use these devices to give a modified version of Kempe's construction of a linkage that "solves" a multivariable polynomial equation [K-1876]. This linkage has certain joints whose positions represent values of variables $x_1, \cdots, x_n$, and the only constraint that the linkage puts on the motion of these joints is that the implied values of the x_i stay within given bounded domains and satisfy a given polynomial equation.

The linkage for solving a polynomial equation plays an important role in both the main results. We use it to keep links outside of triangular regions when we show how to build boundary constraints into a linkage in § 4. We also use it to synchronize the motions of the LBA simulator given in § 5.

Two important subtleties arise in designing special purpose linkages. First, we often want to construct a linkage L having a joint J whose locus is some specified set of points. It is important to understand that, in such a case, L must be able to move to all points in the set but to *no* other points. Historically, some linkages that have been proposed for performing certain tasks have been faulty because, while they are able to move in some desirable way, they can also move to "configurational singularities" at which they can begin undesired motions. Hence for the sake of completeness we include redesigned versions of these linkages that avoid this problem.

The second important subtlety is this: Suppose that the locus of some joint J in linkage L is a set of points S and that the locus of some joint J' in linkage L' is a set S'. Now suppose that J and J' are identified. It is not necessarily true that the joint $J=J'$ can then reach all points in $S \cap S'$. Indeed $S \cap S'$ need not be connected! We have been careful to avoid this pathology in designing our linkages. The crucial observation is the following. Suppose that $x(t)$ and $y(t)$ are given continuous functions of time. When a new linkage is formed from L and L' by identifying joints J and J', this new linkage can move so that the position of $(J=J')$ is given by $(x(t), y(t))$ if and only if L can move so that the position of J is given by $(x(t), y(t))$ and L' can move so that the position of J' is also given by the same $(x(t), y(t))$. This observation should be kept in mind when checking that the linkages we build up from smaller pieces function as claimed.

2.2. Peaucellier's straight line motion linkage. In 1864 Peaucellier [P-1864] designed a linkage, shown in Fig. 2.2.1, that converts circular motion to linear motion. Links AD, AB, DC and BC have equal length, as do links EA and EC. The length of FD equals the distance from E to F. The locations of joints E and F are fixed points in the plane, but the linkage is allowed to rotate about these points. As it does, the joint B traces out the line segment XY. This can be seen by observing two facts. First, joints D and B always lie on a ray through E. Second, the distances h, r and t shown in Fig. 2.2.2 satisfy $h^2=a^2-t^2=b^2-(r+t)^2$, where r is the distance between D and E. Consequently the distance s between E and B is such that rs is equal to the constant b^2-a^2. (Here, h, r, s and t are functions of the position of D.) Hence,

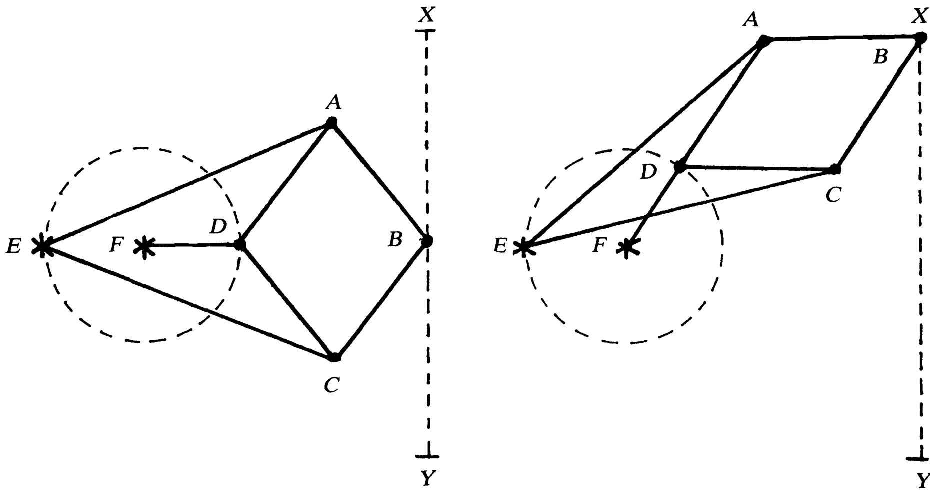


FIG. 2.2.1. *The Peaucellier straight line motion linkage.*

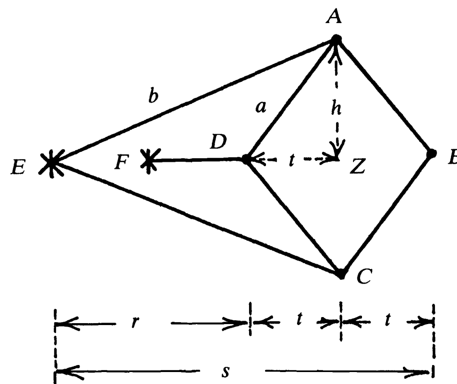


FIG. 2.2.2. *Consideration of triangles EAZ and DAZ shows that $h^2 = a^2 - t^2 = b^2 - (r+t)^2$, where h is half the distance between A and C, t is half the distance between D and B and r is the distance between E and D.*

this device can be thought of as performing the well-known mapping called "inversion with respect to a circle" [E-63]. In this mapping, the image of a point $p = (r, \theta)$ is the point $p' = (r', \theta)$ where rr' is some given constant. It is known that this mapping takes circles to circles, where a straight line is regarded as a circle of infinite radius. Suppose that the joint E of the Peaucellier device is at the origin of the polar coordinate system and that the given constant is $b^2 - a^2$. Then the device computes the images of the points that D can reach. Since the circle of radius $|FD|$ about F goes through the origin, this circle is mapped to a straight line, in particular the line through X and Y. The points X and Y represent the extremes that B can reach.

The relative lengths of the links are not important provided that the linkage can be assembled as shown in Fig. 2.2.1 with E, F, D and B on a straight line. In order to argue that the Peaucellier device works correctly we must demonstrate that joint B cannot reach joint D, for if this could occur, the joint B could leave the line segment XY and trace out part of the circle that D traces. Similarly we must demonstrate that joint A cannot reach joint C. Joint B cannot reach joint D since the line XY does

not intersect the circle of radius $|FD|$ centered at F. To see that joint A cannot reach joint C, suppose that B moves along XY toward X, say. The parallelogram $ABCD$ begins to collapse; diagonal DB lengthens, and diagonal AC shortens. Also, link EA moves counterclockwise about E, thereby increasing the distance from A to F. Consequently, the collapsing of the parallelogram is stopped when angle FDA straightens, preventing further counterclockwise rotation of EA. This occurs when B reaches X, as shown in Fig. 2.2.1. The reader is referred to [E-63] for a more detailed discussion.

Now we point out some consequences of a simple modification of the Peaucellier device that we will need in § 3, where we will describe how to confine a linkage to the inside of a convex polygon.

Suppose that the Peaucellier linkage is modified by adding a new link BG. As joint B travels up and down line segment XY, link BG can rotate freely about B. Clearly the set S of points that joint G can reach is the union of a rectangle and two discs (see Fig. 2.2.3). Note that G can follow any curve that stays inside S but avoids

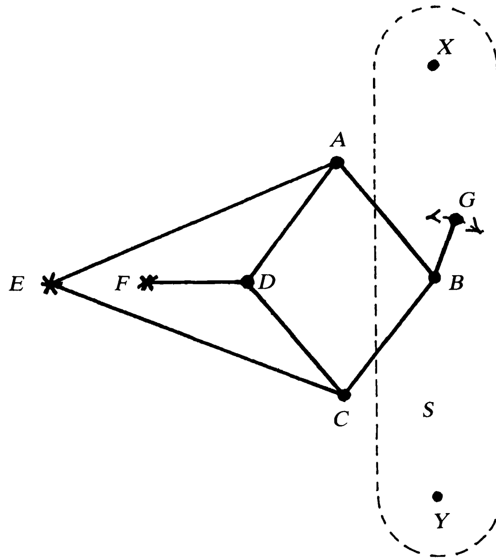


FIG. 2.2.3. *The region of points reachable by the modified Peaucellier linkage.*

the discs centered at X and Y. (G cannot move inside the discs when B is at X or Y.) Consequently, when we are faced in § 3 with the problem of designing a modified Peaucellier linkage whose joint G must be able to move freely inside some given polygonal region R, we simply scale the linkage shown in Fig. 2.2.3 by an approximate constant so that region R fits inside set S with the discs removed.

We will take advantage of the fact that S has a straight line segment in its boundary by placing the modified Peaucellier linkage so an edge of the polygon R lies along a boundary line of S. This will keep joint G from crossing that edge of R.

We will frequently use the Peaucellier linkage to constrain a joint J of some other linkage to a line. This can be done by identifying the joint J with the joint of the Peaucellier linkage that moves on a line. When we do this identification, we say that J is moving in a *slot*. The geometry and positioning of the Peaucellier linkage determine the length and position of the slot.

Also observe that the two points of the Peaucellier device that are normally attached to the plane could instead be attached to a rigid structure made up of links

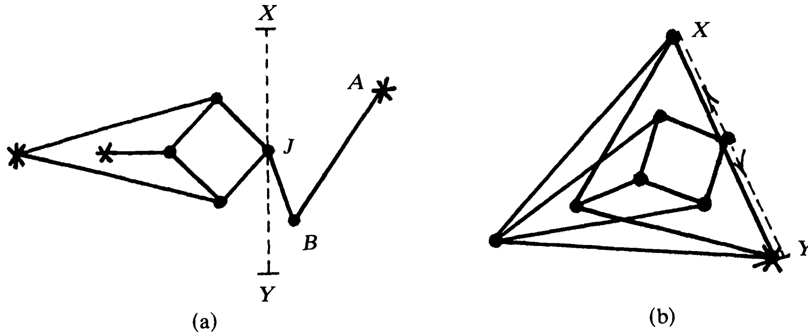


FIG. 2.2.4. (a) *An arm ABJ whose endpoint J is moving in a slot.* (b) *A slot on a platform.*

that is free to move in the plane. In this situation the slot itself has allowable motions, and we say that the slot is on a "platform". (See Fig. 2.2.4.)

2.3. Translators and rotators. We will need linkages to perform certain basic tasks. Since many of the previously published constructions have deficiencies of the sort described earlier, we include correct versions of these linkages. We do not attempt to construct the simplest linkage for a task, but rather one that is conceptually easy to understand and to prove correct. Throughout this section, we assume that R is a given closed bounded planar region.

The first device we construct is a *translator.* A translator is a linkage such that the only restriction on the movement of four of its joints S, T, U and V in the region R is that the position of T relative to S remains the same as the position of V relative to U. Alternatively, any three of these joints can be moved freely, and the position of the fourth joint is uniquely determined by the above relation and the position of the other three.

The linkage consisting of four parallelograms shown in Fig. 2.3.1 is a natural candidate for a translator. Joints S, T and U can be moved to any three points in the

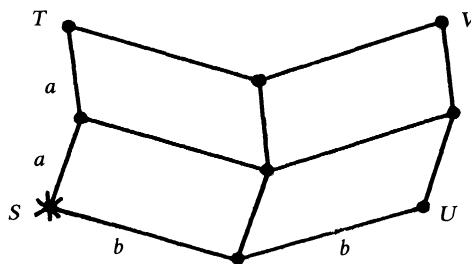


FIG. 2.3.1. *A faulty translator.*

plane, provided the distance between S and T does not exceed $2a$ and the distance between S and U does not exceed $2b$. At first it appears that the position of V relative to U is always the same as the position of T relative to S, i.e., that the vector ST is equal to the vector UV. The difficulty is that one or more of the parallelograms may convert to a contraparallelogram (see Fig. 2.3.2), and thus other motions are possible.

One might attempt to overcome this difficulty by attaching to each diagonal of the parallelograms a sufficiently short two-link segment. This would keep a

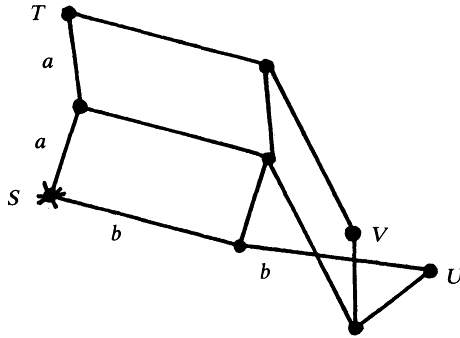


FIG. 2.3.2. *Conversion of a parallelogram to a contraparallelogram.*

parallelogram from straightening. Unfortunately, this also prevents the movement of T to S when S is held fixed, and this motion is essential in a construction of Kempe's that we use. We solve the problem by using a more complex device involving nine parallelograms. The linkage shown in Fig. 2.3.3 will be part of this device. The lengths of the links A_1B_1, B_1C_1 and C_1D_1 can be chosen long enough so that no matter where A_1 is positioned inside the bounded region R, D_1 can move freely in R while A_1 is kept fixed and C_1 is constrained to move on a line l through A_1 by means of a *slot* (see § 2.2). In fact, if the links are sufficiently long, then the slot can be constructed so that the angles between l and links A_1B_1 and B_1C_1 do not exceed 30° no matter how D_1 moves in R. (Of course the joints B_1 and C_1 are outside R, but this does not concern us, as only the joints of the original linkage are required to stay inside R.) Also note that the angle between C_1D_1 and l can be kept to at most 30° by the addition of a two-link segment connecting A_1 to D_1 and having length equal to the diameter of R. A similar linkage with joints A_1, A_2, A_3 and A_4 can be constructed so that A_4 can move freely in R while the angles between A_1A_2, A_2A_3 and A_3A_4 and another line l' through A_1 are kept within 30°. It is convenient to choose l' perpendicular to

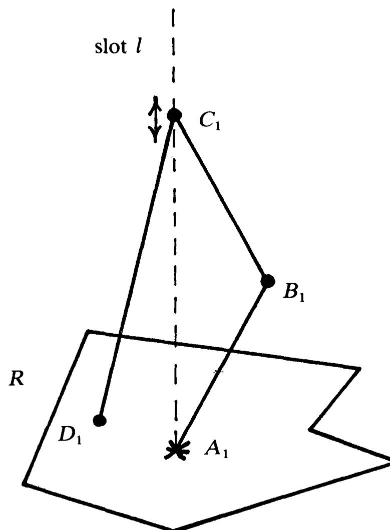


FIG. 2.3.3. *Keeping links nearly "vertical". (The two-link connection between A_1 and D_1 is not shown.)*

l since the links in the segments between A_1 and D_1 and between A_1 and A_4 will appear as sides of parallelograms in our nine-parallelogram translator. The fact that these links can be kept nearly parallel to l and l' will prevent any of the nine parallelograms from straightening. In this way, we avoid the flaw in the faulty four-parallelogram translator.

To construct the main body of the translator, begin with the nine-parallelogram linkage shown in Fig. 2.3.4. The three-link segments connecting A_1, B_1, C_1 and D_1

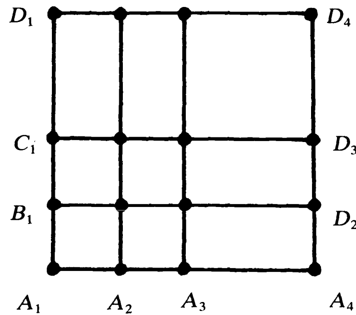


FIG. 2.3.4. *The predecessor of a translator.*

and A_1, A_2, A_3 and A_4 are not yet constrained as described above. Notice (by applying the parallelogram law of vector addition) that it is possible to move these segments independently of each other without breaking links or creating contraparallelograms, although parallelograms may straighten. As long as no contraparallelograms are created, the position of D_1 relative to A_1 is the same as the position of D_4 relative to A_4. Now move D_1 and A_4 (and hence D_4) to A_1, as shown in Fig. 2.3.5, and then attach the constraining devices described in the discussion of Fig. 2.3.3 to A_1, B_1, C_1 and D_1 and to A_1, A_2, A_3 and A_4. Joints D_1 and A_4 can still move freely in R, but the links in the segments between A_1 and D_1 and between A_1 and A_4 must remain nearly parallel to l and l', preventing the formation of contraparallelograms.

The next device we construct is called a *rotator*. A rotator is a linkage such that the only restriction on the movement inside of R of three of its joints A, I and H is that the distance from A to I be equal to the distance from A to H. In this construction,

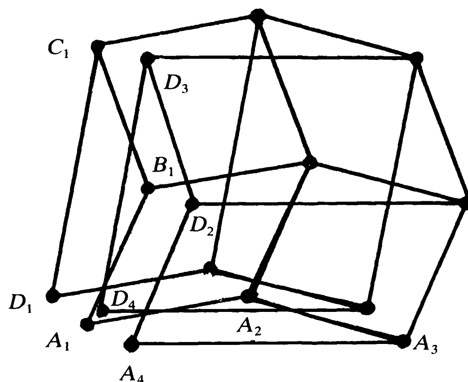


FIG. 2.3.5. *A translator. (The constraining devices attached to* $A_1, \cdots, D_1$ *and* $A_1, \cdots, A_4$ *are not shown. The picture is planar, not 3-dimensional.)*

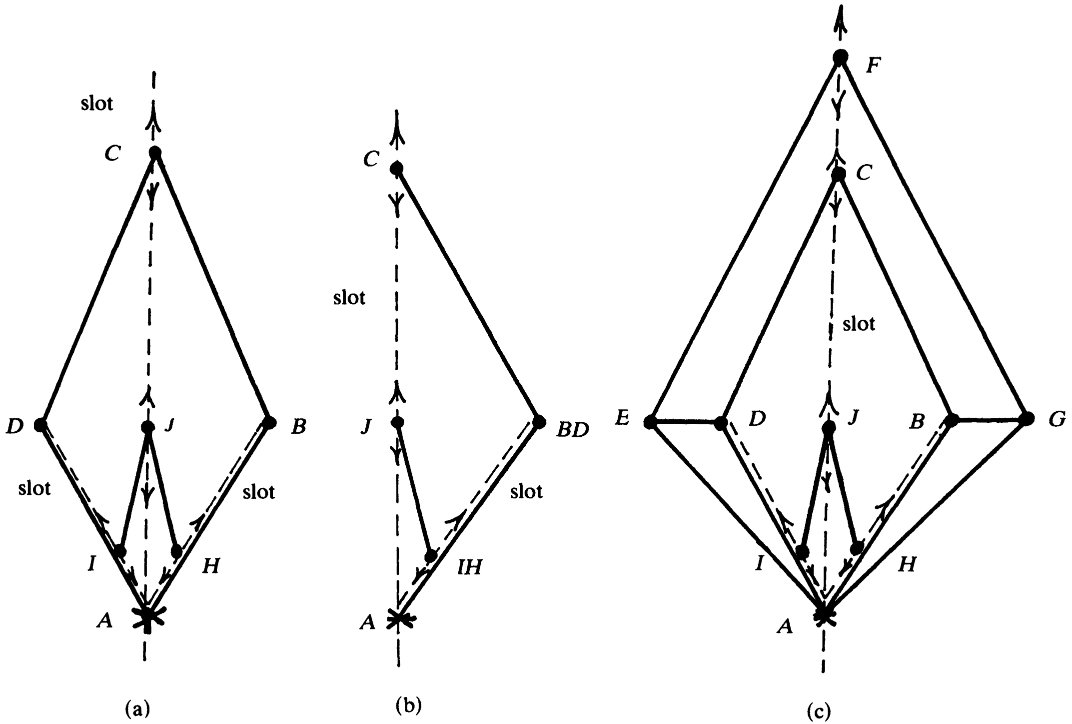


FIG. 2.3.6. *A distance rotator. Without the addition of the quadrilateral AGFE, D could move to B and the two superimposed joints could then move to the same side of the slot through A and C. Hence the slot through A and C would no longer bisect angle DAB.*

we begin with the quadrilateral linkage $ABCD$ shown in Fig. 2.3.6a. The lengths of the sides of $ABCD$ satisfy $|AD| = |AB| < |CD| = |CB|$. Then we constrain C to a slot through A. We want to insure that the slot through A always bisects the angle DAB. We also want to insure that links AD and AB can rotate freely about A, so it is necessary that $ABCD$ be able to straighten to allow links AB and AD to cross over each other. However, when B coincides with D, B and D must not be allowed to simultaneously move off the line AC in the same direction (see Fig. 2.3.6b). If this happens the slot through A would no longer bisect the angle DAB. To solve this problem, we construct another quadrilateral $AGFE$ with link lengths satisfying $|AG| = |AE| < |FG| = |FE|$ and also $|AE| + |EF| > |AD| + |DC|$. Then we constrain F to move in the slot through A in which C moves. Finally, we join the quadrilaterals by adding links ED and BG (see Fig. 2.3.6c). Now D and B can rotate freely about A (for an appropriately designed slot), but $ABCD$ must be straight whenever B and D coincide. Hence, B and D cannot move to the same side of the slot through A, and the slot remains the bisector of angle DAB.

Now we attach "platforms" to AD and AB (as shown in Fig. 2.2.4), and slots that coincide with AD and AB. We then add links IJ and HJ, where I is constrained to move in the slot along AD, H is constrained to move in the slot along AB, and J is constrained to the slot in which C and F move. Since triangles AIJ and AHJ are always congruent, the distance between A and I must equal the distance between A and H. Note that this is the only constraint on the motion of I and H. This completes the construction of a rotator.

Now we combine a translator that keeps the relative position of T to S equal to the relative position of V to U (but does not otherwise restrict their motions inside R) with a rotator that keeps the distance between A and H equal to the distance between A and I (but does not otherwise constrain their motions inside R). We do this simply by identifying A with U and H with V. The result is a linkage containing four joints S, T, $A = U$ and I whose motions inside R must satisfy only one requirement, that the distance between S and T be equal to the distance between $A = U$ and I. We call this device a *distance copier.*

A distance copier can be used to construct an *angle adder.* An angle adder is a linkage containing four equal-length links OA, OB, OC and OD whose motions are constrained only by the requirement that angle AOD be equal to angle AOB plus angle AOC. We will only need a device that correctly adds angles AOB and AOC when angle AOC is less than π. To construct such a device, we take four equal-length links OA, OB, OC and OD and attach a distance copier to A, B, C and D that keeps the distance between A and C equal to the distance between B and D. Then to insure that angle AOC is added to angle AOB rather than subtracted from it, we add two links OE and EB to form a triangle with a right angle at O. Now we connect E to D with a two-link segment of length $|EB|$ (see Fig. 2.3.7). These additional links constrain D to be on the correct side of the line OC.

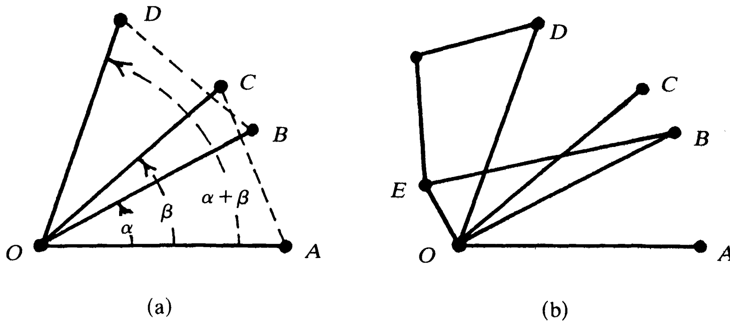


FIG. 2.3.7. *An angle adder. In* b), *the right triangle EOB has been added, together with a two-link segment connecting E to D of length* $|EB|$.

2.4. Linkages for multiplication. In the late 1800's Kempe [K-1876] showed how to construct linkages to "solve" multivariable polynomial equations. We will make important use of a modified version of his construction. Given a set of variables $x_1, x_2, \cdots, x_n$ with bounded domains and a polynomial equation $p(x_1, x_2, \cdots, x_n) = 0$, we can design a linkage that will force the x_i to satisfy the equation.

Consider the links AB and BC of equal length shown in Fig. 2.4.1. Joint A is fastened to the plane, and joint C moves in a slot on the x-axis. The position of C represents the value of a variable x whose domain is determined by the slot. The length of the slot is such that B cannot straighten. Additional links, whose description we omit, can be added to insure that AB remains vertical when $x = 0$. Then $x = a \cos \alpha$. Thus for a fixed value of a greater than $\max |x|$ the variable x can be represented by the angle α, where $0 < \alpha < \pi$.

We can now rewrite the polynomial equation, expressing each x_i as $a \cos \alpha_i$. Replace products of cosines by cosines of sums of angles using the formula

$$\cos \alpha \cos \beta = \tfrac{1}{2} (\cos (\alpha + \beta) + \cos (\alpha - \beta)),$$

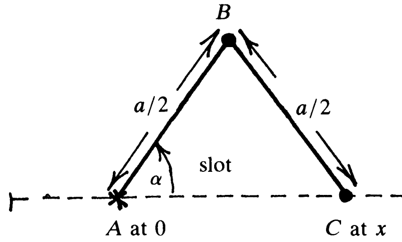


FIG. 2.4.1. *Representing x by angle α*: $x = a \cos \alpha$. Max $|x| < a$ *so* $0 < \alpha < \pi$. *The linkage that keeps AB vertical when* $x = 0$ *is not shown.*

thereby reducing the equation to the form

$$a_0 + \sum a_i \cos \theta_i = 0,$$

where each θ_i is a sum of α_i's.

Using the technique for adding angles described in the previous section, we can design a linkage that constructs each θ_i from the α_i's. Recall that the construction for adding angles works correctly as long as the second summand is in the range $[0, \pi]$, and since joint B cannot straighten, this condition is satisfied by the α_i's. The terms $a_i \cos \theta_i$ can be summed by constructing a sequence of links $L_1, L_2, \cdots$ of lengths $a_1, a_2, \cdots$ connected together at their end points and making each link L_i form the angle θ_i with the horizontal by using a translator. The translator is attached to the end points of L_i and to the endpoints of another link of length a_i that is rigidly attached to the moving side of the angle θ_i. Finally, the free joint of the last link is constrained to a slot on the vertical line $x = -a_0$.

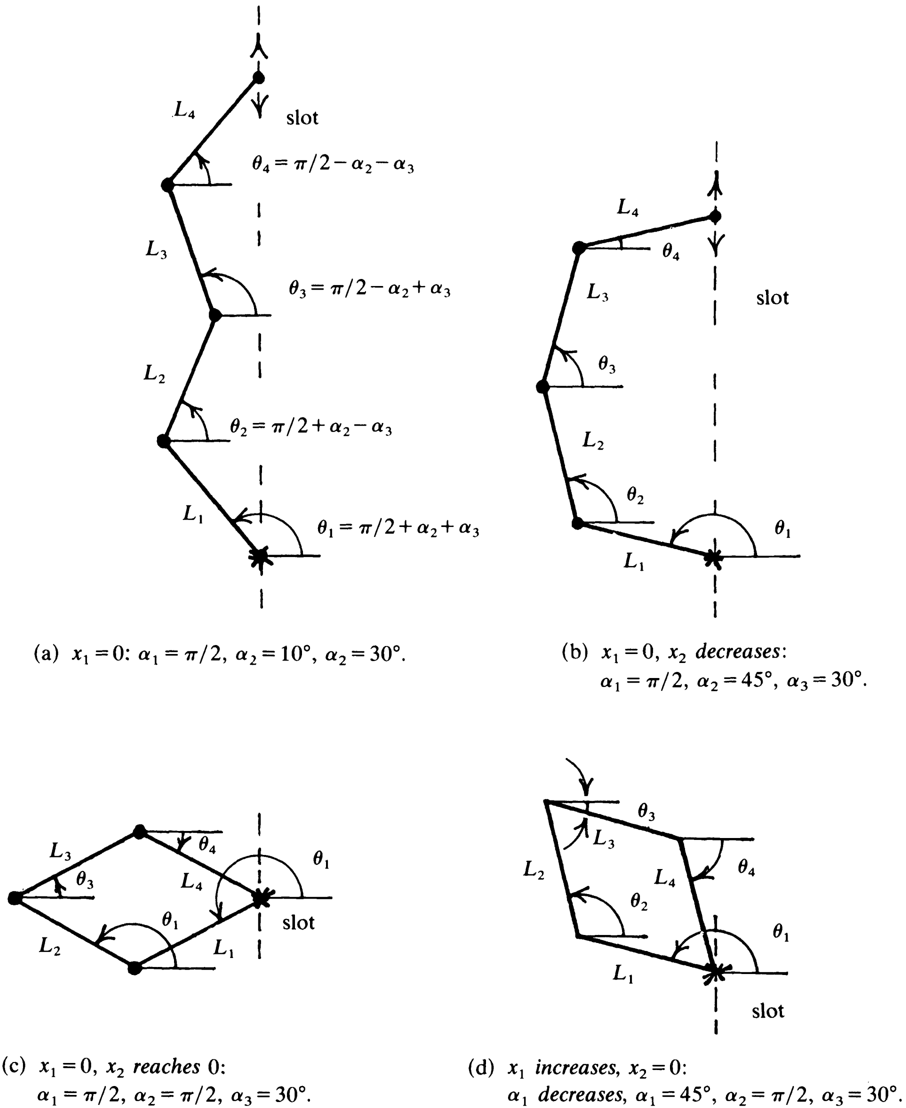
Note that for all motions of the linkages, $p(x_1, \cdots, x_n) = 0$. Furthermore, for each choice of x_i's solving the equation, the linkage can move to a configuration that represents this choice. The number of links in the straightforward implementation of Kempe's idea can be exponential in n because the summation may have exponentially many terms. However, we need the Kempe construction to enforce only two particular equations, so this problem does not concern us.

One of the equations that we are interested in is $x_1 x_2 x_3 = 0$. This equation states that at least one of x_1, x_2 or x_3 must be zero. Substituting $\cos \alpha_i$ for x_i reduces the equation to

$$\cos(\alpha_1 + \alpha_2 + \alpha_3) + \cos(\alpha_1 + \alpha_2 - \alpha_3) + \cos(\alpha_1 - \alpha_2 + \alpha_3) + \cos(\alpha_1 - \alpha_2 - \alpha_3) = 0.$$

Note that in terms of the previous notation, $a_0 = 0$ and $a_1 = a_2 = a_3 = a_4 = 1$. Figure 2.4.2 is a simplified picture of how this equation can be mechanically solved. If $x_1 = 0$, meaning that $\alpha_1 = \pi/2$, then the links L_1 and L_4 must be oriented so that $\theta_4 = \pi - \theta_1$. Similarly links L_2 and L_3 must be oriented so that $\theta_3 = \pi - \theta_2$. If $x_2 = 0$ then L_1 and L_3 must be parallel, as must L_2 and L_4. All these conditions are met when $x_1 = 0$ and $x_2 = 0$. At this point the shape of the figure changes from that in (a) and (b) to a parallelogram, shown in (c).

3. Replacing the boundaries for a convex region. Suppose that L is a linkage and R is a closed bounded region whose boundary is a convex polygon. We will show that by adding additional links to L we can constrain L to the region R without destroying any motions of L that were totally within R. However, the new links may move outside R.



(a) $x_1 = 0$: $\alpha_1 = \pi/2$, $\alpha_2 = 10°$, $\alpha_2 = 30°$.

(b) $x_1 = 0$, x_2 *decreases*: $\alpha_1 = \pi/2$, $\alpha_2 = 45°$, $\alpha_3 = 30°$.

(c) $x_1 = 0$, x_2 *reaches* 0: $\alpha_1 = \pi/2$, $\alpha_2 = \pi/2$, $\alpha_3 = 30°$.

(d) x_1 *increases*, $x_2 = 0$: α_1 *decreases*, $\alpha_1 = 45°$, $\alpha_2 = \pi/2$, $\alpha_3 = 30°$.

FIG. 2.4.2. *Links for the polynomial* $x_1x_2x_3 = 0$. $\theta_1 = (\alpha_1 + \alpha_2 + \alpha_3)$, $\theta_2 = (\alpha_1 + \alpha_2 - \alpha_3)$, $\theta_3 = (\alpha_1 - \alpha_2 + \alpha_3)$, $\theta_4 = (\alpha_1 - \alpha_2 - \alpha_3)$.

The region R is the intersection of a finite number of half-planes. By adding constraining linkages to force the original linkage L to lie in each half-plane, we can force the linkage L to lie within the intersection of the half-planes and hence within the convex polygon. For each edge of the polygon and each joint J of L we construct a modified Peaucellier device that constrains J to the appropriate side of the edge (see Figs. 2.2.3 and 3.1). The device does not interfere with the motion of J inside the polygon provided that the polygon avoids the discs centered at X and Y. Clearly, this will constrain the linkage L to remain within R. However, we must show that we have not restricted the allowable motions of L. As pointed out earlier at the end of § 2.1, identifying joint J_1 of one linkage with joint J_2 of another may restrict the movement

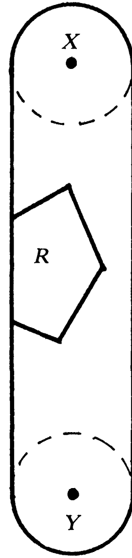


FIG. 3.1. *Polygon inside the reachable region of a modified Peaucellier device.*

of $J_1 = J_2$ to a region smaller than the intersection of the original reachable regions of J_1 and J_2. In fact, the intersection may not even be a connected region. Recall that the subtle point that one must consider is that even when the intersection is connected, the joints still may not be able to reach all points in the intersection since the possible paths the joints can follow may not be compatible.

However, in this particular case, each Peaucellier device constraining J to a region bounded in part by a side of R allows J to move along any curve in R. Hence the allowable motions of a joint J of L are not restricted by the addition of the devices.

The number of Peaucellier devices needed is equal to the product of the number of joints of L and the number of sides of the polygon. The lengths of the links in each Peaucellier device can be determined simply by choosing an appropriate scaling factoı for Fig. 2.2.3. Consequently, the description of the new linkage is polynomial in the size of the description of the original linkage L and region R.

4. Replacing the boundaries for a nonconvex region.

4.1. Overview. In this section we show how to incorporate into a linkage L the boundaries of an arbitrary bounded region R whose boundary consists of a finite number of straight line segments. Here two problems arise. First, the region is not simply the intersection of half-planes. Second, constraining the end points of a link to be in a region does not necessarily constrain the entire link to be in the region. In Fig. 4.1.1, even if A and B are constrained to lie within the region R, the link AB may be partially outside the region.

To handle these problems, let H be the convex hull of the region R. The region H–R can be quickly partitioned into a small set of triangles, the number of triangles being polynomial in the number of line segments in the boundary. (See Eves [E-63].) If we can exclude a link from a triangle without otherwise restricting its motion, then we can restrict a link to R without restricting its motion. (To keep a link from sliding along an edge of the triangulation, we can cover the edge with two additional triangles

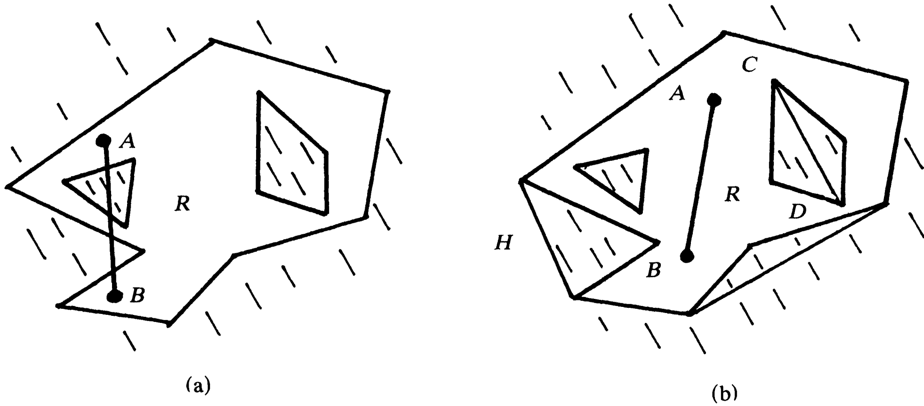


FIG. 4.1.1. *A link with endpoints in a nonconvex region. The link must be kept out of each triangle in the triangulation of H–R. In* (b), *CD is an edge of the triangulation.*

that lie in $H-R$ and then require that the link remain outside those triangles also.) Applying the construction to each link of L will solve the problem.

In § 4.2 we describe a linkage for tracing a triangle and then use this construction in § 4.3 for constraining a link to remain outside a triangle.

4.2. A linkage for tracing a triangle. In order to construct a linkage that can reach all points in the closed exterior of a triangle, we begin by constructing a linkage that traces the boundary of a triangle. Suppose that we are given a triangle XYZ. Then we can construct three straight-line motion linkages with designated joints A, B and C such that the joints A, B and C move along the segments XY, YZ and XZ respectively (see Fig. 4.2.1). We would like to construct a fourth linkage with a

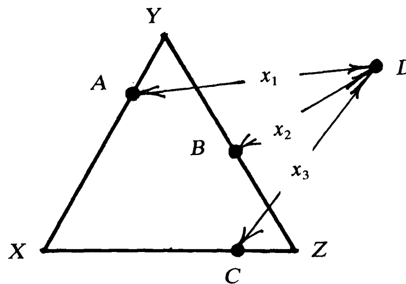


FIG. 4.2.1. *Forcing D to trace the boundary of a triangle XYZ.*

designated joint D such that D must be at the same position as either A, B, or C. Then provided D can move freely subject to the above constraint, we will have constructed a linkage that traces the triangle XYZ.

We force D to be at the same position as either A, B, or C by using Kempe's construction as presented in § 2.4. Let x_1, x_2 and x_3 denote the distances from D to the joints A, B and C respectively. Joint D is at the same location as one of A, B, or C provided $x_1x_2x_3=0$. For convenience the distances x_1, x_2 and x_3 can be translated to the x-axis by means of distance copiers. Adding the linkage to force $x_1x_2x_3=0$ then completes the construction.

4.3. Constraining a link to remain outside a triangle. We now construct a linkage to constrain the motions of a link so that it can move freely outside a triangular region. Consider the triangle XYZ shown in Fig. 4.3.1. The triangle is inside a triangular

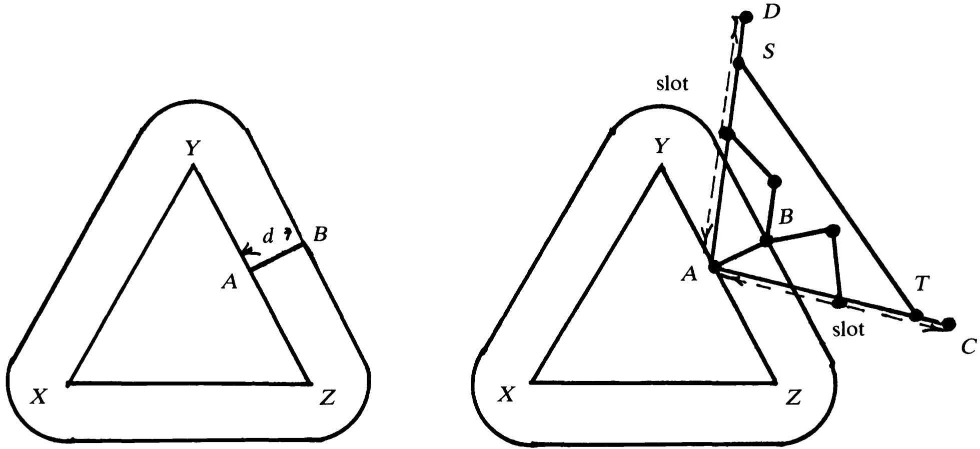


FIG. 4.3.1. *Constraining a link to remain outside a triangle.*

figure with rounded corners. The distance between parallel edges of the inner triangle and the outer is d. The corners of the outer triangle have been replaced by circular arcs of radius d centered at the vertices of the inner triangle.

Using the construction given in § 4.2, we can constrain a joint A to the boundary of XYZ and using a similar construction we can constrain a joint B to the boundary of the outer triangular figure. We can connect A to B with a link AB of length d. The possible motions of the link AB consist of rotating about the inner triangle but always remaining perpendicular to an edge, except at the vertices. At a vertex, the link AB can rotate from a position perpendicular to one edge to a position perpendicular to the other.

We now add two additional links AC and AD at joint A and two-link segments connecting B to AC and AD. The lengths of the segments when fully extended are designed to force the angles BAD and BAC to be in the range $[-\pi/2, \pi/2]$. Thus AD and AC are forced to lie outside triangle XYZ.

Attached to the links AD and AC are platforms that contain slots coinciding with AD and AC. The end points of the link ST that we wish to exclude from the triangle XYZ move in these slots. Clearly, ST can never enter the triangle since its end points are always in a half-plane whose boundary is a line through A perpendicular to AB. The triangle XYZ is outside this half-plane and thus, by convexity, ST does not intersect the triangle XYZ.

We must show that the motions of link ST are not further restricted as long as ST does not move far from the triangle. Since ST is completely contained within a half-plane associated with the perpendicular to AB passing through A, we can fix A at any point on the triangle and move T by rotating D about A and sliding T along AD. The movement of S is obtained by analogous use of AC. Since S and T are confined to slots along AD and AC, the lengths of AD and AC must be long enough to allow S and T to move as desired.

5. PSPACE-hardness of the reachability problem for linkages.

5.1. Overview. We now show that the reachability problem for planar linkages is PSPACE-hard. That is, given an initial configuration of an arbitrary planar linkage L, a joint J in that linkage and a point p in the plane, the question of whether L can be moved so that J reaches p is PSPACE-hard.

Our proof consists of showing that there are linkages that are capable of simulating linear bounded automaton (LBA) computations and that the size of the description of a linkage that simulates a given LBA on inputs of length n is linear in n and the size of the description of the LBA. The PSPACE-hardness of the linkage reachability problem then follows from the fact that the acceptance problem for LBA's is PSPACE-complete. For definitions of an LBA and PSPACE see [HU-79].

5.2. Some useful linkages. We begin by building up a collection of simple devices that perform various functions. First, we define a *cell* to be a horizontal slot of some fixed size containing a joint. The joint represents the value of a Boolean variable. The left end of the slot indicates value 0, the right end value 1. Certain cells will be grouped together to form *registers.*

It is convenient to have a device called a *lock* that can be used to force the value of each cell in a register to be 0 or 1 and to prevent the value of any cell from changing during certain time periods. Figure 5.2.1 shows a lock attached to a register. The

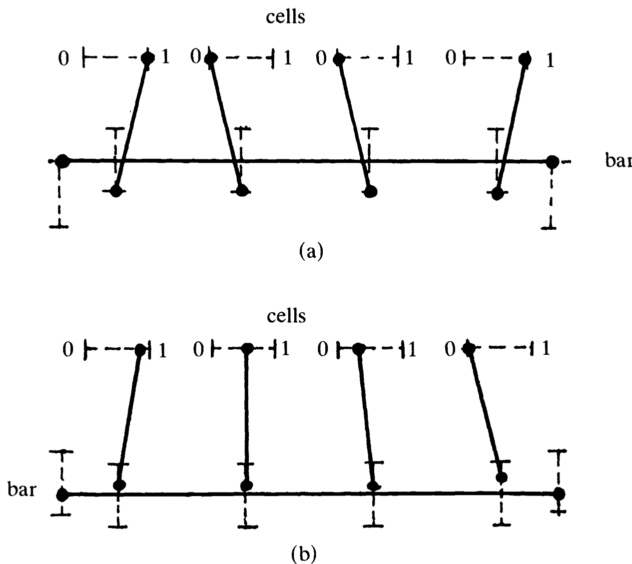


FIG. 5.2.1. *A lock on cells of a register.* (a) *Locked.* (b) *Unlocked.*

horizontal rectangular bar is part of the lock. The bar is attached to slots so that it can only move vertically; no rotation is possible. Attached to the bar are a number of platforms carrying vertical slots. Each joint representing a Boolean variable is attached to a link, the other end of which moves in one of the vertical slots. The links are designed so that when the bar is in the lower, unlocked position the Boolean variable joint can move freely in its cell because the other end of the link can move up and down the vertical slot. When the bar is in the upper, locked position each Boolean variable joint is in a 0-1 position. Note that these variable joints cannot move when the bar is up.

In order to coordinate the linkage motions that take place during the simulation of two moves of the LBA, we design a *sequence controller* with five variables s_1, s_2, s_3, l_1 and l_2. Each variable is represented as a joint in a slot. We restrict the values that the variables can assume by adding a Kempe linkage to force

$$[s_1^2+s_2^2+s_3^2+(1-l_1)^2][s_2^2+s_3^2+(1-l_1)^2+(1-l_2)^2]\cdots[s_1^2+s_2^2+(1-l_1)^2+l_2^2]=0.$$

TABLE 5.2.1
Possible values for variables. Dashes denote arbitrary values between 0 *and* 1.

s_1	s_2	s_3	l_1	l_2
0	0	0	1	–
–	0	0	1	1
1	0	0	–	1
–	0	0	0	1
0	0	–	0	1
0	0	1	–	1
0	–	1	1	1
0	1	1	1	–
0	–	1	1	0
0	0	–	1	0

The consequence of this equation is that the only possible values the variables can assume are those shown in Table 5.2.1. The restriction on the value of the variables allows only one variable to change value at a time, and the variable that can change value is determined by the values of the remaining variables. As a result, the only allowable sequence of changes from one set of 0–1 values to another is that shown in Fig. 5.2.2. Of course, the changes can reverse at any time. We will use the values of

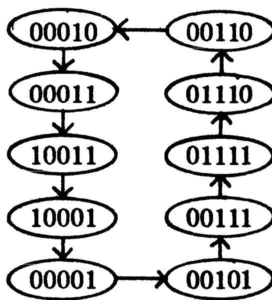


FIG. 5.2.2. *The allowable sequence of values.*

these variables to control certain events, thereby sequencing the order in which the events can take place. In particular, the variables l_1 and l_2 will control locks, and the s's will sequence the order in which these locks are opened and closed.

Since we represent values of variables by positions of joints, we will often use the words "joint" and "variable" interchangeably. Also, we will denote a variable and the joint that represents it with the same symbol.

The next device we need is a gate for NOT and a gate for AND. To obtain negation, we use the distance copier of § 2.3 to force the distance of a joint from one end of a slot to be the same as the distance of another joint from the opposite end of its slot. Thus when one cell has value 0, the other has value 1 and vice versa.

To construct an AND gate, we force the product of the distances of two joints from the 0-end of their slots to equal the distance of a third joint from the 0-end of its slot. Let x_1, x_2 and x_3 be these distances. Then when x_1 and x_2 both have 0–1 values, $x_3 = x_1 \text{AND} x_2$.

Using these linkages it should be clear that we can construct a linkage to compute any Boolean function. However, to make it easy to check the correct behavior of the linkage we must be careful not to form a loop by using the output of a gate as an input to one of its predecessors. This might cause the linkage to be rigid since the loop might imply a relationship between the rates of motion of certain joints that would not be satisfied for any nonzero rate. Our design will contain only two loops, and we will use a decoupling mechanism with them to insure that the entire linkage does not jam.

5.3. Simulation of an LBA. One idea for a mechanical simulation of a given LBA, *M*, is the following. Suppose that we have two registers that can be used for storing instantaneous descriptions (ID's) of *M*. Since Boolean variables are modeled by joints moving in slots, the contents of a register at a given time will not necessarily be a sequence of 0's and 1's. However, we will design a linkage connecting these two registers so that whenever the contents of both registers are sequences of 0's and 1's (i.e., whenever both registers contain ID's), the ID in one represents the result of a legal move of *M* from the ID in the other. We would also like the linkage to have the property that as *M* makes its moves, its ID's appear alternately in one register and then in the other. In this way, we can simulate the operation of an LBA. Since we are only interested in the reachability problem, however, we do not need to build a linkage that actually simulates *M*; rather, we only need a linkage which is *able* to simulate *M*. The linkage could make other moves as well, provided that it never moved a certain joint *J* to a point *p* representing an accepting state of *M* by accident. The linkage we are about to construct *can* simulate *M*, but in addition, it can undo and then redo sequences of moves. Because of this we will assume that *M* is deterministic and has no move from any accepting state.

We begin the construction with the two registers R_1 and R_2 used to store the ID's of *M*. Attached to the cells of the registers are two Boolean circuits constructed from NOT and AND gates. The output f_1 is true whenever the ID in register R_2 follows from the ID in register R_1 by one move of the LBA. The output f_2 is true whenever the ID in register R_1 follows from the ID in the register R_2 by one move of the LBA.

The variables l_1 and l_2 in the sequence controller described in § 5.2 are connected to locks on registers R_1 and R_2, respectively, with $l_i = 1$ when its lock is in the closed position. The variables s_1 and s_2 are connected to the outputs f_1 and f_2 by the linkage in Fig. 5.3.1. Joints f_1 and f_2 are free to move when s_1 and s_2 are 0. However, s_1 or s_2 can move to value 1 only if f_1 or f_2, respectively, has value 1.

Initially R_1 holds the configuration of the LBA at time zero, and $s_1 = 0$, $s_2 = 0$, $s_3 = 0$, $l_1 = 1$ and $l_2 = 0$. This corresponds to the first entry in Table 5.2.1.

We now describe a sequence of events that simulates the behavior of *M* on a given input. Since l_2 is initially 0, the variables in R_2 can move freely. In particular, they can move to the ID of *M* after its first move. Then l_2 can move to a locked

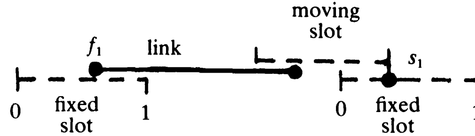


FIG. 5.3.1. *Decoupling mechanism. The moving slot has one end point attached to* s_1, *which moves in a fixed slot.* s_1 *cannot move to* 1 *unless* f_1 *is at* 1.

position, i.e., l_2 can take on the value 1. Note that as variables in R_2 were changing values, f_1 and f_2 were also changing, but this is allowed since $s_1=0$ and $s_2=0$. At this point, the sequence controller has advanced to the second state shown in Table 5.2.1 and can now advance to the third state, with $s_1=1$. This is because f_1 must be 1 since the ID in R_2 follows from the ID in R_1 by one move of M. Hence s_1 can move to 1.

Next R_1 unlocks, allowing s_1 to return to zero. (Note that s_i can change to zero independently of f_i's value.) Now the variables in R_1 can change to the next ID of M. Again, f_1 and f_2 must be changing while R_1 is changing, but this is permitted since s_1 and s_2 have value 0. At this point, the variable s_3 can change to 1 and then l_1 can lock. The next step is for s_2 to change value to 1. This is allowed because f_2 has value 1: the configuration in R_1 follows from the configuration in R_2 by one move of M. As soon as s_2 changes value to 1, then l_2 can unlock, and s_2 can change back to 0. Finally, s_3 can change back to 0, completing a cycle of the sequence controller. During the cycle the linkage has simulated two moves of the LBA.

Observe that the simulation may proceed forward or backward. If the simulation proceeds from ID_1 to ID_2 and then reverses, it may back up into an ID other than ID_1 since two ID's may both have the same successor ID. The only concern here is that the simulation might accidentally back into an accepting ID. This can be prevented by modifying the LBA so that no move is possible from any ID with an accepting state and then basing the design of the linkage on the modified LBA. Note that the linkage may back into a configuration corresponding to an ID of M that could not be reached from its initial state. However, since we are only considering deterministic LBA's, the linkage must move forward along the same computational path on which it has just backed up. Of course its forward progress may be interrupted from time to time by additional backing up and retracing of sequences.

Finally, another Boolean circuit is attached to the two registers R_1 and R_2. The Boolean circuit computes a 1 output whenever one of the registers is locked and contains an accepting ID. The output of this circuit is a joint J. There is a motion of the linkage that moves J to 1 if and only if the LBA reaches an accepting ID.

This completes the proof that the reachability problem for planar linkages is PSPACE-hard.

REFERENCES

[E-63] HOWARD, EVES, *A Survey of Geometry*, Allyn and Bacon, Boston, MA, 1963.

[HCV-52] D. HILBERT AND F. COHN-VOSSEN, *Geometry and the Imagination*, Chelsea, New York, 1952

[HJW-82A] JOHN HOPCROFT, DEBORAH JOSEPH AND SUE WHITESIDES, *On the movement of robot arms in 2-dimensional bounded regions*, TR 82-486, Computer Science Dept., Cornell Univ., Ithaca, NY, April 1982.

[HJW-82b] ———, *Determining the points of a circular region reachable by joints of a robot arm*, TR 82-516, Computer Science Dept., Cornell Univ., Ithaca, NY, 1982.

[HU-79] JOHN HOPCROFT AND JEFFREY D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[K-1876] A. B. KEMPE, *On a general method of describing plane curves of the nth degree by linkwork*, Proc. London Math. Soc., 7 (1876), pp. 213–216.

[L-80] TOMAS, LOZANO-PEREZ, *Automatic planning of manipulation transfer movements*, A.I. Memo 606, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, December 1980.

[LW-79] TOMAS LOZANO-PEREZ AND MICHAEL A. WESLEY, *An algorithm for planning collision-free paths among polyhedral obstacles*, Comm. ACM, 22 (1979), pp. 560–570.

[S-81] JACOB T. SCHWARTZ AND MICHA SHARIR, *On the 'piano movers' problem* I. *The case of a two-dimensional rigid polygonal body moving amidst polygonal barriers*, TR 39, Dept. of Computer Science, New York Univ., New York, October 1981.

[S-82] ———, *On the 'piano movers' problem* II. *General techniques for computing topological properties of real algebraic manifolds*, TR 41, Dept. of Computer Science, New York Univ., New York, February 1982.

[P-1864] M. PEAUCELLIER, *Correspondence*, Nouvelles Annales de Mathematiques, Ser. 2, 3 (1864), pp. 414–415.

[R-79] J. REIF, *Complexity of the movers' problem and generalizations*, in Proc. of the 20th IEEE Foundations of Computer Science Conference, Institute of Electrical and Electronics Engineers, New York, 1979, pp. 421–427.

SIAM J. COMPUT.
Vol. 13, No. 3, August 1984

© 1984 Society for Industrial and Applied Mathematics
011

ON THE OPTIMUM CHECKPOINT SELECTION PROBLEM*

SAM TOUEG† AND ÖZALP BABAOĞLU†

Abstract. We consider a model of computation consisting of a sequence of n tasks. In the absence of failures, each task i has a known completion time t_i. Checkpoints can be placed between any two consecutive tasks. At a checkpoint, the state of the computation is saved on a reliable storage medium. Establishing a checkpoint immediately before task i is known to cost s_i. This is the time spent in saving the state of the computation. When a failure is detected, the computation is restarted at the most recent checkpoint. Restarting the computation at checkpoint i requires restoring the state to the previously saved value. The time necessary for this action is given by r_i. We derive an $O(n^3)$ algorithm to select out of the $n-1$ potential checkpoint locations those that result in the smallest expected time to complete all the tasks. An $O(n^2)$ algorithm is described for the reasonable case where $s_i > s_j$ implies $r_i \geqslant r_j$. These algorithms are applied to two models of failure. In the first one, each task i has a given probability p_i of completing without a failure, i.e., in time t_i. Furthermore, failures occur independently and are detected at the end of the task during which they occur. The second model admits a continuous time failure mode where the failure intervals are independent and identically distributed random variables drawn from any given distribution. In this model, failures are detected immediately. In both models, the algorithm also gives the expected value of the overall completion time and we show how to derive all the other moments.

Key words. fault-tolerance, checkpoint, rollback-recovery, discrete optimization, renewal process

1. Introduction. A variety of hardware and software techniques have been proposed to increase the reliability of computing systems that are inherently unreliable. One such software technique is *rollback-recovery*. In this scheme, the program is *checkpointed* from time to time by saving its state on secondary storage and the computation is restarted at the most recent checkpoint after the detection of a failure [6]. Between the times when the failure is detected and the computation is restarted, the computation must be *rolled back* to the most recent checkpoint by restoring its state to the saved value. Obviously, rollback-recovery is an effective method only against transient failures. Examples of such failures are temporary hardware malfunctions, deadlocks due to resource contention, incorrect human interactions with the computation, and other external factors that can corrupt the computation's state. Persisting failures will block the computation no matter how many times it is rolled back. The ability to detect failures is an essential part of any fault-tolerance method including rollback-recovery. Examples of such failure detection methods are integrity assertion checking [8] and fail-stop processors [10].

In the absence of checkpoints, the computation has to be restated from the beginning whenever a failure is detected. It is clear that with respect to many objectives such as minimum completion time, minimum recovery overhead, maximum throughput, etc., the positioning of the checkpoints involves certain tradeoffs. A survey of an analytical framework for resolving some of these tradeoffs is presented by Chandy [1]. Young [11] and Chandy *et al.*[3] addressed the problem of finding the checkpoint interval so as to minimize the time lost due to recovery for a never-ending program subject to failures constituting a Poisson process. Gelenbe and Derochette

* Received by the editors March 9, 1983, and in revised form September 13, 1983. This article was typeset by the authors at Cornell University on a UNIX™ system. Final copy was produced on November 3, 1983.

† Department of Computer Science, Cornell University, Ithaca, New York 14853. This research was supported in part by the National Science Foundation under Grants MCS 81-03605 and MCS 82-10356.

[5] and Gelenbe [4] have generalized this result to allow the possibility of external requests arriving during the establishment of a checkpoint or rollback-recovery. These requests are queued and serviced later. Consequently, these results for the optimum checkpoint intervals with respect to maximizing system availability and minimizing response time reflect the dependence on the rate of requests. More recently, Koren *et al.*[7] have derived expressions for optimum parameters in a system employing a combination of instruction retry and rollback-recovery.

In the previous work described above, expressions for the optimum checkpoint interval were derived with the assumption that checkpoints could be placed at arbitrary points in the program at a cost (as measured in units of time) that is independent of their position. Recall that establishing a checkpoint implies the saving of the current program state on secondary storage. As the minimum amount of data required to specify the state of a program can vary greatly over time, it is unrealistic to assume that the time necessary to write it to secondary storage is a constant. Furthermore, some programs cannot be blocked to create a checkpoint during the execution of certain intervals. The transaction processing periods of a database system are examples of such intervals. Whether due to prohibitive costs or other practical considerations, most computations display only a discrete set of points where checkpoints can be placed. Informally, we will view these computations as a sequence of *tasks* such that the program state between two consecutive tasks is "compact" (i.e., incurs a reasonable cost to save). In other words, task boundaries define potential checkpoint locations. Statically, such a computation can be represented as a directed graph where the vertices denote the tasks and an edge (i,j) exists if and only if task i may be followed by task j in some execution. This computation model was used by Chandy and Ramamoorthy in the optimum checkpoint selection problem where the objective was to minimize the maximum and expected time spent in saving states [2].

In this paper, we model the execution of a program as a linear sequence of tasks and consider the optimum checkpoint selection problem with respect to minimizing the expected total execution time of the program subject to failures. In the next section, we introduce the formal program model along with the input parameters to the problem. § 3 describes an algorithm based on dynamic programming that generates the set of checkpoint locations so as to minimize the expected completion time. § 4 gives an improved version of this algorithm that is applicable when there is a certain relation between the costs for establishing checkpoints and the costs of rolling back the computation. In § 5, we present two possible failure models to which the algorithm could be applied. Solutions to some extensions of the original problem are presented in § 6. A discussion of the results concludes the paper.

2. The model of computation. Assume that a computation consists of the sequential execution of n tasks where a task may be a program, procedure, function, block, transaction, etc. depending on the environment. For our purposes, any point in the execution where the computation is allowed to block and where its state can be represented by a reasonable amount of information can delimit a task. Clearly, the decomposition of the computation into tasks is not unique -- any convenient one will suffice.

Let t_i denote the time required to complete task i in the absence of failures. We assume that these times are deterministic quantities and are known for all of the tasks. The boundary between two consecutive tasks $i-1$ and i determines the ith candidate checkpoint location. The *setup cost*, s_i, is defined to be the time required

to establish a checkpoint at location i. This is the time necessary to save the state of the computation on secondary storage as it exists just before task i is executed. After a failure is detected, the state of the computation is restored to that of the most recent checkpoint. The *rollback cost*, r_i, is defined to be the time required to roll back the computation to the checkpoint at location i. After the rollback, the computation resumes with the execution of task i. We assume that a checkpoint is always established just before task 1. Initially, we also assume that the checkpoint setups and rollbacks are failure-free. We relax these last assumptions in § 6.

The optimization problem can now be stated as follows: Given t_i, s_i and r_i for $i=1,2,\cdots,n$ and a suitable failure model, select the subset of the $n-1$ potential checkpoint locations such that the resulting expected total completion time (including the checkpoint setup and the rollback-recovery times) for the computation is minimized. The objective function we have selected is a reasonable one for computations that govern time-critical applications such as chemical process control, air traffic control, etc., in the presence of failures.

The following definitions will be used in the subsequent sections. Let $[i,j]$ denote the sequence of tasks i, $i+1,\cdots,j$ where $j \geqslant i$. Note that $[1,n]$ is the entire computation. Let $T_{i,j}^m$ denote the minimum expected execution time for $[i,j]$ over all the possible checkpoint selections in $[i,j]$ with m or fewer checkpoints. Clearly, $T_{i,j}^0$ is the expected execution time of $[i,j]$ without any checkpoints. Among all the checkpoint selections in $[i,j]$ that achieve $T_{i,j}^m$, we consider those with the minimum number of checkpoints. These selections are called *m-optimal solutions* for $[i,j]$, and we denote them by $\mathbf{L}_{i,j}^m$. Note that a m-optimal solution $\mathbf{L}_{i,j}^m$ for $[i,j]$ contains at most m checkpoints. If $\mathbf{L}_{i,j}^m$ contains no checkpoints (i.e., it is the empty selection of checkpoints), it is written as $\mathbf{L}_{i,j}^m = <>$. If $\mathbf{L}_{i,j}^m$ contains k checkpoints ($1 \leqslant k \leqslant m$), we represent it as the ordered sequence of the selected checkpoint locations $\mathbf{L}_{i,j}^m = <u_1, u_2, \cdots, u_k>$, where $i < u_1 < u_2 < \cdots < u_k \leqslant j$. The *rightmost checkpoint location* of $\mathbf{L}_{i,j}^m = <u_1, u_2, \cdots, u_k>$ is u_k, and the rightmost checkpoint location of $\mathbf{L}_{i,j}^m = < >$ is i. There may be more than one m-optimal solution for $[i,j]$. From now on, we will consider only those m-optimal solutions $\mathbf{L}_{i,j}^m$ that satisfy the following additional requirement: Either $\mathbf{L}_{i,j}^m = < >$ or the rightmost checkpoint location of $\mathbf{L}_{i,j}^m$ is greater than or equal to the rightmost checkpoint location of any other m-optimal solution for $[i,j]$. Henceforth, we reserve the notation $\mathbf{L}_{i,j}^m$ to denote only those m-optimal solutions for $[i,j]$ that satisfy this additional requirement.

The next section describes an algorithm to determine an optimum checkpoint selection $\mathbf{L}_{1,n}^{n-1}$ and the corresponding minimum expected execution time $T_{1,n}^{n-1}$ for a given problem.

3. The basic algorithm. Consider $T_{1,j}^k$ and $T_{1,j}^{k-1}$ for some k and j such that $k \geqslant 1$ and $j \geqslant 2$. Note that either $T_{1,j}^k = T_{1,j}^{k-1}$ or $T_{1,j}^k < T_{1,j}^{k-1}$. Suppose $T_{1,j}^k < T_{1,j}^{k-1}$. In this case, any k-optimal solution for $[1,j]$ must contain exactly k checkpoints. Let h be the location of the rightmost checkpoint of a k-optimal solution for $[1,j]$. We must have

$$T_{1,j}^k = T_{1,h-1}^{k-1} + T_{h,j}^0 + s_h .$$

That is, up to $k-1$ checkpoints are optimally established in $[1,h-1]$ and a checkpoint is established at location h. No checkpoints are established in $[h,j]$. Let $\mathbf{L}_{1,h-1}^{k-1}$ be any $(k-1)$-optimal solution for $[1,h-1]$. Then

$$\mathbf{L}_{1,j}^k = \mathbf{L}_{1,h-1}^{k-1} \,||\, <h>$$

```
for i ← 1 until n do
   for j ← i until n do compute T^0_{i,j};

for k ← 1 until n−1 do
   begin
      T^k_{1,1} ← T^0_{1,1}
      L^k_{1,1} ← < >
   end;

for k ← 1 until n−1 do
   for j ← n step −1 until 2 do
      begin
         T ← MIN_{1<i≤j} (T^{k−1}_{1,i−1} + T^0_{i,j} + s_i)
         Let h be the largest minimizing index above
         if T<T^{k−1}_{i,j} then do
            begin
               T^k_{1,j} ← T
               L^k_{1,j} ← L^{k−1}_{1,h−1} || <h>
            end
         else do
            begin
               T^k_{1,j} ← T^{k−1}_{1,j}
               L^k_{1,j} ← L^{k−1}_{1,j}
            end;
      end;
```

FIG. 1. *Dynamic programming algorithm for computing the optimum checkpoint selection (and the corresponding expected execution time) for a n-task computation.*

must be a k-optimal solution $[1,j]$ (the $||$ operator denotes concatenation of sequences, i.e.,

$$\langle u_1, \ ,u_n\rangle \,||\, \langle v\rangle = \langle u_1,\cdots,u_n,v\rangle).$$

From these observations, it is clear that we can compute $T^k_{1,j}$ and $\mathbf{L}^k_{1,j}$ as follows. Let

$$T = \underset{1<i\leqslant j}{\mathrm{MIN}}\,(T^{k-1}_{1,i-1} + T^0_{i,j} + s_i)$$

and let h be the largest index such that $T = T^{k-1}_{1,h-1} + T^0_{h,j} + s_h$. We must have $T^k_{1,j} = \mathrm{MIN}(T, T^{k-1}_{1,j})$. If $T^k_{1,j} = T^{k-1}_{1,j}$ then we have $\mathbf{L}^k_{1,j} = \mathbf{L}^{k-1}_{1,j}$; otherwise $\mathbf{L}^k_{1,j} = \mathbf{L}^{k-1}_{1,h-1} \,||\, \langle h\rangle$.

We just showed that if $T^0_{i,j}$, $T^{k-1}_{1,i}$ and $\mathbf{L}^{k-1}_{1,i}$ are computed first (for all i's and j's such that $1\leqslant i\leqslant j$), then we can also derive $T^k_{1,j}$ and $\mathbf{L}^k_{1,j}$. This suggests a dynamic programming algorithm to compute $T^{n-1}_{1,n}$ and $\mathbf{L}^{n-1}_{1,n}$. The algorithm is described in detail (in "Pidgin Algol") in Fig. 1.

Note that the underlying probabilistic failure model does not explicitly appear in the algorithm. It is implicitly used only during the initialization of the algorithm, when the $T^0_{i,j}$'s are computed (for all i's and j's such that $1\leqslant i\leqslant j\leqslant n$). Therefore, the same algorithm can be applied to any underlying failure model such that the $T^0_{i,j}$'s

can be computed.

If we exclude the computation of the $T_{i,j}^0$'s, the time complexity of this algorithm is $O(n^3)$. In fact, the inner loop is executed $O(n^2)$ times, and the most time-consuming operation in this inner loop is the $\mathrm{MIN}_{1<i\leqslant j}$ operation. Since $j = O(n)$, this operation requires $O(n)$-time.

If we assume that the setup costs and the rollback costs are related as described in the next section, then we can reduce the complexity of the algorithm to $O(n^2)$.

We finally note that computing all the $T_{i,j}^0$'s takes $O(n^2)$-time in the two failure models that we consider in § 5. Therefore, with these models, the overall complexity of the algorithms described in Fig. 1 and in the next section is $O(n^3)$ and $O(n^2)$, respectively.

4. An improved algorithm for a restricted model. We assume the following relation between the setup costs and the rollback costs:

For any two checkpoint locations i and j, if $s_i > s_j$ then $r_i \geqslant r_j$.

Note that if there is a non-decreasing function f relating all the rollback costs to the setup costs, i.e., $r_i = f(s_i)$, then the above relation is satisfied. In particular, it is satisfied if for all i we have $r_i = \alpha s_i + \beta$, for some constant $\alpha \geqslant 0$ and $\beta \geqslant 0$. It is also satisfied if all the setup costs or all the rollback costs are equal.

We further assume that the failure model is such that augmenting any segment cannot decrease the probability of a failure occurring in that segment. Formally, for all i, j and k such that $1 \leqslant i \leqslant j \leqslant k < n$, we have $p_{i,j} \geqslant p_{i,k}$, where $p_{i,j}$ denotes the probability that no failures occur during the execution of $[i,j]$. With these assumptions, we can prove the following two theorems.

THEOREM 1. *Suppose the m-optimal solutions for $[i,j]$ are such that their rightmost checkpoint location is k, $i \leqslant k \leqslant j$. Then, for any $p>j$ the m-optimal solutions for $[i,p]$ are such that their rightmost checkpoint location is some h, $h \geqslant k$.*

THEOREM 2. *Suppose the m-optimal solutions for $[i,j]$ are such that their rightmost checkpoint location is k, $i \leqslant k \leqslant j$. Then the $(m+1)$-optimal solutions for $[i,j]$ are such that their rightmost checkpoint location is some h, $h \geqslant k$.*

The proofs of these two theorems can be found in the Appendix. From these two theorems we can immediately derive the following corollary.

COROLLARY. *Let a, b, and c be the rightmost checkpoint locations of the $(k-1)$-optimal solutions for $[1,j]$, the k-optimal solutions for $[1,j]$, and the k-optimal solutions for $[1,j+1]$, respectively $(1<j,\ k<n)$. We have $a \leqslant b \leqslant c$.*

Using this corollary, we can speed up the basic algorithm described in Fig. 1 by a factor of $O(n)$. The main idea is to restrict the range of indexes scanned by the MIN operation in the algorithm's inner loop. Suppose we want to compute $T_{1,j}^k$ and we already know some $\mathbf{L}_{1,j}^{k-1}$ and $\mathbf{L}_{1,j+1}^k$. Let $l_{1,j}^{k-1}$ and $l_{1,j+1}^k$ be the rightmost checkpoint locations of $\mathbf{L}_{1,j}^{k-1}$ and $\mathbf{L}_{1,j+1}^k$, respectively. From the corollary, we know that the rightmost checkpoint location i of any k-optimal solution of $[1,j]$ is such that $l_{1,j}^{k-1} \leqslant i \leqslant l_{1,j+1}^k$. Therefore, we can compute $T_{1,j}^k$ as follows:

$$T \leftarrow \underset{l_{1,j}^{k-1} \leqslant i \leqslant l_{1,j+1}^k}{\mathrm{MIN}} \left(T_{1,i-1}^{k-1} + T_{i,j}^0\right),$$

$$T \leftarrow \mathrm{MIN}(T, T_{1,j}^{k-1}).$$

Note that, in the algorithm described in Fig. 1, when $T_{1,j}^k$ is computed, $\mathbf{L}_{1,j}^{k-1}$ and $\mathbf{L}_{1,j+1}^k$ have already been derived in some earlier steps of the algorithm. Therefore, $l_{1,j}^{k-1}$ and $l_{1,j+1}^k$ are known at that point and they can be used to compute $T_{1,j}^k$ as shown above; $\mathbf{L}_{1,j}^k$ is then derived as in the basic algorithm.

In the basic algorithm of Fig. 1, the range of indexes scanned by the MIN operation to compute $T_{1,j}^k$ was $1<i\leqslant j$. With the algorithm modification that we just described, the range is $l_{1,j}^{k-1}\leqslant i\leqslant l_{1,j+1}^k$. We now show that this modification results in an $O(n^2)$ algorithm[1]. Note first that a $\text{MIN}_{a\leqslant i\leqslant b}$ operation in this algorithm takes $b-a+1$ steps, i.e., $O(b-a)$ time. Let q, $0\leqslant q\leqslant n-1$, be a fixed positive constant. Let $T(q)$ be the following set

$$T(q)=\{T_{1,j}^k \mid 2\leqslant j\leqslant n,\ 1\leqslant k\leqslant n-1, \text{ and } j-k=q\}.$$

Consider the time taken by all the MIN operations performed while computing all the elements of $T(q)$.

When $T_{1,q+1}^1$ is computed, the range of indexes is $1\leqslant i\leqslant l_{1,q+2}^1$.
When $T_{1,q+2}^2$ is computed, the range of indexes is $l_{1,q+2}^1\leqslant i\leqslant l_{1,q+3}^2$.
When $T_{1,q+3}^3$ is computed, the range of indexes is $l_{1,q+3}^2\leqslant i\leqslant l_{1,q+4}^3$.
...
When $T_{1,n-1}^{n-q-1}$ is computed, the range of indexes is $l_{1,n-1}^{n-q-2}\leqslant i\leqslant l_{1,n}^{n-q-1}$.
When $T_{1,n}^{n-q}$ is computed, the range of indexes is $l_{1,n}^{n-q-1}\leqslant i\leqslant n$.

By summing up the number of computation steps needed by all these MIN operations, we get a total of $n+(n-q)$ steps. Therefore, for any fixed q $(0\leqslant q\leqslant n-1)$, the MIN operations performed while computing all the elements of $T(q)$ take a total of $O(n)$ time. By considering all the n possible values of q, we conclude that the MIN operations performed while computing all the $T_{1,j}^k$'s such that $j-k\geqslant 0$ take a total of $O(n^2)$ time. Similarly, we can show that the MIN operations performed while computing all the $T_{1,j}^k$'s such that $j-k<0$ take $O(n^2)$ time. Therefore, during the execution of the algorithm the total time taken by MIN is $O(n^2)$, and the algorithm's complexity is also $O(n^2)$.

5. Models of failure.

5.1. Discrete case. Our first failure model is discrete where failures occur independently. In the absence of failures each task i has a known completion time t_i and a given probability p_i of completing without failures (i.e., in time t_i). Failures occurring during the execution of task i are detected only at the end of task i. For example, failure detection could be done by checking if an integrity assertion about the state holds at the conclusion of the task. In this model we also assume that the t_i and the rollback times r_i are integers.

To use the algorithms of § 3 and § 4 with this failure model, we need to compute all the $T_{i,j}^0$'s. Consider a segment $[i,j]$. Let the random variable $Y_{i,j}$ denote the time required to execute $[i,j]$ in the presence of failures and no checkpoints. Note that $T_{i,j}^0=E[Y_{i,j}]$. Let $q_t=\text{Prob}[Y_{i,j}=t]$. The moment generating function $\Phi(z)=\Sigma_{t=0}^{\infty} q_t z^t$ of this distribution can be derived as follows. The execution of $[i,j]$ takes at least $t_{i,j}=\Sigma_{k=i}^{j} t_k$ units of time. Therefore,

[1] We exclude the time taken to compute all the $T_{i,j}^0$'s during the initialization of the algorithm. This time depends on the underlying failure model. We consider two failure models in § 5, and with both models it takes $O(n^2)$ time to compute all the $T_{i,j}^0$'s.

$$q_0 = q_1 = \cdots = q_{t_{i,j}-1} = 0 .$$

The execution of $[i,j]$ takes exactly $t_{i,j}$ units of time only if no failures occur. Then we have

$$q_{t_{i,j}} = p_i \, p_{i+1} \cdots p_j .$$

The execution of $[i,j]$ takes more than $t_{i,j}$ units of time only if there is at least one failure during the execution. Consider the first failure that occurs after the computation is started. By conditioning on all the possible tasks where this failure could occur, we derive the following recurrence relation for q_t:

$$q_t = (1-p_i)q_{t-(t_i+r_i)} + p_i(1-p_{i+1})q_{t-(t_i+t_{i+1}+r_i)} + \cdots$$

$$+ p_i p_{i+1} \cdots p_{j-1}(1-p_j)q_{t-(t_i+t_{i+1}+\cdots+t_j+r_i)}$$

for all t such that $t > t_{i,j}$. Multiplying both sides of this equation by z^t and summing over all t, $t > t_{i,j}$, we get

$$\Phi(z) - p_i p_{i+1} \cdots p_j \, z^{t_i+t_{i+1}+\cdots+t_j}$$

$$= (1-p_i)\, z^{t_i+r_i}\Phi(z) + p_i(1-p_{i+1})\, z^{t_i+t_{i+1}+r_i}\Phi(z) + \cdots$$

$$+ p_i p_{i+1} \cdots p_{j-1}(1-p_j)\, z^{t_i+\cdots+t_j+r_i}\Phi(z) .$$

Therefore,

$$\Phi(z) = \frac{p_i p_{i+1} \cdots p_j \, z^{t_i+t_{i+1}+\cdots+t_j}}{1-(1-p_i)z^{t_i+r_i} - p_i(1-p_{i+1})z^{t_i+t_{i+1}+r_i} - .. - p_i p_{i+1} .. p_{j-1}(1-p_j)z^{t_i+..+t_j+r_i}} .$$

By taking the derivatives of $\Phi(z)$ and setting z to one, we can find all the moments of the random variable $Y_{i,j}$ and, in particular, we can obtain $T_{i,j}^0 = E[Y_{i,j}]$. It is then easy to verify that the following recurrence relation holds

$$T_{i,i}^0 = \frac{t_i}{p_i} + \left(\frac{1}{p_i} - 1\right) r_i ,$$

$$T_{i,j}^0 = \frac{1}{p_j}(T_{i,j-1}^0 + t_j) + \left(\frac{1}{p_j} - 1\right) r_i \quad \text{for all } j,\ j > i .$$

Using these recurrence relations, we can compute all the $T_{i,j}^0$'s (for i and j, $1 \leqslant i \leqslant j \leqslant n$) in $O(n^2)$-time.

5.2. Continuous case. Rather than assuming the task executions to constitute independent Bernoulli trials as we have done in the previous section, let us consider the case where failures occur according to a stationary renewal process throughout the computation. In other words, the inter-failure times are independent and identically distributed random variables. We assume the existence of a mechanism to detect failures as soon as they occur. As before, when a failure is detected, the computation is rolled back to the most recent checkpoint before it can be resumed. We assume that the checkpoint setups and rollbacks constitute renewal points.

Let the random variable X denote the time until the next failure after a renewal. $F(x) = \text{Prob}[X \leqslant x]$ denotes the distribution function of X which is assumed known. As before, let the random variable $Y_{i,j}$ denote the time required to execute $[i,j]$ in the presence of failures and no checkpoints. Let $V(t) = \text{Prob}[Y_{i,j} \leqslant t]$. We proceed as follows in order to derive an expression for this distribution.

As before, $t_{i,j} = \Sigma_{k=i}^{j} t_k$ be the time required to execute $[i,j]$ without any failures. Clearly, the execution time in the presence of failures cannot be less than this time; that is, $V(t) = 0$ for $t < t_{i,j}$. Conditioning on the time until the first failure after a renewal, we have

$$V(t) = \int_0^\infty \text{Prob}[Y_{i,j} < t \mid X = x] dF(x), \qquad t \geqslant t_{i,j}.$$

If the length of the first failure-free interval is greater than $t_{i,j}$, the computation completes in exactly $t_{i,j}$ time units. Consequently, the above equation can be written as

$$V(t) = \int_0^{t_{i,j}} \text{Prob}[Y_{i,j} < t \mid X = x] dF(x) + (1 - F(t_{i,j})), \quad t \geqslant t_{i,j}.$$

If, however, $X = x < t_{i,j}$, the computation must be rolled back at least once. Since resuming the computation after a rollback constitutes a renewal point, the probability that the total execution time for $[i,j]$ will be at most t time units after having expended x time units due to the failure and r_i time units for the rollback is simply given by $V(t - x - r_i)$. This observation allows us to write $V(t)$ as a *renewal equation* [9]

$$V(t) = \int_0^{t_{i,j}} V(t - x - r_i) dF(x) + (1 - F(t_{i,j})).$$

Making the change of variable $y = x + r_i$ and the substitutions

$$z(x) = \begin{vmatrix} 0, & x < t_{i,j}, \\ 1 - F(t_{i,j}), & x \geqslant t_{i,j} \end{vmatrix}$$

and

$$G(x) = \begin{vmatrix} 0, & x < r_i, \\ F(x - r_i), & r_i \leqslant x \leqslant t_{i,j} + r_i, \\ F(t_{i,j}), & x > t_{i,j} + r_i, \end{vmatrix}$$

we obtain

$$V(t) = \int_0^t V(t - y) dG(y) + z(t), \qquad \text{for all } t.$$

Note that the integral represents the convolution of the functions V and g where $g(x) = dG(x)/dx$. Taking Laplace transforms yields

$$\tilde{V}(s) = \tilde{V}(s)\tilde{g}(s) + \tilde{z}(s)$$

where $\tilde{f}(s) = \int_0^\infty e^{-sx} f(x) dx$ denotes the Laplace transform of the function $f(x)$. Finally, the transform of the distribution we are interested is given by

$$\tilde{V}(s) = \frac{\tilde{z}(s)}{1-\tilde{g}(s)}$$

where $\tilde{z}(s) = e^{-s\,t_{i,j}}(1-F(t_{i,j}))/s$ and $\tilde{g}(s) = e^{-s\,r_i}\int_0^{t_{i,j}} e^{-s\,x}\,dF(x)$.

Whether the inverse transform of $\tilde{V}(s)$ has a closed form solution depends on the nature of $F(x)$. Note that the probability *density* function of $Y_{i,j}$, $v(t)=dV(t)/dt$, has the transform

$$\tilde{v}(s) = s\tilde{V}(s) = \frac{e^{-s\,t_{i,j}}(1-F(t_{i,j}))}{1-\tilde{g}(s)}.$$

Since the algorithms given in the previous sections need only the values for $T_{i,j}^0$ (by definition, this is the first moment of $Y_{i,j}$) for all $1 \leqslant i \leqslant n$ and $i \leqslant j \leqslant n$, we do not have to obtain the inverse transform of $\tilde{v}(s)$. All of the moments of $Y_{i,j}$ can be obtained by differentiating the transform of its density. In particular, the first moment is given by

$$(5.2.1) \qquad T_{i,j}^0 = E[Y_{i,j}] = -\frac{d}{ds}\tilde{v}(s)\Big|_{s=0} = t_{i,j} - \frac{\frac{d}{ds}\tilde{g}(s)\Big|_{s=0}}{1-\tilde{g}(0)}$$

$$= t_{i,j} + \frac{r_i F(t_{i,j}) + \int_0^{t_{i,j}} t\,dF(t)}{1-F(t_{i,j})}.$$

As an example, we will derive an expression for $E[Y_{i,j}]$ in the presence of Poisson failures (i.e., $F(x)=1-e^{-\lambda x}$ and $dF(x)=\lambda e^{-\lambda x}dx$ where λ is the failure rate). It is important to note that the above derivation of $E[Y_{i,j}]$ does *not* rely on the inter-failure times being exponentially distributed. We have selected the example simply to define one possible form of $F(x)$. Substituting the expressions for $F(x)$ and $dF(x)$ into equation (5.2.1) and simplifying, we obtain

$$T_{i,j}^0 = E[Y_{i,j}] = \frac{(e^{\lambda t_{i,j}}-1)(\lambda r_i+1)}{\lambda}.$$

6. Extensions. Up to this point, we have assumed that the checkpoint setups and rollbacks are failure-free. We now discuss how each one of these assumptions can be easily relaxed.

Let the checkpoint setups be subject to the same failure model as the normal tasks. Consider $[i,j]$ and our computation of $T_{i,j}^0$ for this segment. Recall that in the algorithm, $T_{i,j}^0$ denotes the expected execution time of $[i,j]$ given that a checkpoint is established at the end of task j. We consider this checkpoint setup to be a new task of length s_{j+1} which augments segment $[i,j]$. (Obviously, if $j=n$, the completion of the computation, rather than a checkpoint, marks the end of the segment. For notational convenience, we define $s_{n+1} = 0$.) The execution time of the augmented segment without any failures becomes $t_{i,j} = \Sigma_{k=i}^{j} t_k + s_{j+1}$. Following this change, the expected execution time $T_{i,j}^0$ of the augmented segment can be computed as shown in § 5. Clearly, since the checkpoint setup costs are now included in the $T_{i,j}^0$'s, the minimization step in the algorithm of Fig. 1 becomes

$$T \leftarrow \underset{1<i\leqslant j}{\text{MIN}} \, (T^{k-1}_{1,i-1} + T^0_{i,j}) \, .$$

Suppose now that checkpoint setups are failure-free but the rollbacks are subject to failures. We can apply the algorithm of Fig. 1 provided that the $T^0_{i,j}$'s are computed appropriately. For example, in our discrete model of failure, let ρ_i be the probability that a rollback to location i is failure-free. Proceeding in a similar fashion to § 5.1, we can derive the following recurrence relation for the new $T^0_{i,j}$'s:

$$T^0_{i,i} = \frac{t_i}{p_i} + \left(\frac{1}{p_i} - 1\right)\frac{r_i}{\rho_i} \, ,$$

$$T^0_{i,j} = \frac{1}{p_j}(T^0_{i,j-1} + t_j) + \left(\frac{1}{p_j} - 1\right)\frac{r_i}{\rho_i} \quad \text{for all } j, \, j>i.$$

This extension can be similarly incorporated into the continuous failure model analysis.

Finally, since the necessary modifications for the above extensions to the problem are orthogonal, both of them could be present in the basic algorithm.

7. Discussion and conclusions. We have presented an algorithm to select a set of checkpoint locations out of the $n-1$ candidate locations such that the resulting expected execution time for the computation is minimized. The algorithm can be applied to any failure model such that the expected execution time can be computed for any segment with no checkpoints. The time complexity of this algorithm was shown to be $O(n^3)$ where n is the number of tasks making up the computation. We also described an $O(n^2)$ algorithm for the case that, for all i and j, $s_i > s_j$ implies $r_i \geqslant r_j$. In most applications, this is a reasonable assumption since rolling back the state involves loading the same state information which was saved at the checkpoint.

The objective function we have selected for the optimization problem is the expected execution time. For certain time-critical applications, we may be interested in knowing the probability with which the computation will complete in less than some given time. Note that, given the optimum checkpoint locations, the computation can be viewed as a sequence of *segments* rather than tasks where each segment is delimited by a checkpoint. The execution times for these segments are independent and we have derived their moment generating functions (in the two failure models that we considered). The moment generating function for the total execution time of the computation is simply the product of the individual segment moment generating functions. In principle, this function can be inverted to obtain the distribution of the total execution time. Given this distribution, confidence bounds for the total execution time can be derived. If inverting the moment generating function is not possible, we can use it to derive the mean, the variance, and any higher moments of the total execution time (note that the mean is already part of the algorithm's output). Given these values, we can derive the probability with which the computation completes within a given interval about the known mean by an application of Chebyshev's Inequality.

If the static structure of a computation is not simple, its dynamic characterization as a sequence of tasks may be difficult to obtain. Given this observation, a generalization of our work would represent the static computation as a Directed Acyclic Graph (DAG) where the vertices are tasks and an edge (i,j) denotes a possible execution where task j follows task i. Here we may want to position the checkpoints on this DAG in a way that minimizes the maximum expected execution time over all the

possible execution paths. If we augment the DAG by associating probabilities with each edge such that the two tasks connected by the edge appear in succession with that probability, we can pose the problem of selecting checkpoint locations such that the total expected execution time for the computation is minimized.

Appendix. We introduce some definitions to be used in the following proofs leading to those of Theorems 1 and 2 of § 4. Consider segment $[i,j]$ for some $i \leqslant j$. Let $p_{i,j}$ be the probability that no failures occur during the execution of $[i,j]$, $\alpha_{i,j}$ be $1/p_{i,j}$ and $t_{i,j}$ be the execution time of $[i,j]$ when no failures occur. Given that a failure occurs during the execution of $[i,j]$, $l_{i,j}$ denotes the expected time interval from the beginning of task i to the time the failure is detected.

LEMMA 1. *For all i and j such that $i \leqslant j$ we have*

$$T_{i,j}^0 = t_{i,j} + (\alpha_{i,j} - 1)(l_{i,j} + r_i).$$

For all i,j and k such that $i < k \leqslant j$ we have

$$T_{i,j}^0 = \alpha_{k,j} T_{i,k-1}^0 + t_{k,j} + (\alpha_{k,j} - 1)(l_{k,j} + r_i).$$

Proof. We present the proofs for the case where the failure times are discrete random variables. The continuous case proofs follow simply by replacing the sums with integrals and discrete probabilities with density functions.

The first equality can be proven as follows. Let the random variable X denote the time until the first failure is detected after the beginning of $[i,j]$. Conditioning on this time, we have

$$T_{i,j}^0 = t_{i,j} \cdot \text{Prob}[X > t_{i,j}] + \sum_{x \leqslant t_{i,j}} (x + r_i + T_{i,j}^0) \cdot \text{Prob}[X = x].$$

By definition, $p_{i,j} = \text{Prob}[X > t_{i,j}]$. Consequently,

$$T_{i,j}^0 = t_{i,j} p_{i,j} + (r_i + T_{i,j}^0)(1 - p_{i,j}) + \sum_{x \leqslant t_{i,j}} x \,\text{Prob}[X = x].$$

Noting that, by definition, $l_{i,j} = \Sigma_{x \leqslant t_{i,j}} x \,\text{Prob}[X = x]/(1 - p_{i,j})$, and solving for $T_{i,j}^0$ we have $p_{i,j} T_{i,j}^0 = p_{i,j} t_{i,j} + (1 - p_{i,j})(l_{i,j} + r_i)$. Dividing both sides by $p_{i,j}$ results in the desired expression.

To prove the second equality, we condition on the time until the first failure is detected after the beginning of $[k,j]$. Letting X denote this time, we have

$$T_{i,j}^0 = T_{i,k-1}^0 + t_{k,j} \cdot \text{Prob}[X > t_{k,j}] + \sum_{x \leqslant t_{k,j}} (x + r_i + T_{i,j}^0) \cdot \text{Prob}[X = x].$$

As before, rewriting this equation in terms of $l_{k,j}$, and solving for $T_{i,j}^0$ we obtain

$$p_{k,j} T_{i,j}^0 = T_{i,k-1}^0 + p_{k,j} t_{k,j} + (1 - p_{k,j})(l_{k,j} + r_i).$$

Dividing the equation by $p_{k,j}$ completes the proof. □

LEMMA 2. *For all i,j and k such that $i < k \leqslant j$ and $\alpha_{k,j} \neq 1$ we have*

$$T_{i,k-1}^0 + T_{k,j}^0 + s_k \leqslant T_{i,j}^0$$

$$\textit{if and only if } s_k \leqslant (\alpha_{k,j} - 1)(r_i + T_{i,k-1}^0 - r_k).$$

Furthermore, the equality holds in the left expression if and only if it holds in the right one.

Proof. From Lemma 1 we have

$$T^0_{i,j} = \alpha_{k,j}\, T^0_{i,k-1} + t_{k,j} + (\alpha_{k,j} - 1)(l_{k,j} + r_i)$$

and

$$T^0_{k,j} = t_{k,j} + (a_{k,j} - 1)\,(l_{k,j} + r_k)\,.$$

Therefore $T^0_{i,k-1} + T^0_{k,j} + s_k \leqslant T^0_{i,j}$ if and only if

$$T^0_{i,k-1} + t_{k,j} + (\alpha_{k,j} - 1)(l_{k,j} + r_k) + s_k \leqslant \alpha_{k,j}\, T^0_{i,k-1} + t_{k,j} + (\alpha_{k,j} - 1)(l_{k,j} + r_i).$$

This inequality holds if and only if

$$(\alpha_{k,j} - 1)(l_{k,j} + r_k) + s_k \leqslant (\alpha_{k,j} - 1)(l_{k,j} + r_i + T^0_{i,k-1})\,.$$

Since $\alpha_{k,j} > 1$, this is satisfied if and only if

$$l_{k,j} + r_k + s_k\,(\alpha_{k,j} - 1)^{-1} \leqslant l_{k,j} + r_i + T^0_{i,k-1}\,.$$

This last inequality holds if and only if $s_k \leqslant (\alpha_{k,j} - 1)(r_i + T^0_{i,k-1} - r_k)$. It is also easy to check that $T^0_{i,k-1} + T^0_{k,j} + s_k = T^0_{i,j}$ if and only if $s_k = (\alpha_{k,j} - 1)(r_i + T^0_{i,k-1} - r_k)$. □

LEMMA 3. *Suppose the m-optimal solutions for $[i,j]$ are such that their rightmost checkpoint location is k, $i < k \leqslant j$. Then we have $\alpha_{k,j} > 1$ and $s_k < (\alpha_{k,j} - 1)(r_i + T^0_{i,k-1} - r_k)$.*

Proof. The proof is by induction on m (note that by hypothesis $m \geqslant 1$).

Suppose $m = 1$. By hypothesis we have

$$T^1_{i,j} = T^0_{i,k-1} + T^0_{k,j} + s_k < T^0_{i,j}\,. \qquad \text{(L3.1)}$$

If we show that $\alpha_{k,j} \neq 1$ then Lemma 3 follows directly from Lemma 2. Suppose $\alpha_{k,j} = 1$. From Lemma 1 we have $T^0_{i,j} = T^0_{i,k-1} + t_{k,j}$ and $T^0_{k,j} = t_{k,j}$. Therefore, $T^0_{i,j} = T^0_{i,k-1} + T^0_{k,j}$ and this contradicts (L3.1). So $\alpha_{k,j} > 1$ and Lemma 3 follows.

Now assume Lemma 3 holds for all m, $1 \leqslant m < r$, for some $r > 1$. We show that it must also hold for $m = r$. Suppose the r-optimal solutions for $[i,j]$ are such that their rightmost checkpoint location is k, $i < k \leqslant j$. Let $\mathbf{L}^{r-1}_{i,k-1}$ be a $(r-1)$-optimal solution for $[i,k-1]$. We consider $\mathbf{L}^r_{i,j} = \mathbf{L}^{r-1}_{i,k-1} \,\|\, \langle k \rangle$. Note that this must be a r-optimal solution for $[i,j]$. Let h be the rightmost checkpoint location of $\mathbf{L}^{r-1}_{i,k-1}$. We have $i \leqslant h < k \leqslant j$.

Assume first that $i < h$. By induction hypothesis we have $s_h < (\alpha_{h,k-1} - 1)(r_i + T^0_{i,h-1} - r_h)$, and $\alpha_{h,k-1} > 1$. Note that $s_h \geqslant 0$, and therefore

$$r_h < r_i + T^0_{i,h-1}\,. \qquad \text{(L3.2)}$$

Since $\mathbf{L}^1_{h,j} = \langle k \rangle$ is a 1-optimal solution for $[h,j]$ then, by induction hypothesis, we have $\alpha_{k,j} > 1$ and $s_k < (\alpha_{k,j} - 1)(r_h + T^0_{h,k-1} - r_k)$. Suppose, for contradiction, that $s_k \geqslant (\alpha_{k,j} - 1)(r_i + T^0_{i,k-1} - r_k)$. Combining the last two inequalities we have $r_i + T^0_{i,k-1} < r_h + T^0_{h,k-1}$. From Lemma 1 we have

$$r_i + [\alpha_{h,k-1}\, T^0_{i,h-1} + t_{h,k-1} + (\alpha_{h,k-1} - 1)(l_{h,k-1} + r_i)]$$
$$< r_h + [t_{h,k-1} + (\alpha_{h,k-1} - 1)(l_{h,k-1} + r_h)]\,.$$

So $(r_i + T^0_{i,h-1})\alpha_{h,k-1} < r_h\, \alpha_{h,k-1}$, and $r_i + T^0_{i,h-1} < r_h$ contradicting (L3.2). Therefore we must have $s_k < (\alpha_{k,j} - 1)(r_i + T^0_{i,k-1} - r_k)$.

Suppose now that $i = h$. In this case it is clear that $\mathbf{L}^{r-1}_{i,k-1} = <>$ and $\mathbf{L}^r_{i,j} = <k>$, therefore $\mathbf{L}^1_{i,j} = <k>$, i.e., $<k>$ is the 1-optimal checkpoint selection for $[i,j]$. Then, by induction hypothesis, we have $\alpha_{k,j} > 1$ and $s_k < (\alpha_{k,j} - 1)(r_i + T^0_{i,k-1} - r_k)$. □

LEMMA 4. *Suppose the m-optimal solutions for $[i,j]$ are such that the rightmost checkpoint location is k, $i < k \leq j$. Then for all h, $i < h < k$, we have*

$$s_k - s_h \leq (\alpha_{k,j} - 1)(r_h + T^0_{h,k-1} - r_k).$$

Proof: We prove the lemma by induction on m (note that by hypothesis $m \geq 1$).

Assume first that $m = 1$. Suppose, for contradiction, that for some h, $i < h < k$, we have

$$\text{(L4.1)} \qquad s_k - s_h > (\alpha_{k,j} - 1)(r_h + T^0_{h,k-1} - r_k).$$

From Lemma 3 we have $s_k < (\alpha_{k,j} - 1)(r_i + T^0_{i,k-1} - r_k)$, and $\alpha_{k,j} > 1$. Therefore, $r_h + T^0_{h,k-1} < r_i + T^0_{i,k-1}$, and from Lemma 1 we have

$$r_h + [t_{h,k-1} + (\alpha_{h,k-1} - 1)(l_{h,k-1} + r_h)]$$

$$< r_i + [\alpha_{h,k-1} T^0_{i,h-1} + t_{h,k-1} + (\alpha_{h,k-1} - 1)(l_{h,k-1} + r_i)]$$

so, $\alpha_{h,k-1} r_h < \alpha_{h,k-1}(r_i + T^0_{i,h-1})$, and

$$\text{(L4.2)} \qquad r_h < r_i + T^0_{i,h-1}.$$

Since $<k>$ is the 1-optimal solution for $[i,j]$ we have

$$T^1_{i,j} = T^0_{i,k-1} + T^0_{k,j} + s_k \leq T^0_{i,h-1} + T^0_{h,j} + s_h$$

and therefore $T^0_{i,k-1} - T^0_{i,h-1} + s_k \leq T^0_{h,j} - T^0_{k,j} + s_h$. We define $\Delta_1 = T^0_{i,k-1} - T^0_{i,h-1}$ and $\Delta_2 = T^0_{h,j} - T^0_{k,j}$, and we write the last inequality as

$$\text{(L4.3)} \qquad \Delta_1 + s_k \leq \Delta_2 + s_h.$$

Applying Lemma 1 we have

$$\Delta_1 = [\alpha_{h,k-1} T^0_{i,h-1} + t_{h,k-1} + (\alpha_{h,k-1} - 1)(l_{h,k-1} + r_i)] - T^0_{i,h-1}$$

so

$$\Delta_1 = t_{h,k-1} + (\alpha_{h,k-1} - 1)(l_{h,k-1} + r_i + T^0_{i,h-1}).$$

We also have

$$\Delta_2 = [\alpha_{k,j} T^0_{h,k-1} + t_{k,j} + (\alpha_{k,j} - 1)(l_{k,j} + r_h)] - [t_{k,j} + (\alpha_{k,j} - 1)(l_{k,j} + r_k)]$$

and therefore

$$\Delta_2 = \alpha_{k,j} T^0_{h,k-1} + (\alpha_{k,j} - 1)(r_h - r_k).$$

From (L4.1) we have $r_h - r_k < -T^0_{h,k-1} + (s_k - s_h)(\alpha_{k,j} - 1)^{-1}$. Therefore, $\Delta_2 < \alpha_{k,j} T^0_{h,k-1} - (\alpha_{k,j} - 1) T^0_{h,k-1} + s_k - s_h$, that is

(L4.4) $$\Delta_2 < T^0_{h,k-1} + s_k - s_h .$$

From (L4.3) and (L4.4) we obtain

(L4.5) $$\Delta_1 < T^0_{h,k-1} ,$$

that is, $t_{h,k-1} + (\alpha_{h,k-1} - 1)(l_{h,k-1} + r_i + T^0_{i,h-1}) < T^0_{h,k-1}$. By Lemma 1 we have

$$t_{h,k-1} + (\alpha_{h,k-1} - 1)(l_{h,k-1} + r_i + T^0_{i,h-1}) < t_{h,k-1} + (\alpha_{h,k-1} - 1)(l_{h,k-1} + r_h)$$

so $(\alpha_{h,k-1} - 1)(r_i + T^0_{i,h-1}) < (\alpha_{h,k-1} - 1) r_h$. Suppose $\alpha_{h,k-1} = 1$. Then $\Delta_1 = t_{h,k-1}$ and $T^0_{h,k-1} = t_{h,k-1}$, a contradiction to (L4.5). So we must have $\alpha_{h,k-1} > 1$ and $r_i + T^0_{i,h-1} < r_h$. But this contradicts (L4.2) and therefore $s_k - s_h \leqslant (\alpha_{k,j} - 1)(r_h + T^0_{h,k-1} - r_k)$ for all h, $i < h < k$.

Now assume the lemma holds for all m, $1 \leqslant m < r$, for some $r > 1$. We show that it must also hold for $m = r$. Suppose the r-optimal solutions for $[i,j]$ are such that their rightmost checkpoint location is k, $i < k \leqslant j$. Let $\mathbf{L}^{r-1}_{i,k-1}$ be a $(r-1)$-optimal solution for $[i,k-1]$. We consider $\mathbf{L}^r_{i,j} = \mathbf{L}^{r-1}_{i,k-1} \,||\, \langle k \rangle$. Note that this must be a r-optimal solution for $[i,j]$. Let p be the location of the rightmost checkpoint of $\mathbf{L}^{r-1}_{i,k-1}$. Note that $i \leqslant p < k \leqslant j$.

Assume first that $i < p$. Consider some h such that $i < h < k$. There are two possible cases, either $p < h$ or $h \leqslant p$. We show that in both cases $s_k - s_h \leqslant (\alpha_{k,j} - 1)(r_h + T^0_{h,k-1} - r_k)$.

(*i*) Suppose $i < p < h < k \leqslant j$. Since k must be the (rightmost) checkpoint location of the 1-optimal solution for $[p,j]$ then, by induction hypothesis, we have $s_k - s_h \leqslant (\alpha_{k,j} - 1)(r_h + T^0_{h,k-1} - r_k)$.

(*ii*) We now consider h such that $i < h \leqslant p < k \leqslant j$. Suppose, for contradiction, that $s_k - s_h > (\alpha_{k,j} - 1)(r_h + T^0_{h,k-1} - r_k)$. Since $\mathbf{L}^1_{p,j} = \langle k \rangle$ is a 1-optimal solution for $[p,j]$ then, by Lemma 3, we have $s_k < (\alpha_{k,j} - 1)(r_p + T^0_{p,k-1} - r_k)$, and $\alpha_{k,j} > 1$. Combining the last two inequalities we have $r_h + T^0_{h,k-1} < r_p + T^0_{p,k-1}$. Therefore $h \neq p$ (i.e., $h < p$), and from Lemma 1 we have

$$r_h + [\alpha_{p,k-1} T^0_{h,p-1} + t_{p,k-1} + (\alpha_{p,k-1} - 1)(l_{p,k-1} + r_h)]$$
$$< r_p + [t_{p,k-1} + (\alpha_{p,k-1} - 1)(l_{p,k-1} + r_p)] .$$

By simplifying we obtain

(L4.6) $$r_h + T^0_{h,p-1} < r_p .$$

Since the rightmost checkpoint location of the $(r-1)$-optimal solutions for $[i,k-1]$ is p, and $i < h < p \leqslant k-1$, then, by induction hypothesis, we have

(L4.7) $$s_p - s_h \leqslant (\alpha_{p,k-1} - 1)(r_h + T^0_{h,p-1} - r_p) .$$

From Lemma 3 we also have $\alpha_{p,k-1} > 1$. From (L4.6) and (L4.7) we obtain $s_p - s_h < 0$. From our assumption about checkpointing costs either $s_p = s_h$, a contradiction, or $r_p \leqslant r_h$ which contradicts (L4.6). So we must have $s_k - s_h \leqslant (\alpha_{k,j} - 1)(r_h + T^0_{h,k-1} - r_k)$.

Suppose now that $i = p$. In this case it is clear that $\mathbf{L}_{i,k-1}^{r-1} = \langle\rangle$ and $\mathbf{L}_{i,j}^{r} = \langle k \rangle$, and therefore $\mathbf{L}_{i,j}^{1} = \langle k \rangle$, i.e., $\langle k \rangle$ is the 1-optimal checkpoint selection for $[i,j]$. Then, by induction hypothesis, we have $s_k - s_h \leqslant (\alpha_{k,j} - 1)(r_h + T_{h,k-1}^0 - r_k)$, for all $h, i \leqslant h < k$. □

LEMMA 5. *Suppose the m-optimal solutions for $[i,j]$ are such that their rightmost checkpoint location is k, $i \leqslant k \leqslant j$. Consider h and p such that $i \leqslant h \leqslant k \leqslant j \leqslant p$. If $i = h$ or $s_h \leqslant s_k$ then we have $T_{k,p}^0 - T_{k,j}^0 \leqslant T_{h,p}^0 - T_{h,j}^0$.*

Proof. The result is obvious for $i = k$, or $h = k$, or $j = p$. Otherwise, suppose that for contradiction

$$T_{k,p}^0 - T_{k,j}^0 > T_{h,p}^0 - T_{h,j}^0 \tag{L5.1}$$

for some $i \leqslant h < k \leqslant j < p$ such that $i = h$ or $s_h \leqslant s_k$. From Lemma 1 we have

$$T_{k,p}^0 = \alpha_{j+1,p}\, T_{k,j}^0 + t_{j+1,p} + (\alpha_{j+1,p} - 1)(l_{j+1,p} + r_k)$$

and therefore

$$T_{k,p}^0 - T_{k,j}^0 = t_{j+1,p} + (\alpha_{j+1,p} - 1)(T_{k,j}^0 + l_{j+1,p} + r_k)\,. \tag{L5.2}$$

Similarly we have

$$T_{h,p}^0 - T_{h,j}^0 = t_{j+1,p} + (\alpha_{j+1,p} - 1)(T_{h,j}^0 + l_{j+1,p} + r_h)\,. \tag{L5.3}$$

From (L5.1), (L5.2) and (L5.3) we have

$$(\alpha_{j+1,p} - 1)(T_{k,j}^0 + l_{j+1,p} + r_k) > (\alpha_{j+1,p} - 1)(T_{h,j}^0 + l_{j+1,p} + r_h)\,.$$

Suppose $\alpha_{j+1,p} = 1$. In this case, from (L5.2) and (L5.3) we have $T_{k,p}^0 - T_{k,j}^0 = T_{h,p}^0 - T_{h,j}^0 = t_{j+1,p}$ which contradicts (L5.1). Therefore $\alpha_{j+1,p} > 1$, and we have

$$r_k + T_{k,j}^0 > r_h + T_{h,j}^0\,. \tag{L5.4}$$

Note that $h < k \leqslant j$ and by applying Lemma 1 we derive

$$r_k + [t_{k,j} + (\alpha_{k,j} - 1)(l_{k,j} + r_k)] > r_h + [\alpha_{k,j}\, T_{h,k-1}^0 + t_{k,j} + (\alpha_{k,j} - 1)(l_{k,j} + r_h)].$$

Then we have $\alpha_{k,j}\, r_k > \alpha_{k,j}\,(r_h + T_{h,k-1}^0)$ and

$$r_k > r_h + T_{h,k-1}^0\,. \tag{L5.5}$$

From Lemma 3, we have $s_k < (\alpha_{k,j} - 1)(r_i + T_{i,k-1}^0 - r_k)$, and $\alpha_{k,j} > 1$, therefore

$$r_k < r_i + T_{i,k-1}^0\,. \tag{L5.6}$$

If $i = h$ then (L5.6) contradicts (L5.5). Suppose $i < h$. From Lemma 4, we have $s_k - s_h \leqslant (\alpha_{k,j} - 1)(r_h + T_{h,k-1}^0 - r_k)$. Therefore, if $s_h \leqslant s_k$, then $r_k \leqslant r_h + T_{h,k-1}^0$ which also contradicts (L5.5). Then neither $i = h$ nor $s_h \leqslant s_k$ and the proof is complete. □

LEMMA 6. *If $i \leqslant h \leqslant k \leqslant j$ and $r_k \leqslant r_h$ then $T_{k,j}^0 \leqslant T_{h,j}^0$.*

Proof. The lemma is obvious for $h = k$. We now assume $i \leqslant h < k \leqslant j$. From Lemma 1 we have

$$T_{h,j}^0 = \alpha_{k,j}\, T_{h,k-1}^0 + t_{k,j} + (\alpha_{k,j} - 1)(l_{k,j} + r_h)$$

and $T_{k,j}^0 = t_{k,j} + (\alpha_{k,j} - 1)(l_{k,j} + r_k)$. Therefore we have

$$T_{h,j}^0 - T_{k,j}^0 = \alpha_{k,j}\, T_{h,k-1}^0 + (\alpha_{k,j} - 1)(r_h - r_k)$$

and since $r_k \leqslant r_h$ then $T_{h,j}^0 - T_{k,j}^0 \geqslant 0$. □

LEMMA 7. *If* $i \leqslant h \leqslant k \leqslant j \leqslant p$ *and* $r_k \leqslant r_h$ *then*

$$T_{h,j}^0 - T_{k,j}^0 \leqslant T_{h,p}^0 - T_{k,p}^0 .$$

Proof. The lemma is obvious if $h = k$ or $j = p$. We now assume that $i \leqslant h < k \leqslant j < p$. In the proof of Lemma 6 we showed that

$$T_{h,j}^0 - T_{k,j}^0 = \alpha_{k,j}\, T_{h,k-1}^0 + (\alpha_{k,j} - 1)(r_h - r_k)$$

and similarly we have

$$T_{h,p}^0 - T_{k,p}^0 = \alpha_{k,p}\, T_{h,k-1}^0 + (\alpha_{k,p} - 1)(r_h - r_k) .$$

Since $p > j$ it is clear that $\alpha_{k,p} \geqslant \alpha_{k,j}$. By hypothesis we also have $r_h - r_k \geqslant 0$ and therefore $T_{h,j}^0 - T_{k,j}^0 \leqslant T_{h,p}^0 - T_{k,p}^0$. □

THEOREM 1. *Suppose the* m*-optimal solutions for* $[i,j]$ *are such that their rightmost checkpoint location is* k*,* $i \leqslant k \leqslant j$ *. Then for any* $p > j$ *the* m*-optimal solutions for* $[i,p]$ *are such that their rightmost checkpoint location is some* h*,* $h \geqslant k$ *.*

Proof. If $i = k$, the theorem is obvious. Assume $i < k$ and therefore $m \geqslant 1$. Suppose, for contradiction, that the rightmost checkpoint location of the m-optimal solutions for $[i,p]$ is h, $i \leqslant h < k$.

Assume first that $i < h$. By hypothesis we must have

$$T_{i,k-1}^{m-1} + T_{k,j}^0 + s_k \leqslant T_{i,h-1}^{m-1} + T_{h,j}^0 + s_h . \tag{T1.1}$$

From our definition of h we also have

$$T_{i,k-1}^{m-1} + T_{k,p}^0 + s_k \geqslant T_{i,h-1}^{m-1} + T_{h,p}^0 + s_h . \tag{T1.2}$$

Suppose equality holds in (T1.1). Then the $(m-1)$-optimal solutions for $[i,h-1]$ cannot include fewer checkpoints than the $(m-1)$-optimal solutions for $[i,k-1]$. Therefore equality cannot also hold in (T1.2), otherwise k would be the rightmost checkpoint location of the m-optimal solutions for $[i,p]$ contradicting our definition of h. So equality cannot hold simultaneously in (T1.1) and in (T1.2). Then, subtracting (T1.1) from (T1.2) we obtain

$$T_{k,p}^0 - T_{k,j}^0 > T_{h,p}^0 - T_{h,j}^0 . \tag{T1.3}$$

If $s_h \leqslant s_k$ then (T1.3) contradicts Lemma 5; therefore $s_k < s_h$. From our assumption about checkpointing costs we have $r_k \leqslant r_h$. Then, from Lemma 6, we also have $T_{k,j}^0 \leqslant T_{h,j}^0$, so

$$r_k + T_{k,j}^0 \leqslant r_h + T_{h,j}^0 . \tag{T1.4}$$

However, in the proof of Lemma 5 we showed that (T1.3) implies $r_k + T_{k,j}^0 > r_h + T_{h,j}^0$ which contradicts (T1.4).

Suppose now that $i = h$, i.e., the m-optimal solutions for $[i,p]$ contain no checkpoints ($\mathbf{L}_{i,p}^m = \langle\, \rangle$). In this case we must have

(T1.5) $$T^{m-1}_{i,k-1} + T^0_{k,j} + s_k < T^0_{i,j} .$$

Since $\mathbf{L}^m_{i,p} = <>$ then

(T1.6) $$T^{m-1}_{i,k-1} + T^0_{k,p} + s_k \geqslant T^0_{i,p} .$$

Subtracting (T1.5) from (T1.6) we obtain (T1.3). From (T1.3) we can also derive a contradiction exactly as before, and the proof is complete. □

THEOREM 2. *Suppose the m-optimal solutions for $[i,j]$ are such that their rightmost checkpoint location is k, $i \leqslant k \leqslant j$. Then the $(m+1)$-optimal solutions for $[i,j]$ are such that their rightmost checkpoint location is some h, $h \geqslant k$.*

Proof. If $i = k$, the theorem is obvious. Assume $i < k$ and therefore $m \geqslant 1$. If $T^{m+1}_{i,j} = T^m_{i,j}$ then m-optimal solutions for $[i,j]$ are also $(m+1)$-optimal solutions for $[i,j]$ and the theorem clearly holds. We now assume $T^{m+1}_{i,j} < T^m_{i,j}$. Note that in this case $(m+1)$-optimal solutions for $[i,j]$ have exactly $m+1$ checkpoints. Let g be the number of checkpoints in the m-optimal solutions for $[i,j]$. Note that $1 \leqslant g \leqslant m$. We form a particular m-optimal solution $\mathbf{L}^m_{i,j} = <u_g, u_{g-1}, \cdots, u_1>$ for $[i,j]$ as follows. Let u_1 be the rightmost checkpoint location of the m-optimal solutions for $[i,j]$, let u_2 be the rightmost checkpoint location of the $(m-1)$-optimal solutions for $[i,u_1 - 1]$, $\cdots$, and let u_g be the rightmost checkpoint location of the $[(m+1)-g]$-optimal solutions for $[i,u_{g-1} - 1]$. Note that with this construction $<u_g, u_{g-1}, \cdots, u_f>$ is a $[(m+1)-f]$-optimal solution for $[i,u_{f-1} - 1]$, for all f, $2 \leqslant f \leqslant g$. By hypothesis we have $u_1 = k$. We also form in a similar way a particular $(m+1)$-solution $\mathbf{L}^{m+1}_{i,j} = <v_{m+1}, v_m, \cdots, v_1>$ for $[i,j]$. We define $v_1 = h$. Suppose, for contradiction, that $h < k$ (i.e., $v_1 < u_1$). There are two possible cases. We show that each one leads to a contradiction.

I. Suppose first that $v_m \leqslant u_g$. Note that $v_{m+1} < v_m$ and therefore $v_{m+1} < u_g$. Since $\mathbf{L}^m_{i,j} = <u_g, u_{g-1}, \cdots, u_1>$ is a m-optimal solution for $[i,j]$ we have

$$T^m_{i,j} = T^0_{i,u_g - 1} + \sum_{l=g-1}^{1} T^0_{u_{l+1},u_l - 1} + T^0_{u_1,j} + \sum_{l=g}^{1} s_{u_l}$$

and

$$T^m_{i,j} \leqslant T^0_{i,v_m - 1} + \sum_{l=m-1}^{1} T^0_{v_{l+1},v_l - 1} + T^0_{v_1,j} + \sum_{l=m}^{1} s_{v_l}$$

where the indexes of the summations scan the segments from left to right, and the summation $\Sigma^b_{l=a} \ldots$ is defined to be zero if $a < b$

Therefore,

(T2.1) $$T^0_{i,u_g - 1} - T^0_{i,v_m - 1} + F \leqslant 0$$

where

$$F = \sum_{l=g-1}^{1} T^0_{u_{l+1},u_l - 1} - \sum_{l=m-1}^{1} T^0_{v_{l+1},v_l - 1} + T^0_{u_1,j} - T^0_{v_1,j} + \sum_{l=g}^{1} s_{u_l} - \sum_{l=m}^{1} s_{v_l} .$$

Since $\mathbf{L}^{m+1}_{i,j} = <v_{m+1}, v_m, \cdots, v_1>$ is a $(m+1)$-optimal solution for $[i,j]$ we have

(T2.2) $$T^{m+1}_{i,j} = T^0_{i,v_{m+1} - 1} + T^0_{v_{m+1},v_m - 1} + \sum_{l=m-1}^{1} T^0_{v_{l+1},v_l - 1} + T^0_{v_1,j} + s_{v_{m+1}} + \sum_{l=m}^{1} s_{v_l} .$$

Note that if the $<v_{m+1}, u_g, u_{g-1}, \cdots, u_1>$ selection of checkpoints achieved the time $T_{i,j}^{m+1}$ (or less) this would contradict our assumption that the rightmost checkpoint location of the $(m+1)$-optimal solutions for $[i,j]$ is $v_1 < u_1$. Therefore we must have

$$T_{i,j}^{m+1} < T_{i,v_{m+1}-1}^0 + T_{v_{m+1},u_g-1}^0 + \sum_{l=g-1}^{1} T_{u_{l+1},u_l-1}^0 + T_{u_1,j}^0 + s_{v_{m+1}} + \sum_{l=g}^{1} s_{u_l} .$$

(T2.3)

Subtracting (T2.2) from (T2.3) we get

(T2.4) $$0 < T_{v_{m+1},u_g-1}^0 - T_{v_{m+1},v_m-1}^0 + F .$$

From (T2.1) and (T2.4) we have

(T2.5) $$T_{v_{m+1},u_g-1}^0 - T_{v_{m+1},v_m-1}^0 > T_{i,u_g-1}^0 - T_{i,v_m-1}^0 .$$

Note that $i < v_{m+1} \leqslant v_m - 1 \leqslant u_g - 1$ and $<v_{m+1}>$ is the 1-optimal solution for $[i,v_m - 1]$. By Lemma 5 we have

$$T_{v_{m+1},u_g-1}^0 - T_{v_{m+1},v_m-1}^0 \leqslant T_{i,u_g-1}^0 - T_{i,v_m-1}^0$$

which contradicts (T2.5).

II. Suppose now that $v_m > u_g$. Let k be the minimum r such that $v_1 \leqslant u_1$, $v_2 \leqslant u_2$, $\cdots$, $v_{r-1} \leqslant u_{r-1}$, and $v_r > u_r$. Note that such k exists because $v_m > u_g$ (and therefore $v_g > u_g$). There are two possible cases.

(*i*) Suppose $v_{k+1} \leqslant u_k$. In this case we have $v_{k+1} \leqslant u_k < v_k < v_{k-1} \leqslant u_{k-1}$. Since $\mathbf{L}_{i,j}^m = <u_g, \cdots, u_1>$ is a m-optimal solution for $[i,j]$ we have

$$T_{i,j}^m = T_{i,u_g-1}^0 + \sum_{l=g-1}^{k} T_{u_{l+1},u_l-1}^0 + T_{u_k,u_{k-1}-1}^0 + \sum_{l=k-2}^{1} T_{u_{l+1},u_l-1}^0 + T_{u_1,j}^0 + \sum_{l=g}^{1} s_{u_l} .$$

(T2.6)

The checkpoint selection $<u_g, \cdots, u_k, v_{k-1}, \cdots, v_1>$ cannot achieve a smaller time than $T_{i,j}^m$ for $[i,j]$. Therefore we have

(T2.7) $$T_{i,j}^m \leqslant T(u,v)$$

where $T(u,v)$ is defined as

$$T(u,v) = T_{i,u_g-1}^0 + \sum_{l=g-1}^{k} T_{u_{l+1},u_l-1}^0 + T_{u_k,v_{k-1}-1}^0 + \sum_{l=k-2}^{1} T_{v_{l+1},v_l-1}^0 + T_{v_1,j}^0 + \sum_{l=g}^{k} s_{u_l} + \sum_{l=k-1}^{1} s_{v_l} .$$

From (T2.6) and (T2.7) we have

(T2.8) $$T_{u_k,u_{k-1}-1}^0 - T_{u_k,v_{k-1}-1}^0 + G \leqslant 0$$

where

$$G = \sum_{l=k-2}^{1} (T_{u_{l+1},u_l-1}^0 - T_{v_{l+1},v_l-1}^0) + T_{u_1,j}^0 - T_{v_1,j}^0 + \sum_{l=k-1}^{1} (s_{u_l} - s_{v_l}) .$$

Since $\mathbf{L}_{i,j}^{m+1} = <v_{m+1}, \cdots, v_k, v_{k-1}, \cdots, v_1>$ is a $(m+1)$-optimal solution for $[i,j]$ we have

$$T_{i,j}^{m+1} = T_{i,v_{m+1}-1}^0 + \sum_{l=m}^{k} T_{v_{l+1},v_l-1}^0 + T_{v_k,v_{k-1}-1}^0 + \sum_{l=k-2}^{1} T_{v_{l+1},v_l-1}^0 + T_{v_1,j}^0 + \sum_{l=m+1}^{1} s_{v_l}.$$

(T2.9)

Since with $u_1 > v_1$ then the $<v_{m+1}, \cdots, v_k, u_{k-1}, \cdots, u_1>$ checkpoint selection cannot achieve the time $T_{i,j}^{m+1}$ (or less) for $[i,j]$ without contradicting the optimality of $\mathbf{L}_{i,j}^{m+1}$. Therefore we have

(T2.10) $$T_{i,j}^{m+1} < T(v,u)$$

where $T(v,u)$ is defined as

$$T(v,u) = T_{i,v_{m+1}-1}^0 + \sum_{l=m}^{k} T_{v_{l+1},v_l-1}^0 + T_{v_k,u_{k-1}-1}^0$$
$$+ \sum_{l=k-2}^{1} T_{u_{l+1},u_l-1}^0 + T_{u_1,j}^0 + \sum_{l=m+1}^{k} s_{v_l} + \sum_{l=k-1}^{1} s_{u_l}.$$

Subtracting (T2.9) from (T2.10) we have

(T2.11) $$0 < T_{v_k,u_{k-1}-1}^0 - T_{v_k,v_{k-1}-1}^0 + G.$$

From (T2.8) and (T2.11) we have

(T2.12) $$T_{v_k,u_{k-1}-1}^0 - T_{v_k,v_{k-1}-1}^0 > T_{u_k,u_{k-1}-1}^0 - T_{u_k,v_{k-1}-1}^0.$$

Note that we have $v_{k+1} \leqslant u_k < v_k \leqslant v_{k-1} - 1 \leqslant u_{k-1} - 1$ and $<v_k>$ is the 1-optimal solution for $[v_{k+1}, v_{k-1} - 1]$. By Lemma 5 we know that if $s_{v_k} \geqslant s_{u_k}$ then

(T2.13) $$T_{v_k,u_{k-1}-1}^0 - T_{v_k,v_{k-1}-1}^0 \leqslant T_{u_k,u_{k-1}-1}^0 - T_{u_k,v_{k-1}-1}^0$$

which contradicts (T2.12); therefore $s_{v_k} < s_{u_k}$. From our assumption about checkpointing costs we have $r_{v_k} \leqslant r_{u_k}$.

From (T2.7) and (T2.10) we have $T(u,v) - T_{i,j}^{m+1} > T_{i,j}^m - T(v,u)$, that is

(T2.14) $$T_{u_k,v_{k-1}-1}^0 - T_{v_k,v_{k-1}-1}^0 > T_{u_k,u_{k-1}-1}^0 - T_{v_k,u_{k-1}-1}^0.$$

But since $v_{k+1} \leqslant u_k < v_k \leqslant v_{k-1} - 1 \leqslant u_{k-1} - 1$, and $r_{v_k} \leqslant r_{u_k}$ then we can apply Lemma 7 and obtain $T_{u_k,v_{k-1}-1}^0 - T_{v_k,v_{k-1}-1}^0 \leqslant T_{u_k,u_{k-1}-1}^0 - T_{v_k,u_{k-1}-1}^0$. This contradicts (T2.14) and therefore we cannot have $v_{k+1} \leqslant u_k$.

(*ii*) Suppose $v_{k+1} > u_k$. In this case we have $u_k < v_{k+1} < v_k < v_{k-1} \leqslant u_{k-1}$. We show that this is not possible. Observe that $<u_g, \cdots, u_k>$ is a $[(m+1)-k]$-optimal solution for $[i, u_{k-1} - 1]$; its rightmost checkpoint location is u_k, $u_k < v_{k+1}$. We also note that $<v_{m+1}, \cdots, v_{k+1}>$ is a $[(m+1)-k]$-optimal solution for $[i, v_k - 1]$. Since $u_{k-1} - 1 > v_k - 1$ then, by Theorem 1, the rightmost checkpoint location of the $[(m+1)-k]$-optimal solutions for $[i, u_{k-1} - 1]$ is some u such that $u \geqslant v_{k+1}$. Since $u_k < v_{k+1}$, this contradicts the optimality of $<u_g, \cdots, u_k>$ for $[i, u_{k-1} - 1]$, and the proof is complete. □

REFERENCES

[1] K. M. CHANDY, *A survey of analytic models of rollback and recovery strategies*, Computer, 5 (1975), pp. 40–47.

[2] K. M. CHANDY AND C. V. RAMAMOORTHY, *Rollback and recovery strategies for computer programs,* IEEE Trans. on Comput., 6 (1972), pp. 546–556.

[3] K. M. CHANDY, J. C. BROWNE, C. W. DISSLY AND W. R. UHRIG, *Analytic models for rollback and recovery strategies in data base systems*, IEEE Trans. Soft. Engr., 1 (1975), pp. 100–110.

[4] E. GELENBE, *On the optimum checkpoint interval*, J. Assoc. Comput. Mach., 2 (1979), pp. 259–270.

[5] E. GELENBE AND D. DEROCHETTE, *Performance of rollback recovery systems under intermittent failures*, Comm. Assoc. Comput. Mach., 6 (1978), pp. 493–499.

[6] J. GRAY, P. MCJONES, M. BLASGEN, B. LINSAY, R. LORIE, T. PRICE, F. PUTZOLU AND I. TRAIGER, *The recovery manager of the System R database manager*, Comput. Surveys, 2 (1981), pp. 223–242.

[7] I. KOREN, Z. KOREN AND S. SU, *Analysis of a recovery procedure*, Tech. Report, Technion-Israel Institute of Technology, 1983, IEEE Trans. Comput., to appear.

[8] B. RANDELL, P. LEE AND P. TRELEAVEN, *Reliability issues in computing system design*, Comput. Surveys, 2 (1978), pp. 123–166.

[9] S. M. ROSS, *Applied Probability Models With Optimization Applications*, Holden-Day, San Francisco, 1970.

[10] R. D. SCHLICHTING AND F. B. SCHNEIDER, *Fail-stop processors: an approach to designing fault-tolerant computing systems*, ACM Trans. Computer Systems, 3 (1983), 222–238.

[11] J. W. YOUNG, *A first order approximation to the optimum checkpoint interval*, Comm. Assoc. Comput. Mach., 9 (1974), pp. 530–531.

SIAM J. COMPUT.
Vol. 13, No. 3, August 1984

© 1984 Society for Industrial and Applied Mathematics
012

ON SOME VARIANTS OF THE BANDWIDTH MINIMIZATION PROBLEM*

JOSEPH Y-T. LEUNG,** OLIVER VORNBERGER† AND JAMES D. WITTHOFF‡

Abstract. We consider the following variants of the bandwidth minimization problem: (1) the cycle-bandwidth problem which for a given graph G and positive integer k, asks if there is a circular layout such that every pair of adjacent vertexes have a distance at most k, (2) the separation problem which asks if there is a linear layout such that every pair of adjacent vertexes have a distance greater than k, and (3) the cycle-separation problem which asks if there is a circular layout such that every pair of adjacent vertexes have a distance greater than k.

We show that the cycle-bandwidth problem is NP-complete for each fixed $k \geqslant 2$, the separation and cycle-separation problems are both NP-complete for each fixed $k \geqslant 1$, and the directed separation problem is NP-complete for arbitrary k. We give polynomial time algorithms for several special cases of the directed separation problem. Finally, we show the relationships of the directed separation problem with several scheduling problems by giving reductions among them.

Key words. NP-completeness, bandwidth minimization, cycle-bandwidth, cycle-separation, layout problems, multiprocessor scheduling, directed separation problem, directed graphs, forests, interval orders

1. Introduction. In recent years, there has been a great deal of interest in studying various graph layout problems. One of those problems that has been under intensive investigation is the *bandwidth minimization* problem. The problem can be formally stated as follows: We are given a graph $G=(V,E)$ and a positive integer k. A layout f of G is a bijection f: $V \rightarrow \{1,2,...,n\}$, where n is the cardinality of V. The bandwidth minimization problem asks if there is a layout f such that $|f(u)-f(v)| \leqslant k$ for all edges $\{u,v\} \in E$. In other words, we are asked to determine if there is an arrangement of the vertexes on a straight line such that any pair of adjacent vertexes have a distance at most k. If the graph is directed, we further stipulate that any layout f must satisfy $f(u) < f(v)$ whenever (u,v) is a directed edge from vertex u to vertex v. This requirement also restricts the directed graph to be acyclic.

Papadimitriou[12] had shown that the bandwidth minimization problem is NP-complete for arbitrary k. Garey et al.[4] gave a linear time algorithm for $k=2$ and showed that for arbitrary k the problem remains NP-complete even for trees.[1] Recently, Saxe[14] gave an $O(n^{k+1})$ time algorithm to solve this problem, thus showing that the bandwidth minimization problem can be solved in polynomial time for each fixed k. Monien and Sudborough[11] later improved the running time of the

* Received by the editors July 26, 1982, and in revised form July 25, 1983. This paper was typeset at AT&T Bell Laboratories, Naperville, IL 60566, using the *Troff* software developed for the UNIX® operating system. Final copy was produced February 20, 1984.

** Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201. The work of this author was supported in part by the National Science Foundation under Grant MCS - 79-04898.

† Fachbereich Mathematik-Informatik, Universitat Paderborn, 4790 Paderborn, West Germany.

‡ AT&T Bell Laboratories, Naperville, IL 60566. The work of this author was done while he was a graduate student at Northwestern University.

[1]The bandwidth minimization problem can easily be solved for $k=1$.

algorithm to $O(n^k)$. All of the above results hold for both undirected and directed graphs. Thus, it appears that for the bandwidth minimization problem there are no distinctions in complexity between the directed case and the undirected case.

In this paper, we consider several variants of the bandwidth minimization problem. The first problem is concerned with arranging the vertexes of a graph on a circle so that any pair of adjacent vertexes have a distance at most k. We call this the *cycle-bandwidth* problem.[2] The cycle-bandwidth problem arises, for example, when a set of computers is to be connected in a ring structure. Under this interconnection scheme, each computer can only communicate with its two neighbors. Thus, in order to send a message to some other computers, it needs to send it along the shortest path on the cycle. Suppose the communication pattern among the computers is known and is given by the graph $G=(V,E)$ where V is the set of computers and $\{u,v\}\in E$ whenever computer u wants to communicate with computer v. We want to find an arrangement of these computers on a circle so that every message sent can arrive at its destination in k steps. The cycle-bandwidth problem also has potential applications in VLSI when we try to lay out a set of circuit elements on a cylindrical type of surface so that two connected circuit elements are close to each other.

The second problem we shall be concerned with is the dual problem of the bandwidth minimization problem. Instead of arranging the vertexes on a straight line so that adjacent vertexes are close to each other, we seek to find an arrangement so that every pair of adjacent vertexes have a distance *greater than* k. We call this the *separation* problem. The separation problem arises, for example, when we try to line up a group of violent prisoners. Some of the prisoners are not very friendly to the others. Not only they don't talk to each other, but in any line-up if they are not separated by at least k persons, they will start killing each other. As a prison guard, you would like to find out if there is any line-up so that no fights can ever be started. We shall also study the analogous *cycle-separation* problem in which we want to find an arrangement of vertexes on a circle so that every pair of adjacent vertexes are separated by at least k vertexes.[3]

The directed separation problem appears to have some connections with the *multiprocessor scheduling* problem that has been studied extensively in the literature[3],[10],[13]. In the multiprocessor scheduling problem, we are given two positive integers k and D, and a directed acyclic graph $G=(V,E)$, where each vertex of G represents a task with an execution time of one unit and the edges of G represent operational precedence constraints among tasks (i.e., if there is a directed edge from u to v, then task u must finish execution before task v can start). We are asked to determine if there is a schedule on k identical processors such that the precedence constraints are obeyed and the schedule length is at most D. Now, suppose we are given an instance $<k,D,G>$ of the multiprocessor scheduling problem, and without loss of generality we may assume that the number of tasks is exactly kD (for otherwise we can introduce dummy tasks to satisfy this requirement). If the instance $<k-1,G>$ of the directed separation problem has a solution, then the instance $<k,D,G>$ of the multiprocessor scheduling problem also has a solution. This is because we can take a solution of the directed separation problem, and schedule the first k tasks in the first time unit, the next k tasks in the second time unit, and so on. The schedule thus constructed clearly obeys all precedence constraints and has a

---

[2]The cycle-bandwidth problem is only meaningful for undirected graphs.

[3]Again, the cycle-separation problem is only meaningful for undirected graphs.

schedule length D. However, the converse of the above statement is not true; i.e., that the instance $<k,D,G>$ of the multiprocessor scheduling has a solution does not imply that the instance $<k-1,G>$ of the directed separation problem must have a solution. This can be seen from the example task system G given in Figure 1. Clearly, G can

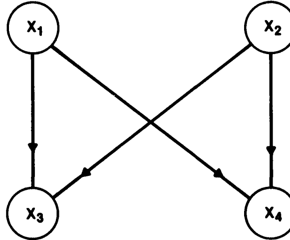


FIG. 1. *Example task system showing the multiprocessor scheduling problem has a solution while the directed separation problem does not.*

be scheduled on two processors in two time units. However, the instance $<1,G>$ of the directed separation problem does not have a solution. Despite the apparent differences, we shall see in the sequel that there seems to be a strong relationship between the complexities of these two problems.

In this paper, we shall give complexity results concerning these problems. First, we show that the cycle-bandwidth problem is NP-complete for $k=2$.[4] Next, we show that the separation problem and the cycle-separation problem are both NP-complete for $k=1$. We then show that the directed separation problem is NP-complete for arbitrary k, but can be solved in polynomial time for $k=1$. For fixed $k>1$, we have not been able to come up with a polynomial time algorithm nor to show that it is NP-complete. However, we give some evidence of the difficulties of this problem by relating it to several open problems in scheduling theory. Finally, we show that the directed separation problem can be solved in polynomial time for two special classes of directed acyclic graphs - forests and interval orders (we shall define this kind of partial order in the sequel), even when k is arbitrary.

The NP-completeness of the cycle-bandwidth problem is somewhat surprising since it is tempting to try to extend the algorithms of Saxe[14] or Monien and Sudborough[11] to solve the cycle-bandwidth problem. Indeed, if the graph is connected, their algorithms *can* be modified to solve the cycle-bandwidth problem. Trouble occurs when the graph is not connected. The bandwidth of a graph is simply the maximum of the bandwidths of its connected components. (The *bandwidth of a graph G* is the smallest integer k for which the instance $<k,G>$ of the bandwidth minimization problem has a solution. The *cycle-bandwidth of a graph* is defined analogously.) Thus, to solve the bandwidth minimization problem, all we need to do is to solve each of its connected components. The same approach, however, does not work for the cycle-bandwidth problem since the cycle-bandwidth of a graph is not necessarily the maximum of the cycle-bandwidths of its connected components. To see this, consider the graph consisting of two vertex-disjoint cycles. The cycle-bandwidth of this graph is two, whereas the cycle-bandwidth of each of its connected component

[4] The cycle-bandwidth problem can easily be solved for $k=1$.

is one. This phenomenon is similar to the relationship between the subtree isomorphism problem and the subforest isomorphism problem discussed in [6] which shows that the subforest isomorphism problem is NP-complete whereas the subtree isomorphism problem is solvable in polynomial time.

The cycle-bandwidth problem has another interesting relationship with the bandwidth minimization problem which can be used as a basis for designing approximation algorithms with a constant worst-case performance bound. That is: (1) the cycle-bandwidth of a graph G is no larger than the bandwidth of G and (2) the bandwidth of a graph G is no larger than twice the cycle-bandwidth of G. Now suppose we have an algorithm X which, when given a graph G, produces a linear layout with the minimum bandwidth. (The *bandwidth of a linear layout* f is defined to be $\max_{\{u,v\}\in E}\{|f(u)-f(v)|\}$. The *cycle-bandwidth of a circular layout* is defined analogously.) We can devise an approximation algorithm X' for the cycle-bandwidth problem having a worst-case performance bound of two as follows: X' merely calls X to obtain a linear layout f and wraps it around to produce a circular layout f'. Since the cycle-bandwidth of f' is the same as the bandwidth of f and since the cycle-bandwidth of a graph G is at least one-half of the bandwidth of G, X' will produce a solution no larger than twice the optimum solution. (Similarly, if X is only an approximation algorithm for the bandwidth minimization problem having a worst-case performance bound of β, then X' will be an approximation algorithm for the cycle-bandwidth problem with a worst-case performance bound of 2β.) Conversely, suppose we have an algorithm Y which, when given a graph G, produces a circular layout with the minimum cycle-bandwidth. We can obtain an approximation algorithm Y' for the bandwidth minimization problem as follows: Y' calls Y to obtain a circular layout f and linearize it to obtain a linear layout f'. This is done by assigning the two vertexes on a diameter of the circle to positions 1 and n (where n is the cardinality of the vertex set of the graph), and assigning vertexes on the top half of the circle to odd positions and vertexes on the bottom half of the circle to even positions. The linear layout f' thus obtained clearly has a bandwidth no larger than twice the cycle-bandwidth of f. Since the bandwidth of a graph G must be at least as large as the cycle-bandwidth of G, the solution produced by Y' is no larger than twice the optimum solution; hence, the worst-case performance bound of Y' is again two. Similar remark applies to Y' if Y is only an approximation algorithm with a worst-case performance bound of β.

The separation problem appears to be harder for undirected graphs than for directed graphs, at least for $k=1$. This is to be contrasted with the bandwidth minimization problem in which there are no distinctions in complexity between the undirected case and the directed case. The separation problem also has an interesting comparison with the bandwidth minimization problem in that while it appears to be harder than the bandwidth minimization problem for the undirected case, it seems to be easier for the directed case. This follows from the observations that for undirected graphs the separation problem is NP-complete for each fixed k whereas the bandwidth minimization problem can be solved in polynomial time. On the other hand, the directed bandwidth minimization problem is NP-complete (for arbitrary k) for trees whereas the directed separation problem can be solved in polynomial time.

We mentioned earlier that there seems to be a strong relationship between the complexities of the multiprocessor scheduling problem and the directed separation problem. Our remark is based on the following observations. First, we have been able to reduce the multiprocessor scheduling problem to the directed separation problem, thus showing that the directed separation problem is at least as hard as the

multiprocessor scheduling problem. (In fact, the NP-completeness of the directed separation problem follows from this reduction and the fact that the multiprocessor scheduling problem is NP-complete for arbitrary k[15].) Although we have been unable to provide a reduction in the opposite direction, we have not found any special cases for which the multiprocessor scheduling problem can be solved in polynomial time[3], [10], [3] yet the directed separation problem cannot. Indeed, as we shall see in the sequel, our polynomial time algorithms for the directed separation problem are all natural extensions of the corresponding algorithms for the multiprocessor scheduling problem[3], [10], [13]. This state of affairs does suggest that the directed separation problem is not much harder than the multiprocessor scheduling problem.

In addition to the multiprocessor scheduling problem, the directed separation problem is also related to several other scheduling problems - these are: (1) *multiprocessor scheduling with integer release times and deadlines* (abbreviated by MSIRD), (2) *multiprocessor scheduling with rational release times and deadlines* (abbreviated by MSRRD), and (3) *multiprocessor scheduling on uniform processor systems* (abbreviated by MSUPS). The MSIRD problem is exactly the same as the multiprocessor scheduling problem except that each task x_i has associated with it an integer release time r_i and deadline d_i, and we are simply asked to determine if there is a schedule such that each task x_i executes in the time interval $[r_i,d_i]$ and all precedence constraints are observed. The MSRRD problem is the same as the MSIRD problem except that the release times and deadlines can be rational numbers. Finally, the MSUPS problem is the same as the multiprocessor scheduling problem except that each processor P_i has associated with it a speed s_i so that a task takes $1/s_i$ time units to execute on processor P_i.

It is known that both the multiprocessor scheduling problem[3] and the MSIRD problem[5] are solvable in polynomial time for $k \leqslant 2$ processors. However, their complexities are still open for each fixed $k>2$. The MSRRD problem[7] and the MSUPS problem can be solved in polynomial time for $k=1$ processor, but their complexities are open for each fixed $k>1$. Of course, all these problems are NP-complete for arbitrary k since the multiprocessor scheduling problem is a special case of them and the multiprocessor scheduling problem is known to be NP-complete for arbitrary k. In this paper, we shall show that both the multiprocessor scheduling problem and the MSIRD problem are reducible to the directed separation problem, and that the directed separation problem is reducible to both the MSRRD problem and the MSUPS problem. Although our results do not provide answers to the above open questions, we hope that they can be used in the future as a vehicle in settling these issues.

In the next section, we shall give the NP-completeness proofs as well as various reductions among the directed separation problem and the scheduling problems. In Section 3 we shall give polynomial time algorithms for several special cases of the directed separation problem. Finally, in the last section we shall remark on the implications of our results and directions for future research.

2. Reductions. In this section, we shall first show the NP-completeness of the cycle-bandwidth problem, the separation problem, and the cycle-separation problem. Next, we show that both the multiprocessor scheduling problem and the MSIRD problem are reducible to the directed separation problem. Finally, we show that the directed separation problem is reducible to both the MSRRD problem and the MSUPS problem.

THEOREM 1. *The cycle-bandwidth problem is* NP-*complete for* $k=2$.

Proof. It is easy to see that the cycle-bandwidth problem is in NP. To complete the proof, we shall reduce the 3-partition problem to it. The 3-partition problem had been shown to be NP-complete in the strong sense[6] and it can be stated as follows: Given a positive integer b and a set of $3m$ positive integers $A=\{a_1,a_2,...,a_{3m}\}$[5] such that $\sum_{j=1}^{3m} a_j=mb$ and $b/4<a_j<b/2$ for each $1\leqslant j\leqslant 3m$, is there a partition of A into m disjoint sets $A_1,A_2,\cdots A_m$ such that $\sum_{a_j\in A_i} a_j=b$ for each $1\leqslant i\leqslant m$? (Note that the constraints on the a_j imply that each such A_i must contain exactly three elements from A.) Since the 3-partition problem is NP-complete in the strong sense, we may assume that the sizes of the integers in A are bounded by a polynomial function of the cardinality of A.

Given an instance $A=\{a_1,a_2,...,a_{3m}\}$ and b of the 3-partition problem, we construct a graph $G=(V,E)$ as follows. Informally, G consists of: (1) a cycle of $m(b+1)+1$ vertexes, (2) "ridges" placed in the appropriate places of the cycle, and (3) $3m$ chains such that the jth chain has exactly a_j vertexes. Formally, let $V_1=\{x_0,x_1,...,x_{m(b+1)}\}$ and $E_1=\{\{x_{i-1},x_i\}|1\leqslant i\leqslant m(b+1)\}\bigcup\{x_{m(b+1)},x_0\}$. ($V_1$ and E_1 make up a cycle of $m(b+1)+1$ vertexes.) Let $V_2=\{y_0,y_1,...,y_m\}$ and $E_2=\{\{x_{i(b+1)-1},y_i\},\{x_{i(b+1)},y_i\}|1\leqslant i\leqslant m\}\bigcup\{\{x_{m(b+1)},y_0\},\{x_0,y_0\},\{y_m,y_0\}\}$. ($V_2$ and E_2 make up the "ridges" placed on the cycle.) Let $V'_j=\{z_{j,1},z_{j,2},...,z_{j,a_j}\}$ and $E'_j=\{\{z_{j,i-1},z_{j,i}\}|2\leqslant i\leqslant a_j\}$ for each $1\leqslant j\leqslant 3m$. (V'_j and E'_j make up the $3m$ chains.) Finally, let $V=V_1\bigcup V_2\bigcup V'_1\bigcup\cdots\bigcup V'_{3m}$ and let $E=E_1\bigcup E_2\bigcup E'_1\bigcup\cdots\bigcup E'_{3m}$. We note that the graph G has $n=2m(b+1)+2$ vertexes and hence can be constructed in polynomial time. Figure 2 shows a picture of G.

We now show that A has a partition $A_1,A_2,...,A_m$ if and only if G has a circular layout f with cycle-bandwidth no larger than 2. First, suppose A has a partition $A_1,A_2,...,A_m$ such that $A_j=\{a_{j_1},a_{j_2},a_{j_3}\}$ for each $1\leqslant j\leqslant m$. We can obtain a layout f as follows: $f(x_i)=2i+1$ for each $0\leqslant i\leqslant m(b+1)$, $f(y_i)=2i(b+1)$ for each $1\leqslant i\leqslant m$ and $f(y_0)=2m(b+1)+2=n$, and for each $1\leqslant j\leqslant m$, $f(z_{j_1,i})=2[(j-1)(b+1)+i]$ for each $1\leqslant i\leqslant a_{j_1}$, $f(z_{j_2,i})=2[(j-1)(b+1)+a_{j_1}+i]$ for each $1\leqslant i\leqslant a_{j_2}$ and $f(z_{j_3,i})=2[(j-1)(b+1)+a_{j_1}+a_{j_2}+i]$ for each $1\leqslant i\leqslant a_{j_3}$. It is easy to verify that f has a cycle-bandwidth two.

Conversely, suppose G has a circular layout f with cycle-bandwidth no larger than two. We shall show that A also has a partition. Consider the vertex $x_{m(b+1)}$. Without loss of generality, we may assume that $f(x_{m(b+1)})=n-1$; for otherwise we can renumber the vertexes to satisfy this requirement. The vertex $x_{m(b+1)}$ is adjacent to four other vertexes - namely, x_0, y_0,y_m, and $x_{m(b+1)-1}$. Since f has a cycle-bandwidth no larger than two, these vertexes must be assigned the numbers $1,n,n-2$, and $n-3$. Because of the adjacency relation among these vertexes, there is only one way (disregarding symmetry) to number these vertexes to maintain a cycle-bandwidth no larger than two - namely, $f(x_0)=1$, $f(y_0)=n$, $f(y_m)=n-2$, and $f(x_{m(b+1)-1})=n-3$. Now, notice that there is a path with $m(b+1)$ vertexes from the vertex x_0 to the vertex $x_{m(b+1)-1}$ (including both endpoints). The numbers used by f for these vertexes must be chosen from the set $\{1,2,...,n-3\}$, since $n-2$, $n-1$, and n have already been assigned to the vertexes y_m, $x_{m(b+1)}$, and y_0. Since vertex x_0 and vertex $x_{m(b+1)-1}$ have already been assigned the numbers 1 and $n-3$ respectively, and since the

[5] A is actually a multiset since we allow duplicate integers to be in A. For convenience in presentation, we shall refer to a multiset simply as a set.

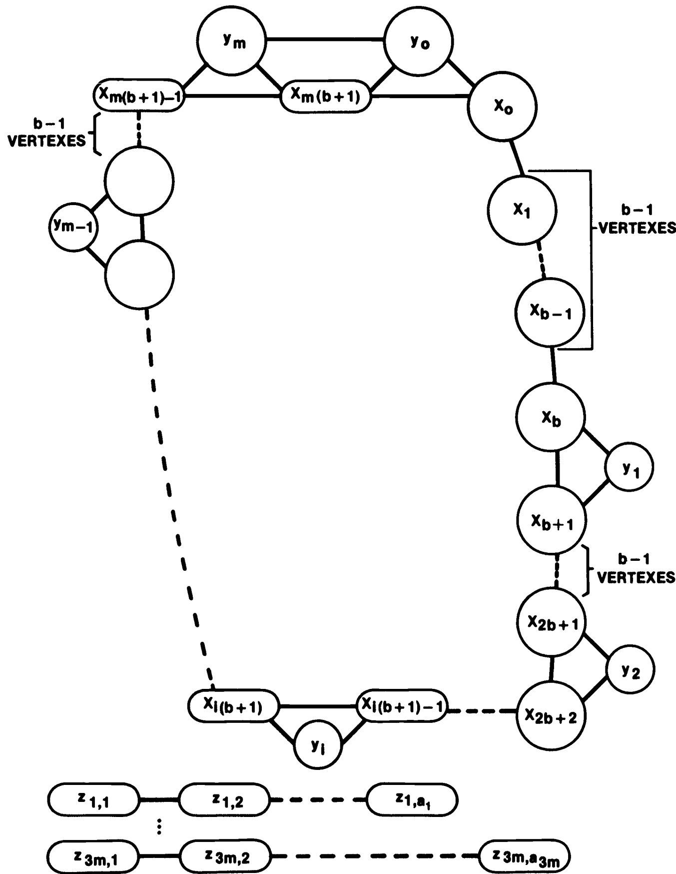


FIG. 2. *Illustrating the construction in Theorem* 1.

numbers assigned to any two neighbors on the path can differ by at most two (because of the cycle-bandwidth two condition), there is only one way to number the vertexes on this path - namely, $f(x_i)=2i+1$.

Now, consider the vertexes $y_1,y_2,\ldots,y_{m-1}$. Since y_i is adjacent to both $x_{i(b+1)-1}$ and $x_{i(b+1)}$ which have been assigned the numbers $2i(b+1)-1$ and $2i(b+1)+1$ respectively, the only way to satisfy the cycle-bandwidth two condition is by having $f(y_i)=2i(b+1)$. From the above discussions, we see that the numbers left for the $3m$ chains $V'_1,V'_2,\ldots,V'_{3m}$ consist of m sets of b consecutive even numbers $I_1=\{2,4,\ldots,2b\}$, $I_2=\{2b+4,2b+6,\ldots,4b+2\},\ldots,$ $I_m=\{2[(m-1)(b+1)+1], 2[(m-1)(b+1)+2],\ldots,n-4\}$. Since only even numbers are left, a chain having l vertexes must use l consecutive even numbers inorder to satisfy the cycle-bandwidth two condition. Moreover, since the numbers in I_j and I_{j+1} differ by more than two, each chain must be assigned numbers from only one set, say I_k. We have m sets of numbers with b numbers each and a total of mb vertexes in all chains. It follows that we can arrange the $3m$ chains into m groups such that the sum of the numbers of vertexes of the chains in each group is b. Since each number $a_j \in A$ was represented by a chain having a_j vertexes, we must have a partition $A_1,A_2,\ldots,A_m$ such that $\sum_{a_j \in A_i} a_j=b$ for each $1 \leqslant i \leqslant m$. □

We note that it is easy to modify the above proof to show that the cycle-bandwidth problem is NP-complete for each fixed $k>2$.

THEOREM 2. *The separation problem is* NP-*complete for* $k=1$.

Proof. The separation problem is clearly in NP. We shall reduce the NP-complete hamiltonian path problem[6] to it. In the hamiltonian path problem, we are given a graph $G=(V,E)$ and we are asked to determine if there is a path going through each vertex in V exactly once. Given an instance $G=(V,E)$ of the hamiltonian path problem, we construct an instance $\bar{G}=(V,\bar{E})$ of the separation problem, where $\bar{E}=\{\{u,v\}|u,v\in V,u\neq v$ and $\{u,v\}\notin E\}$; i.e., $\bar{G}$ is the complementary graph of G. It is easy to see that G has a hamiltonian path if and only if $\bar{G}$ has a linear layout such that any pair of adjacent vertexes are separated by one or more vertexes. □

COROLLARY 3. *The cycle-separation problem is* NP-*complete for* $k=1$.

Proof. Reduce the hamiltonian cycle problem[6] to it. □

We note that it is easy to reduce the separation problem with $k=k_0$ to the separation problem with $k=k_0+1$, thus showing that the separation problem is NP-complete for each fixed $k\geqslant 1$. The same remark applies to the cycle-separation problem as well.

THEOREM 4. *The multiprocessor scheduling problem is reducible to the directed separation problem.*

Proof. Given an instance $<k,D,G_1=(V_1,E_1)>$ of the multiprocessor scheduling problem, we construct an instance $<k,G_2=(V_2,E_2)>$ of the directed separation problem as follows. Without loss of generality, assume that G_1 has kD vertexes and let $V_1=\{x_1,x_2,\ldots,x_{kD}\}$. Let $G=(V,E)$ be the graph consisting of $k(D+1)$ vertexes, where $V=\{y_{i,j}|0\leqslant i\leqslant D,1\leqslant j\leqslant k\}$ and $E=\{(y_{i,p},y_{i+1,q})|0\leqslant i\leqslant D-1,1\leqslant p,q\leqslant k\}$. G_2 is simply the union of G_1 and G; i.e., $V_2=V_1\bigcup V$ and $E_2=E_1\bigcup E$. Clearly, G_2 can be constructed in polynomial time.

To complete the proof, we need to show that the instance $<k,D,G_1>$ of the multiprocessor scheduling problem has a solution if and only if the instance $<k,G_2>$ of the directed separation problem has a solution. First, suppose there is a schedule of G_1 on k identical processors. For each $1\leqslant i\leqslant D$, let $\{x_{i,1},x_{i,2},\ldots,x_{i,k}\}$ be the set of k tasks scheduled in the ith time slot. We can obtain a layout f as follows: $f(x_{i,j})=(2i-1)k+j$ for each $1\leqslant i\leqslant D$ and $1\leqslant j\leqslant k$, and $f(y_{i,j})=2ik+j$ for each $0\leqslant i\leqslant D$ and $1\leqslant j\leqslant k$. It is easy to verify that f satisfies the separation requirement.

Conversely, suppose f is a layout that satisfies the separation requirement. We first consider how f assigns numbers to the vertexes in $G=(V,E)$. For each $0\leqslant i\leqslant D$, let s_i and b_i denote the smallest and the largest number assigned by f to the vertexes in the set $\{y_{i,1},y_{i,2},\ldots,y_{i,k}\}$, respectively. Clearly, we have $b_i\geqslant s_i+(k-1)$, since there are k vertexes in the set. Since there is a directed edge from each vertex in the set $\{y_{i,1},y_{i,2},\ldots,y_{i,k}\}$ to each vertex in the set $\{y_{i+1,1},y_{i+1,2},\ldots,y_{i+1,k}\}$, we must have $s_{i+1}>b_i+k$ in order for f to satisfy the separation requirement. Now, define the interval $I_i=[s_i,b_i+k]$ for each $0\leqslant i\leqslant D-1$ and $I_D=[s_D,b_D]$. From the above argument, we have $I_0<I_1<\cdots<I_D$. Furthermore, since we have exactly $n=(2D+1)k$ vertexes in G_2, we must have $b_D\leqslant(2D+1)k$ and $s_0\geqslant 1$. Now, each of the intervals $I_0,I_1,\ldots,I_{D-1}$ contains at least $2k$ numbers and the interval I_D contains at least k numbers. It then follows that for each $0\leqslant i\leqslant D-1$ the interval I_i contains exactly the numbers $2ki+1,2ki+2,\ldots,2ki+2k$. From the definition of I_i, the remaining unassigned numbers must consist of intervals $J_1,J_2,\ldots,J_D$ where $J_i=\{(2i-1)k+1,(2i-1)k+2,\ldots,(2i-1)k+k\}$ for each $1\leqslant i\leqslant D$. These numbers are used to assign to vertexes in G_1. Since each

interval J_i contains exactly k consecutive numbers, any two vertexes of G_1 assigned to the same interval must be independent in order for f to satisfy the separation requirement. Thus, by scheduling the vertexes $f^{-1}((2i-1)k+1)$, $f^{-1}((2i-1)k+2),\ldots,$ $f^{-1}((2i-1)k+k)$ in the ith time slot for each $1 \leqslant i \leqslant D$, we obtain a valid schedule of G_1 on k identical processors. □

COROLLARY 5. *The* MSIRD *problem is reducible to the directed separation problem.*

Proof. If a task $x_i \in V_1$ has a release time $r_i > 0$ and a deadline $d_i < D$, then we add the edges $\{(y_{r_i-1,j}, x_i) \mid 1 \leqslant j \leqslant k\}$ and $\{(x_i, y_{d_i+1,j}) \mid 1 \leqslant j \leqslant k\}$. This will guarantee that x_i be assigned a number from one of the intervals $J_{r_i+1}, J_{r_i+2}, \ldots, J_{d_i}$, and hence its release time and deadline constraints are satisfied. □

COROLLARY 6. *The directed separation problem is* NP-*complete for arbitrary k.*

Proof. The directed separation problem is clearly in NP. The result then follows from Theorem 4 and the fact that the multiprocessor scheduling problem is NP-complete for arbitrary k. □

THEOREM 7. *The directed separation problem is reducible to the* MSRRD *problem.*

Proof. Given an instance $<k, G_1=(V_1,E_1)>$ of the directed separation problem, we construct an instance $<k+1, G_2=(V_2,E_2), r, d>$ of the MSRRD problem as follows, where r and d are the release time function and deadline function respectively (i.e., for each $x \in V_2$, $r(x)$ and $d(x)$ denote the release time and the deadline of x respectively). In order to simplify our discussions, we shall first assume that $n = |V_1|$ is divisible by $k+1$. For n not divisible by $k+1$ we have to make some slight modifications, which will be described later. Let $V' = \{x_i \mid 0 \leqslant i \leqslant k\}$ and $V'' = \{y_i \mid 0 \leqslant i \leqslant k\}$. G_2 is simply G_1 and the isolated vertexes in V' and V''; i.e., $V_2 = V_1 \cup V' \cup V''$ and $E_2 = E_1$. The release time function and the deadline function are defined as follows:

$$r(x) = \begin{cases} 1 & \text{for } x \in V_1, \\ \dfrac{i}{k+1} & \text{for } x = x_i,\ 0 \leqslant i \leqslant k, \\ \dfrac{n+i}{k+1} + 1 & \text{for } x = y_i,\ 0 \leqslant i \leqslant k \end{cases}$$

and

$$d(x) = \begin{cases} \dfrac{n+k}{k+1} + 1 & \text{for } x \in V_1, \\ r(x)+1 & \text{for } x \in V' \cup V'' \end{cases}$$

Since G_2 has $n+2(k+1)$ vertexes, the construction can clearly be done in polynomial time.

We now show that the instance $<k, G_1>$ of the directed separation problem has a solution if and only if the instance $<k+1, G_2, r, d>$ of the MSRRD problem has a solution. First, suppose $f: V_1 \to \{1,2,\ldots,n\}$ is a layout that satisfies the separation requirement. We shall construct a valid schedule of G_2 on $k+1$ identical processors. Let the processors be denoted by $P_0, P_1, P_2, \ldots, P_k$ Let $s(x)$ denote the starting time of

task x in G_2 and $p(x)$ denote the index of the processor on which task x executes. Then, we have

$$s(x) = \begin{cases} r(x) & \text{for } x \in V' \bigcup V'', \\ \dfrac{k+f(x)}{k+1} & \text{for } x \in V_1 \end{cases}$$

and

$$p(x) = \begin{cases} i & \text{for } x=x_i \text{ or } x=y_i, \\ (f(x)-1)\bmod(k+1) & \text{for } x \in V_1 \end{cases}$$

It is easy to verify that the release time and deadline constraints of each task are met, and that no processor executes more than one task at the same time. Moreover, if (u,v) is a directed edge in G_2, then by our construction, (u,v) must be a directed edge in G_1. Thus, $f(v)-f(u) \geqslant k+1$ and hence

$$s(v)-s(u) = \frac{k+f(v)}{k+1} - \frac{k+f(u)}{k+1} = \frac{f(v)-f(u)}{k+1} \geqslant 1.$$

Therefore, all precedence constraints are observed in the schedule.

Conversely, suppose there is a valid schedule of G_2 on $k+1$ identical processors. For each $x \in V_2$, let $s(x)$ denote the starting time of task x in the schedule. Because of the deadline constraint, each task $x \in V' \bigcup V''$ must be executed immediately after it is released; i.e., $s(x)=r(x)$ for each $x \in V' \bigcup V''$. Consider the time interval $[1,(n+k)/(k+1)+1]$, which is the time interval the tasks in V_1 can execute. During this interval of time, the total amount of time devoted to tasks in $V' \bigcup V''$ is exactly k. Thus, the total amount of time available to tasks in V_1 is exactly n. Since V_1 has n tasks, we see that the schedule cannot have any idle time in the time interval $[1,(n+k)/(k+1)+1]$. This fact, along with the release time and deadline constraints of tasks in V', implies that tasks in V_1 are started at times $1+i/(k+1)$ for $0 \leqslant i \leqslant n-1$. We can obtain a layout f of G_1 as follows: $f(x)=(s(x)-1)(k+1)+1$ for each $x \in V_1$. From the above discussions, we see that $f\colon V_1 \rightarrow \{1,2,\ldots,n\}$. Moreover, if (u,v) is a directed edge in G_1, then by our construction, (u,v) is also a directed edge in G_2. Thus, $s(v)-s(u) \geqslant 1$ and hence $f(v)-f(u)=[(s(v)-1)(k+1)+1]-[(s(u)-1)(k+1)+1]=(s(v)-s(u))(k+1) \geqslant k+1 > k$. It follows that f satisfies the separation requirement.

We now come back to the point when n is not divisible by $k+1$. In this case, we introduce additional tasks $z_j, z_{j+1},\ldots,z_k$, where $j=n \bmod(k+1)$. By setting $r(z_i)=r(y_i)-1$ and $d(z_i)=r(y_i)$ for each $j \leqslant i \leqslant k$, we can enforce that tasks in V_1 are executed as soon as a processor becomes available for assignment. Thus, the same argument applies. □

COROLLARY 8. *The directed separation problem is reducible to the* MSUPS *problem.*

Proof. Given an instance $<k,G_1=(V_1,E_1)>$ of the directed separation problem, we construct an instance $<k+2,D,G_2=(V_2,E_2),u>$ of the MSUPS problem as follows, where u is the processor speed function (i.e., $u(P_i)$ denotes the speed of processor P_i

for each $0 \leqslant i \leqslant k+1$). The construction is very similar to the construction in Theorem 7, except that we have one "high-speed" processor in addition to the $k+1$ "normal" processors. This "high-speed" processor is used to simulate the individual release times and deadlines which are not allowed in the MSUPS problem. In order to simplify our discussions, we shall first assume that $n=|V_1|$ is divisible by $k+1$; for n not divisible by $k+1$ we have to make some slight modifications. Let $V'=\{x_i \mid 0 \leqslant i \leqslant k\}$, $V''=\{y_i \mid 0 \leqslant i \leqslant k\}$ and $V'''=\{h_i \mid 1 \leqslant i \leqslant n+3k+2\}$. Let $V_2=V_1 \bigcup V' \bigcup V'' \bigcup V'''$. Let $E'=\{(x_0,h_{k+2})\} \bigcup \{(h_i,x_i),(x_i,h_{i+k+2}) \mid 1 \leqslant i \leqslant k\}$, $E''=\{(h_{n+k+1+i},y_i),(y_i,h_{n+2k+3+i}) \mid 0 \leqslant i \leqslant k-1\} \bigcup \{(h_{n+2k+1},y_k)\}$ and $E'''=\{(h_i,h_{i+1}) \mid 1 \leqslant i \leqslant n+3k+1\}$. Let $E_2=E_1 \bigcup E' \bigcup E'' \bigcup E'''$. Let $u(P_{k+1})=k+1$ and $u(P_i)=1$ for each $0 \leqslant i \leqslant k$. Finally, let $D=(n+k)/(k+1)+2$. It is clear that the construction can be done in polynomial time.

The main idea of the proof is that the chain V''' must be executed entirely on processor P_{k+1} in order to meet the overall deadline D. This fact, along with the precedence relations among tasks in V''' and tasks in V', will ensure that each x_i be executed in a specified time interval; specifically, x_i can only be executed in the time interval $[i/(k+1),1+i/(k+1)]$. Similarly, each y_i can only be executed in the time interval $[1+(n+i)/(k+1),2+(n+i)/(k+1)]$. This structure will then enable us to use the same argument as in Theorem 7 to show that the instance $<k,G_1>$ of the directed separation problem has a solution if and only if the instance $<k+2,D,G_2,u>$ of the MSUPS problem has a solution. Finally, if n is not a multiple of $k+1$, then we can introduce additional tasks $z_j,z_{j+1},\ldots,z_k$ as we did in Theorem 7. □

3. Algorithms. The purpose of this section is to show that several scheduling algorithms[3],[10],[13] for the multiprocessor scheduling problem can be adapted to solve the directed separation problem. First, we show that Coffman-Graham's algorithm[3] can be adapted to solve the directed separation problem for $k=1$. Next, we show that Hu's algorithm[10] can be adapted to solve the directed separation problem for forests and arbitrary k. Finally, we show that Papadimitriou-Yannakakis's algorithm[13] can be adapted to solve the directed separation problem for interval orders and arbitrary k. These results are given in the next three sections.

3.1. Arbitrary directed acyclic graphs and $k=1$. Coffman-Graham's scheduling algorithm[3] takes as input a directed acyclic graph $G=(V,E)$ without transitive edges,[6] and produces a minimum-length schedule of G on two identical processors. We shall first review how Coffman-Graham algorithm works and then show that it can be adapted to solve the directed separation problem for $k=1$. The algorithm first assigns a label to each vertex of G (which we shall call Coffman-Graham labeling algorithm and will be described later). After the labeling is done, the schedule is constructed as follows: Whenever a processor becomes idle, scan the list of unassigned vertexes instantaneously. From among those unassigned vertexes, choose the one all of whose *immediate predecessors*[7] have already been executed and which has the largest label. Assign this vertex to the idle processor.

For each vertex $x \in V$, let $S(x)$ denote the set of *immediate successors*[7] of x.

[6]If the graph has transitive edges, we can use the algorithm given in [1] to remove all transitive edges.

[7]A vertex u is said to be an immediate predecessor of a vertex v if there is a directed edge from u to v; v is also called an immediate successor of u.

Coffman-Graham's labeling algorithm assigns to each vertex $x \in V$ an integer label $\alpha(x) \in \{1,2,...,n\}$, where n is the cardinality of V, and is defined recursively as follows: (1) An arbitrary vertex x with $S(x)=\varnothing$ is chosen and $\alpha(x)$ is defined to be one. (2) Suppose for some $i \leqslant n$ the integers $1,2,...,i-1$ have already been assigned. For each vertex x for which α has been defined on all elements of $S(x)$, let $N(x)$ denote the decreasing sequence of integers formed by ordering the set $\{\alpha(y) \mid y \in S(x)\}$. Choose a vertex x^* such that $N(x^*) \leqslant N(x)$[8] for all such vertexes x, and define $\alpha(x^*)$ to be i. (3) Repeat step (2) until all vertexes have been assigned a label.

We are now ready to state the algorithm for the directed separation problem. This is given as algorithm A below. The algorithm takes as input a directed acyclic graph $G=(V,E)$ without transitive edges, and produces a layout f, whenever one exists, such that $f(v)>f(u)+1$ for all $(u,v) \in E$.

ALGORITHM A

1. Label all vertexes of G by Coffman-Graham's labeling algorithm.
2. From the list of unassigned vertexes, assign the next "ready" vertex to the layout whose label is the largest among all "ready" vertexes, where a vertex is defined to be "ready" to be put into the layout if all of its immediate predecessors are already in the layout and none of its immediate predecessors was the last vertex put into the layout.
3. Repeat step 2 until all vertexes have been assigned, in which case the algorithm has successfully produced a layout, or there are some unassigned vertexes but none of them are ready, in which case the algorithm reports that no such layout is possible. □

We note that the layout produced by Algorithm A is in general not identical to the schedule produced by Coffman-Graham's scheduling algorithm. It is therefore somewhat surprising to find that the correctness proof of Algorithm A is very similar to the correctness proof of Coffman-Graham's scheduling algorithm. This is shown in the next theorem.

THEOREM 9. *Algorithm A correctly solves the directed separation problem for* $k=1$.

Proof. By the nature of the algorithm, it is easy to see that if Algorithm A terminates successfully, then the layout produced by the algorithm must satisfy the separation requirement. To complete the proof, we need to show that if Algorithm A terminates unsuccessfully, then there can be no layout satisfying the separation requirement. The proof of this statement is very similar to the proof that Coffman-Graham's scheduling algorithm produces a minimum-length schedule.

We shall divide the partial layout produced by Algorithm A (at the time it unsuccessfully terminates) into segments. To this end, we define vertexes V_i and W_i recursively as follows: (1) V_0 is defined to be the last vertex put into the partial layout when Algorithm A terminates unsuccessfully. W_0 is undefined. (2) In general, for $i \geqslant 1$, W_i is defined to be the last vertex put into the partial layout before V_{i-1} such that $\alpha(W_i)<\alpha(V_{i-1})$. V_i is defined to be the vertex put into the partial layout

[8] Let $N=(n_1,n_2,...,n_t)$ and $N'=(n'_1,n'_2,...,n'_{t'})$ be two decreasing sequences of positive integers; i.e., $n_i>n_{i+1}$ and $n'_j>n'_{j+1}$ for $1 \leqslant i<t$ and $1 \leqslant j<t'$ We say that $N<N'$ if either (a) for some i, $1 \leqslant i \leqslant t$, we have $n_j=n_j'$ for all $1 \leqslant j \leqslant i-1$ and $n_i<n_i'$, or (b) $t<t'$ and $n_j=n_j'$ for all $1 \leqslant j \leqslant t$. We say that $N=N'$ if $t=t'$ and $n_j=n'_j$ for all $1 \leqslant j \leqslant t$. We say that $N \leqslant N'$ if either $N<N'$ or $N=N'$.

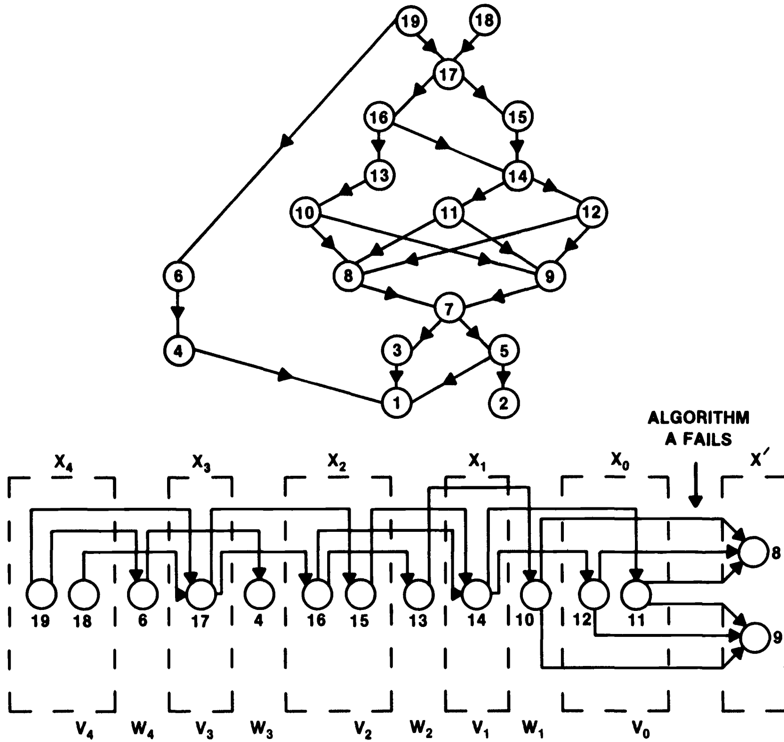


FIG. 3. (a) *Graph labeled by Coffman-Graham labeling algorithm.*
(b) *Layout produced by Algorithm A with definitions of V_i, W_i, X_i,* and X'.

immediately before W_i. Suppose now we are able to define W_l but not W_{l+1}. We shall divide the partial layout into $l+1$ segments $X_0, X_1, \ldots, X_l$, as follows: (1) For each $0 \leqslant i \leqslant l-1$, X_i is the set of vertexes between V_i and W_{i+1}, including V_i but not W_{i+1}. (2) X_l is the set of vertexes starting from the first vertex in the partial layout up until V_l, and including V_l. Finally, we define X' to be the set of *unassigned* vertexes all of whose immediate predecessors have been put into the partial layout when Algorithm A terminates unsuccessfully. A graph for which Algorithm A terminates unsuccessfully is shown in Figure 3.a. (This graph is taken from the example given in [2] which is used to illustrate the correctness proof of Coffman-Graham's scheduling algorithm.) The vertexes of the graph are labeled by Coffman-Graham's labeling algorithm. In Figure 3.b, we show the partial layout produced by Algorithm A along with the sets X_i $(0 \leqslant i \leqslant l)$ and X'.

The heart of the proof is contained in the following claim. We shall omit the proof of this claim, since it can be proved by double induction in a similar manner as in [3].

CLAIM 1. *For each $i \leqslant i \leqslant l$, every vertex in X_i is a predecessor of all vertexes in X_{i-1}.*[9] *Moreover, every vertex in X_0 is a predecessor of all vertexes in X'.*

By Claim 1, every vertex in X_i must be assigned before any vertex in X_{i-1} for each $1 \leqslant i \leqslant l$. Similarly, every vertex in X_0 must be assigned before any vertex in X'.

[9] A vertex u is said to be a predecessor of a vertex v if there is a directed path from u to v.

Thus, we have $l+2$ sets of vertexes each of which has to be separated from the others by at least one vertex. Since we have only l vertexes (namely, $W_1, W_2, \ldots, W_l$) that can be used for this purpose, there can be no layout of the graph that satisfies the separation requirement. □

3.2. Forests and arbitrary *k*. In this section we shall give an algorithm for the directed separation problem when the graph is a forest. The algorithm is very similar to Hu's scheduling algorithm[10] for tree-precedence constrained task systems. Before we present our algorithm, we shall first define some terminologies. An *out-tree (in-tree)* is a directed acyclic graph in which every vertex, except one vertex called the *root*, has exactly one immediate predecessor (successor). The root of an out-tree (in-tree) has no immediate predecessor (successor). A *leaf* of an out-tree (in-tree) is a vertex that has no immediate successor (predecessor). The *length of a directed path* is the number of vertexes on the directed path. The *height* of an out-tree (in-tree) is the length of the longest directed path from the root (a leaf) to a leaf (the root). A *forest* is a collection of out-trees or in-trees, but not both.

There are two observations concerning the directed separation problem when the graph is a forest. First, we can restrict our attentions to forests consisting of out-trees only. This is because if we are given a forest consisting of in-trees, then we can convert it to a forest consisting of out-trees by reversing the arrows on the in-trees. If a valid layout exists for the converted forest, then by reading the layout backwards we obtain a valid layout for the original forest. Conversely, if no valid layout exists for the coverted forest, then there can be no valid layout for the original forest. Thus, for the remainder of this section we shall restrict our attentions to forests consisting of out-trees only. For convenience, we shall refer to an out-tree simply as a tree. Secondly, if a forest contains l trees, then there can be no layout of the forest with separation l or more. This is because one of the roots can never be separated from its immediate successors by l or more vertexes.

We are now ready to state our algorithm which is given as Algorithm B below. The algorithm takes as input a positive integer k and a forest. It produces a layout f, whenever one exists, such that $f(v)>f(u)+k$ for all directed edges (u,v) in the forest. Theorem 10 below shows the correctness of Algorithm B.

ALGORITHM B

1. Count the number of roots in the forest. If there are less than $k+1$ roots, report no valid layout exists and terminate the algorithm; otherwise, continue.
2. Choose the roots of the $k+1$ tallest trees and place them in the layout beginning with the root of the tallest tree, followed by the root of the next tallest tree, etc., until all $k+1$ roots have been put in the layout. If two or more trees are of the same height, choose the root of the tree whose immediate predecessor was assigned earliest. Roots without immediate predecessors are chosen last among trees of the same height.
3. Eliminate the $k+1$ roots chosen in step 2, along with all the edges emanating from them, to form a new forest.
4. Repeat steps 1-3 until the algorithm terminates unsuccessfully, or all trees are of height 1 in which case a valid layout can be produced. The remaining roots may now be put in the layout in the order their immediate predecessors were assigned (roots without an immediate predecessor are assigned last). □

THEOREM 10. *Algorithm* B *correctly solves the directed separation problem for forests.*

Proof. The proof of the theorem consists of two parts: (1) showing that the layout produced by Algorithm B is valid whenever the algorithm terminates successfully, and (2) showing that no valid layout can exist whenever the algorithm terminates unsuccessfully. First, suppose Algorithm B terminates successfully. We shall show that every non-leaf vertex is separated by k or more vertexes from its immediate successors in the layout. Take an arbitrary non-leaf vertex and call it y. We know that it was one of the $k+1$ vertexes chosen during a given iteration of the algorithm. Assume it was the lth vertex put in the layout among those $k+1$ vertexes. Since all of the vertexes chosen during an iteration are roots, none of the $k+1-l$ vertexes chosen after y in the given iteration can be immediate successors of it. It can also be seen that the $l-1$ vertexes assigned before y in the given iteration will each have at least one immediate successor that will be used in the next iteration before any immediate successors of y. Thus, there are $k+1-l$ vertexes assigned after y in the iteration in which y was chosen, and at least $l-1$ vertexes assigned in the next iteration before the immediate successors of y are assigned. Therefore, there are at least $(k+1-l)+(l-1)=k$ vertexes between a non-leaf vertex and any of its immediate successors, satisfying the separation requirement.

Conversely, suppose Algorithm B terminates unsuccessfully at the start of the ith iteration. We know that at this point there are less than $k+1$ roots in the forest and at least one tree is of height larger than 1. Let Y be the set of roots of those trees with height larger than 1 at the ith iteration. Since there are less than $k+1$ roots at the ith iteration, at the $(i-1)$st iteration a root was chosen which had no immediate successors (i.e., a tree of height 1). Since roots of taller trees are always assigned first, the only reason why one of the vertexes in Y was not used instead of this vertex was because the immediate predecessors of all vertexes in Y were used in the $(i-1)$st iteration. Continuing this argument, it is easy to show that the vertexes in Y must have had a predecessor used in each of the previous $i-1$ iterations. Since each vertex in Y had a predecessor used in each of the $i-1$ previous iterations and since only $k+1$ vertexes are used in each iteration, the vertexes in Y cannot be assigned at an earlier position than they would be according to Algorithm B. Finally, since there are less than $k+1$ vertexes which could possibly be used at the ith iteration, one of the vertexes in Y cannot be separated from its immediate successors by k or more vertexes. Hence, there can be no layout satisfying the separation requirement. □

3.3. Interval orders and arbitrary *k*. If $G=(V,E)$ is an arbitrary directed acyclic graph, then $G^+=(V,E^+)$ is the transitive closure of G, where $E^+=\{(u,v) \mid u,v,\in V, u\neq v$, and there is a directed path from u to v in $G\}$. Clearly, G^+ corresponds in a natural way to an *irreflexive partial order.*[10] An irreflexive partial order $P=(W,A)$ is called an *interval order* if there is a one-to-one correspondence between W and a set of intervals on a straight line such that $(u,v)\in A$ if and only if the interval corresponding to u is entirely to the left of the interval corresponding to v. Papadimitriou and Yannakakis[13] first considered the class of directed acyclic graphs whose transitive closures are interval orders.[11] They showed that the multiprocessor scheduling problem

[10] An irreflexive partial order $P=(W,A)$ is an ordered pair where W is a set of elements and A is an irreflexive, antisymmetric and transitive relation on W.

[11] It is easy to recognize this class of directed acyclic graphs. It was shown in [13] that this class of graphs is incomparable to forests. We refer the reader to [13] for some of the characterizations of this class of graphs.

is solvable in polynomial time for this class of directed acyclic graphs. We shall show in this section that their algorithm can be adapted to solve the directed separation problem for the same class.

We shall first review Papadimitriou-Yannakakis's scheduling algorithm. Let $G=(V,E)$ be a directed acyclic graph such that its transitive closure $G^+=(V,E^+)$ is an interval order. For each vertex $u \in V$, let $S(u)=\{v \mid (u,v) \in E^+\}$. Papadimitriou-Yannakakis's scheduling algorithm first assigns a label $\alpha(u)$ to each vertex $u \in V$ such that $\alpha(u)>\alpha(v)$ whenever $|S(u)|>|S(v)|$. (We shall call this labeling algorithm the Papadimitriou-Yannakakis's labeling algorithm.) After the labeling is done, the scheduling is done in the same way as Coffman-Graham's scheduling algorithm (see Section 3.1 for details).

The algorithm for the directed separation problem is given as Algorithm C below. The algorithm takes as input a positive integer k and a directed acyclic graph $G=(V,E)$ whose transitive closure $G^+=(V,E^+)$ is an interval order. It produces a layout f, whenever one exists, such that $f(v)>f(u)+k$ for all $(u,v) \in E$.

ALGORITHM C

1. Label all vertexes of G by Papadimitriou-Yannakakis's labeling algorithm.
2. From the list of unassigned vertexes, assign the next "ready" vertex to the layout whose label is the largest among all "ready" vertexes, where a vertex is defined to be "ready" if all of its immediate predecessors are already in the layout and none of its immediate predecessors were among the last k vertexes put in the layout.
3. Repeat step 2 until all vertexes have been assigned, in which case the algorithm has successfully produced a layout, or there are some unassigned vertexes but none of them are ready, in which case the algorithm reports that no such layout is possible. □

THEOREM 11. *Algorithm* C *correctly solves the directed separation problem for graphs whose transitive closures are interval orders.*

Proof. By the nature of the algorithm, if Algorithm C terminates successfully, then the layout produced by the algorithm satisfies the separation requirement. Conversely, if Algorithm C terminates unsuccessfully, then we can show in the same manner as in [13] that no valid layout can possibly exist. We omit the routine details. □

4. Discussions. In this paper we gave complexity results for several variants of the bandwidth minimization problem. We showed that the cycle-bandwidth problem is NP-complete for each fixed $k \geqslant 2$, the separation and cycle-separation problems are both NP-complete for each fixed $k \geqslant 1$, and the directed separation problem is NP-complete for arbitrary k. We gave reductions from the multiprocessor scheduling problem and the MSIRD problem to the directed separation problem, and reductions from the directed separation problem to the MSRRD and MSUPS problems. Finally, we gave polynomial time algorithms for several special cases of the directed separation problem. It should be noted that the algorithms given in Section 3 can be modified to solve a related problem in which we are asked to find the minimum number of isolated vertexes needed to add to the graph G so that it has a layout with separation k or more.

There are several open questions which we think are worthwhile to pursue for future research. The first and the most important one is the complexity of the directed

separation problem for each fixed $k \geqslant 2$. An answer to this question will solve several open problems in deterministic scheduling theory. If the directed separation problem were shown to be solvable in polynomial time for each fixed $k \geqslant 2$, then the multiprocessor scheduling problem and the MSIRD problem are solvable in polynomial time for each fixed $k \geqslant 3$, answering the open questions posed in [5,15]. On the other hand, if the directed separation problem were shown to be NP-complete for each fixed $k \geqslant 2$, then the MSUPS problem is NP-complete for each fixed $k \geqslant 4$ and the MSRRD problem is NP-complete for each fixed $k \geqslant 3$, answering an open question posed in [7]. On a less fundamental level, is it possible to reduce the directed separation problem to the multiprocessor scheduling problem and the MSIRD problem (thus showing they are equivalent), or reduce the MSUPS and MSRRD problems to the directed separation problem? As we noted in section 1, the directed separation problem appears to be not much harder than the multiprocessor scheduling problem. It is conceivable that such reductions exist.

Secondly, how hard is it to find an approximate solution with a constant worst-case bound to the problem of finding a layout that minimizes the circular-bandwidth or the bandwidth? As we noted in section 1, an approximation algorithm with a constant worst-case performance bound for one problem gives an approximation algorithm for the other. Thus, an answer to one question gives an answer to the other. Finally, our generalization of linear layouts to circular layouts can be applied to other graph layout problems such as the cutwidth minimization problem[6],[9] and the optimal linear arrangement problem[6],[8]. What are the complexities of these problems for circular layouts?

Acknowledgment. I. Hal Sudborough provided the proof of Theorem 2 and the relationships between the bandwidth and cycle-bandwidth of a graph.

REFERENCES

[1] A. V. AHO, M. R. GAREY, AND J. D. ULLMAN, *The Transitive Reduction of a Directed Graph,* SIAM J. Comput., 1 (1972), 131-137.

[2] E. G. COFFMAN, JR. AND P. J. DENNING, *Operating Systems Theory,* Prentice-Hall, Englewood Cliffs, NJ, 1973.

[3] E. G. COFFMAN, JR. AND R. L. GRAHAM, *Optimal Scheduling for Two-Processor Systems,* Acta Informat., 1 (1972), 200-213.

[4] M. R. GAREY, R. L. GRAHAM, D. S. JOHNSON, AND D. E. KNUTH, *Complexity Results for Bandwidth Minimization,* SIAM J. Appl. Math., 34 (1978), 477-495.

[5] M. R. GAREY AND D. S. JOHNSON, *Two-Processor Scheduling With Start-Times and Deadlines,* SIAM J. Comput., 6 (1977), 416-426.

[6] ———, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman, San Francisco, CA, 1979.

[7] M. R. GAREY, D. S. JOHNSON, B. B. SIMONS, AND R. E. TARJAN, *Scheduling Unit-Time Tasks with Arbitrary Release Times and Deadlines,* SIAM J. Comput., 10 (1981), 256-269.

[8] M. R. GAREY, D. S. JOHNSON, AND L. STOCKMEYER, *Some Simplified NP-Complete Graph Problems,* Theoretical Computer Science, 1 (1976), 237-267.

[9] F. GAVRIL, *Some NP-Complete Problems on Graphs,* Proc. 11th Conf. on Information Sciences and Systems, Johns Hopkins University, Baltimore, MD, 1977, 91-95.

[10] T. C. HU, *Parallel Sequencing and Assembly Line Problems,* Operations Res., 9 (1961) 841-848.

[11] B. MONIEN AND I. H. SUDBOROUGH, *Bandwidth Problems in Graphs,* Proc. 18th Annual Allerton Conference on Communication, Control, and Computing, University of Illinois, Urbana-Champaign, IL, 1980, 650-659.

[12] C. H. PAPADIMITRIOU, *The NP-Completeness of the Bandwidth Minimization Problem,* Computing, 16 (1976), 177-192.

[13] C. H. PAPADIMITRIOU AND M. YANNAKAKIS, *Scheduling Interval-Ordered Tasks,* SIAM J. Comput., 8 (1979), 405-409.

[14] J. B. SAXE, *Dynamic-Programming Algorithms for Recognizing Small-Bandwidth Graphs in Polynomial Time,* Proc. 17th Annual Allerton Conference on Communication, Control, and Computing, University of Illinois, Urbana-Champaign, IL, 1979, 499-508.

[15] J. D. ULLMAN, *NP-Complete Scheduling Problems,* J. Comput. Systems Sci., 10 (1975), 384-393.

SIAM J. COMPUT.
Vol. 13, No. 4, November 1984
© 1984 Society for Industrial and Applied Mathematics
001

DISK PERFORMANCE IN A TRANSACTION-ORIENTED SYSTEM*

D. P. HEYMAN† AND S. TSUR‡

Abstract. In this paper we address the performance issues that arise as a result of the two level scheduling in a database/operating system. At the upper (database) level *transactions* are *logically scheduled* so as to maintain the database integrity. At the lower (operating system) level *physical disk requests* are scheduled. We consider the performance problems that result from the interplay between these two levels of scheduling. Our model assumes the existence of a dictionary that must be consulted prior to the execution of a transaction. We derive the optimal disk placement of this dictionary and our results show that the waiting time for dictionary look-up is the critical component in the transaction response-time for a wide range of transaction sizes. To alleviate this problem we suggest alternative approaches to dictionary placement in such systems.

Key words. database, transaction scheduling, disk scheduling

1. Introduction. Many contemporary database applications are centered around the concept of a *transaction.* A transaction is a sequence of operations on the data of a database which, from the users' point of view, must be executed as one indivisible unit. From a system perspective, the objective is to preserve the *database integrity*[1] in a concurrent-user environment. The database component entrusted with this task is the *transaction manager.*

The transaction manager must schedule the various transactions in a way that is consistent with the integrity preservation requirement. The most common way is to produce a *serializable schedule* [5]—one that is equivalent to a schedule in which each of the transactions is executed in a non-overlapping fashion without interference from other co-resident transactions.

To produce a schedule, the transaction manager consults a *dictionary*—a repository of descriptive data which is kept with the database. The information gleaned from this dictionary lookup is checked for potential read/write conflicts between transactions and translated into information about access to the participating entities (data elements) of the transactions to be scheduled. Consequently, at the transaction scheduling level,

1. the scheduling of each transaction is preceded by dictionary lookup, and
2. as a result of this lookup all participating entities are known.[2]

The transaction schedule is passed to the underlying operating system for execution. At this level a standard disk scheduling algorithm such as SCAN [3] is employed. Disk scheduling at this level does not utilize any information specific to the database environment.

This paper addresses the performance issues that arise as a result of the interplay between the two levels of scheduling that we described: the logical, transaction-oriented level at the database and the physical, disk oriented level at the operating system. We will see that the requirement that a dictionary lookup precedes the entity-access strongly influences the order in which the transactions are processed.

* Received by the editors November 30, 1982, and in final revised form June 14, 1983.
† Bell Communications Research, Holmdel, New Jersey 07733.
‡ IBM Scientific Center, Haifa, Israel.

[1] A set of conditions on the stored data that are determined by the semantics of the database application—e.g., AGE > 0 in a database containing personnel data. These conditions must hold irrespective of the operations that are performed on the database.

[2] Those cases in which the accessed entities cannot be known prior to execution because of data-dependency are excluded from this discussion.

The model presented in this paper, and its analysis, build on the analysis of the SCAN rule by Coffman, Klimko and Ryan [1] and recently by Coffman and Hofri [2]. Their model as well as other published work on disk scheduling [4], [6], [8] concentrates on the lower, operating-system level. Our work here differs in the following respects:

1. In the published literature, the basic work-unit to be scheduled is a single disk-access whereas we consider a transaction.

2. The performance measure that we optimize is the response-time of individual transactions.

3. The operating constraints are different in that the imposition of a dictionary limits the randomness of access to the data. Therefore, the use of a dictionary should not be regarded as a device to improve the performance of the system, but rather as a "necessary evil" with which the system must work.

Section 2 describes the transaction model and the assumptions that we have made with respect to the operating system environment. The performance of the model is analyzed in § 3 and some numerical results and their interpretation are presented in § 4. In concluding this paper we suggest some extensions to our work in § 5.

2. The transaction model and its environment. The transaction arrival process in the system is assumed to be Poisson with mean λ. Each transaction is of the form $[e_0, e_1, e_2, \cdots, e_n]$ where e_0 is a dictionary entry and $e_1, \cdots, e_n$ are the related data entities that must be fetched.[3]

We assume that N, the number of entities fetched for a transaction, is a random variable with a known distribution. The dictionary and data entities reside on a moving-arm disk (Fig. 1).

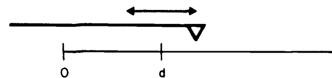


FIG. 1. *A simple disk model.*

We approximate the discrete track-set of the disk by the continuous interval [0, 1] and assume that the dictionary occupies one cylinder or a part thereof which is positioned at a distance d from the leftmost position of the arm. The disk arm moves at speed a (time/cylinder) over the disk surface and the database entities are distributed over the surface locations with a known density $f(\cdot)$. The arm movement is governed by a simple scan algorithm which moves it cyclically from 0 to 1 and back to 0. During this cycle the arm stops on those tracks where it reads either data or the dictionary. The time (rotation + transfer) to read/write an entity is the constant τ and the time to read the dictionary entry at position d for a transaction is the constant δ. These are the times for block-reading and consequently, δ is independent of the transaction-size.

The transaction service process in the system is as follows. During a cycle the arm visits the dictionary twice: once when it moves right from 0 to 1 and once when it moves left from 1 to 0. During each visit the dictionary is read and the transactions that have arrived *since the last visit* are available to be processed. That is, the positions of their entities are obtained through the dictionary lookup and they are fetched during the subsequent part of the cycle when the arm passes over them. A complete cycle of this process is show in Fig. 2. Note that in this regime,

[3] The same model is also applicable for indexed file access. In this case, the dictionary consists of the index to the file and the entities are the data pages accessed through the index lookup.

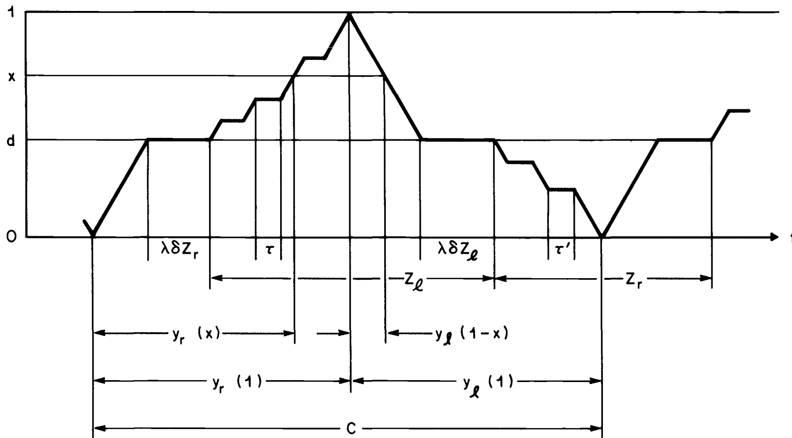


FIG. 2. *A transaction processing cycle.* $\lambda\delta Z_r$, $\lambda\delta Z_l$ *dictionary reading phases*; τ, τ' *typical entity reading/writing phases (all quantities depicted are expected values).*

1. the servicing of separate transactions may be interleaved and is governed by the movement of the arm and the positions of the data entities, and

2. entities for a given transaction may be fetched in an order which may be different from their logical order in the transaction specification. We assume that the system employs an appropriate buffering system in which the entities can be "reconstructed" before unlocking the transaction.

In § 4 we analyze the question of the *optimal placement of the dictionary* in this system, i.e., what is the optimal value of d when the objective is to minimize the *transaction response-time* for an individual transaction in the system. Note that this objective is different from the published work on disk performance analysis in that the usual measure of performance is to maximize the service rate *per task* where a task is the processing of either a dictionary entry or an entity.

3. Analysis of the model. In this section we will describe our model in detail and solve for the mean delay of a transaction in the steady state. For completeness we list here the notation that we will use in this analysis.

δ = time to read a dictionary record (a constant),

τ = time to read an entity (a constant),

p_n = probability that a transaction contains n entities to be fetched,

$\nu = \sum_1^\infty np_n$ = mean number of entities to be fetched in a transaction,

$F(x) = \int_0^x f(y)\,dy$ = probability an entity is located in $[0, x]$,

$F^c(x) = 1 - F(x)$,

λ = arrival rate of transactions,

d = location of the dictionary,

a = disk arm speed (time/cylinder).

3.1. Equilibrium conditions. By a *cycle* we mean an interval of time in which the head starts at d and is traveling left (i.e., towards 0) and ends at the next such instant. That is, it is an interval in which the arm moves from d to 0 to 1 to d. Let c be the

expected length of a cycle in the steady state. A simple algebraic derivation of a formula for c will emerge from our subsequent analysis, but the following heuristic derivation will lead to a condition for the existence of the steady state.

In every cycle, $2a$ time units are spent moving the head. In the steady state, the average number of transactions that arrive is λc, so λc must be the mean number of transactions that are processed. Each transaction requires one dictionary look-up and an average of ν entity transfers, so $\delta + \nu\tau$ is the mean time to process each transaction. Therefore,

$$c = 2a + \lambda(\delta + \nu\tau)c = \frac{2a}{1 - \lambda(\delta + \nu\tau)}. \tag{1}$$

Since $0 < c < \infty$ is required, we must have

$$\lambda(\delta + \nu\tau) < 1. \tag{2}$$

Since $1/\lambda$ is the mean time between transaction arrivals, (2) asserts that the mean interarrival time is larger than the mean processing time. By using the theory of regenerative processes (see e.g., [7, § 6.4]), it can be shown that (2) is sufficient for the existence of the steady state. We will not pursue that issue here.

For the remainder of this section we assume that (2) is valid and that the system is in the steady state.

3.2. Head travel times. Let $y_r(x)$ be the expected time for the head to travel from 0 to x and $y_l(1-x)$ be the expected time for the head to travel from 1 to x (a distance of $1-x$). These are the functions we will obtain in this subsection.

Since all entities that are in $[0, d]$ are fetched while traveling left from d to 0,

$$y_r(x) = ax, \qquad x < d. \tag{3a}$$

When the head reaches d, all the transactions that arrived since the last visit to d receive a dictionary look-up. When the head is traveling to the right, let Z_r be the expected value of the length of time since the head was last at d plus the current length of time spent at d. Then $\lambda Z_r \delta$ is the expected amount of time needed to read the dictionary. Note that we assume that a transaction which arrives when the head is at d uses the dictionary at that time. Thus, Z_r satisfies

$$Z_r = y_l(1) - y_l(1-d) + ad + \lambda\delta Z_r \tag{4}$$

and so

$$y_r(d) = ad + \lambda\delta Z_r. \tag{3b}$$

When the head is traveling to the right, those entities it can read in $(d, 1]$ must have arrived during the last interval of time in which the head traveled from d to 1 to 0 to d. It is easy to see that the expectation of the length of this interval is the expected cycle length c. Since $F(x) - F(d)$ is the probability that an entity resides in $(d, x]$, $\lambda c[F(x) - F(d)]\nu\tau$ is the expected amount of time spent reading entities as the head travels from d to x, $x > d$, so

$$y_r(x) = ax + \lambda\delta Z_r + \lambda c[F(x) - F(d)]\nu\tau, \qquad x > d. \tag{3c}$$

When the head is traveling left, equations for $y_l(\cdot)$ are obtained from (3) by symmetry. From (3a) we obtain:

$$y_l(x) = ax, \qquad x < 1 - d. \tag{5a}$$

When the head reaches d, let Z_l be the expected value of the length of time since the head was last at d. Adapting the derivation of (4) yields

$$Z_l = y_r(1) - y_r(d) + a(1-d) + \lambda\delta Z_l. \tag{6}$$

Adapting the arguments leading to (3b) and (3c) yields

$$y_l(1-d) = a(1-d) + \lambda\delta Z_l \tag{5b}$$

and

$$y_l(x) = ax + \lambda\delta Z_l + \lambda c[F(d) - F(1-x)]\nu\tau, \qquad x > 1-d. \tag{5c}$$

In order to solve (4) and (6) for Z_r and Z_l, we use (3) and (5). From (5b) and (5c),

$$y_l(1) - y_l(1-d) = ad + \tau\nu\lambda cF(d); \tag{7}$$

substituting (7) into (4) yields

$$Z_r = [2ad + \tau\nu\lambda cF(d)]/(1-\lambda\delta). \tag{8}$$

Similarly, we obtain

$$Z_l = \{2a(1-d) + \tau\nu\lambda c[1-F(d)]\}/(1-\lambda\delta). \tag{9}$$

Observe that $c = Z_r + Z_l$, which with (8) and (9) yields (1) and provides a partial check of our results.

3.3. Waiting times. In this section we provide equations for calculating the expected wait that is incurred per transaction. By the wait of a transaction we mean the length of the time interval that starts when the transaction arrives and ends when all the entities of the transaction have been fetched.

Let the generic random variable D denote the length of the time interval that starts when a transaction arrives and ends at the very next completion of a dictionary look-up. For the same transaction, let the generic random variable S denote the time to fetch the entities. The expected waiting time of the transaction is $E(D)+E(S)$. We will obtain $E(D)$ and then $E(S)$. The first step is to describe the position of the head.

Let $G_r(x)$ be the probability that the head position is between 0 and x and that the head is traveling right at an arbitrary epoch. Since the cycle lengths are i.i.d. random variables, regenerative-reward arguments (see, e.g., [7, Thm. 6–8]) show that

$$G_r(x) = y_r(x)/c. \tag{10}$$

For $x \neq d$, we may write $g_r(x) = G_r'(x) = y_r'(x)/c$, where a prime denotes differentiation. For notational convenience, we will write

$$g_r(d)\,dx = [G_r(d) - G_r(d^-)] = \lambda\delta Z_r/c,$$

which is the probability that the head is at d and traveling right.

Similarly,

$$\begin{aligned} G_l(x) &= P\,\{\text{head position} \leqq x \text{ and moving left}\} \\ &= [y_l(1) - y_l(1-x)]/c \end{aligned} \tag{11}$$

and so

$$g_l(x) = y_l'(1-x)/c, \qquad x \neq d$$

with the convention

$$g_l(d)\,dx = \lambda\delta Z_l/c.$$

From (10) and (3c) we obtain

(12) $P\{\text{head traveling to the right}\} = y_r(1)/c = (a+\lambda\delta Z_r)/c + \tau\nu\lambda[1-F(d)].$

Similarly, (11) and (5c) yield

(13) $$P\{\text{head traveling to the left}\} = (a+\lambda\delta Z_l)/c + \tau\nu\lambda F(d).$$

The independent-increments property of a Poisson process implies that the condition "an event occurs at t" does not affect the distribution of the events which may occur before or after t. For example, if it is given that a transaction arrives when the head is at position $x<d$ and traveling to the right, the mean time until the head reaches $x+\varepsilon<d$ is $y_r(x+\varepsilon)-y_r(x)$. The reason for this is that the amount of time required to move from x to $x+\varepsilon$ depends only on those transactions which arrived prior to the last visit to the dictionary, and they are completely independent of the event "a transaction has just arrived at position x." A formal proof of these assertions is given in a more general setting in [9].

When we trace the progress of the transactions in the arrival process we note that they all have effects first when the dictionary is read, and secondly, when their corresponding entities are fetched. We will select one particular transaction and call it the *tagged transaction.* The dictionary processing time for the tagged transaction is δ, and during this time an average of $\lambda\delta$ new transactions arrive which require processing time, during which new transactions arrive and so forth. Let A be the amount of additional time spent in dictionary reading due to the arrival of the tagged transaction (this time is conditional on the arrival of the tagged transaction); then

(14) $$E(A) = \delta + \lambda\delta E(A) = \delta/(1-\lambda\delta)$$

where (2) ensures that $E(A)<\infty$. Let α be the expected number of transactions that arrived during an interval of length A; then

(15) $$\alpha = \lambda E(A) = \lambda\delta/(1-\lambda\delta).$$

To obtain $E(D)$ we consider six cases. For brevity we use the symbol ☛ (resp. ☚) to indicate that the head is moving right (resp. left).

Case 1. The transaction arrives when the head is in position $x<d$ and traveling right. Here, D corresponds to a trip from x to d plus A, so

(16a) $$E(D|x<d, \text{☛}) = y_r(d) - y_r(x) = a(d-x) + \lambda\delta Z_r + E(A).$$

Case 2. The transaction arrives when the head is at the dictionary and traveling right. In this case, D equals A plus the equilibrium excess (or forward recurrence-time) distribution of a renewal process where the renewal lengths correspond to the lengths of visits to the dictionary. We know that the mean of this distribution depends on the first and second moments of the renewal lengths. We have not been able to obtain the required second moment, so we will make the approximation

(16b) $$E(D|x=d, \text{☛}) \doteq \lambda\delta Z_r + E(A).$$

This approximation will be good when $E(\Delta^2)$ is close to $2[E(\Delta)]^2$ where Δ is the length

of a visit to the dictionary. If the probability that the head is at d is small, the inaccuracy of this approximation will not be significant. We will return to this point below.

Case 3. The transaction arrives when the head is at position $x > d$ and traveling right. Here, D is a trip from x to 1 to d plus A, so

$$\text{(16c)} \quad \begin{aligned} E(D|x > d, \rightarrow) &= y_r(1) - y_r(x) + y_l(1-d) + E(A) \\ &= a(2-x-d) + \tau\nu\lambda c[1-F(x)] + \lambda\delta Z_l + E(A). \end{aligned}$$

Case 4. The transaction arrives when the head is at position $x > d$ and traveling left. Here, D is a trip from x to d plus A, so

$$\text{(16d)} \quad E(D|x > d, \leftarrow) = y_l(1-d) - y_l(1-x) + E(A) = a(x-d) + \lambda\delta Z_l + E(A).$$

Case 5. The transaction arrives when the head is at position d and traveling left. This is similar to Case 2, and we make the approximation,

$$\text{(16e)} \quad E(D|x = d, \leftarrow) \doteq \lambda\delta Z_l + E(A).$$

Case 6. The transaction arrives when the head is at position $x < d$ and traveling left. Here, D is a trip from x to d plus A, so

$$\text{(16f)} \quad \begin{aligned} E(D|x < d, \leftarrow) &= y_l(1) - y_l(1-x) + y_r(d) + E(A) \\ &= a(x+d) + \tau\nu\lambda c F(x) + \lambda\delta Z_r + E(A). \end{aligned}$$

Removing the conditions yields

$$\text{(17)} \quad E(D) = \int_0^1 E(D|x, \rightarrow) g_r(x)\, dx + \int_0^1 E(D|x, \leftarrow) g_l(x)\, dx.$$

Let ξ be the probability that the head is at d. The expected amount of time in a cycle that the head is at d is $\lambda\delta(Z_r + Z_l) = \lambda\delta c$, so

$$\xi = (\lambda\delta c)/c = \lambda\delta.$$

When ξ is small, the approximation errors in (16b) and (16e) should not significantly affect (17). In the examples presented in § 4, $\delta = \tau = 18$ msec., and $a = 100$ msec. Only in case c of Fig. 3 does ξ exceed 0.08 (in case c, $\xi = 0.45$).

Now we will obtain $E(S)$. The first step is to determine the location distribution of the entities in a transaction. Recall that the random variable N denotes the number of entities in a transaction and $p_n = P\{N = n\}$. When $N = n$, the probability that all the entities are to the left of y is $[F(y)]^n$. Therefore, the probability that the largest entity location, denoted by L, is no bigger than y, $L(y)$ say, is

$$\text{(18)} \quad L(y) = \sum_1^{\infty} [F(y)]^n P\{N = n\} = \hat{P}[F(y)]$$

where $\hat{P}(z) = \sum_1^{\infty} z^n p_n$. Let $\hat{p}(\cdot)$ be the derivative of $\hat{P}(\cdot)$ and $L'(\cdot)$ be the derivative of $L(\cdot)$; then

$$\text{(18a)} \quad L'(y) = \hat{p}[F(y)] f(y)$$

is the density function for the location of the rightmost entity location.

The probability that not all of the entities are to the right of y, when $N=n$, is $1-[F^c(y)]^n$. Let M be the minimum location of all the entities. The probability that M is no bigger than y, $M(y)$ say, is

$$M(y)=\sum_1^\infty (1-[F^c(y)]^n)P\{N=1\}=1-\hat{P}[F^c(y)]. \tag{19}$$

The corresponding density function is

$$m(y)=\hat{p}[F^c(y)]f(y). \tag{19a}$$

To obtain $E(S)$, we observe that every transaction requires, on average, an amount $\nu\tau$ of time to do the data transfer on the entities. The remaining component of $E(S)$ is the expected travel time to fetch the entities, which we denote by $E(T)$. To obtain $E(T)$ we examine four cases. We will use the notation $E(T_i)$ for $E(T)$ in case i. In each case we divide T into two parts which we call T^* and T^{**}. The former is the travel time that would have occurred had we not conditioned on the presence of the tagged transaction and the latter is the travel time generated by the presence of the tagged transaction. This dichotomy is justified by the independent increments property of a Poisson process, as described after (13).

Case 1. The head is traveling right and $d<M$. In this case T^* is the length of a trip from d to L, and so

$$E(T^*|L=x>d, ☛)=y_r(x)-y_r(d)=a(x-d)+\tau\nu\lambda c[F(x)-F(d)].$$

Since $[F(x)-F(d)]\nu$ is the mean number of entities located between d and x for an arbitrary transaction, and α is the expected number of additional transactions present due to the tagged transaction,

$$E(T^{**}|L=x>d, ☛)=[F(x)-F(d)]\nu\tau\alpha.$$

Let π_r be the probability that the head is traveling right when it leaves the dictionary. Since we condition on the event that the tagged transaction has arrived, π_r is not $1/2$. Equivalently, this is the probability that a transaction arrives when the head is to the left of the dictionary. Then

$$\pi_r=G_r(d^+)+G_l(d^-);$$

using (3b), (7), (8), (10) and (11) we obtain

$$\pi_r=[2ad+\tau\nu\lambda cF(d)]/c(1-\lambda\delta). \tag{20}$$

We have

$$P\{L\leqq x \,\&\, d<M\}=\sum_{n=1}^\infty [F(x)-F(d)]^n p_n=\hat{P}[F(x)-F(d)],$$

so

$$E(T_1)=\pi_r\int_d^1 [E(T^*|L=x, ☛)+E(T^{**}|L=x, ☛)]\hat{p}[F(x)-F(d)]f(x)\,dx. \tag{21a}$$

Case 2. The head is traveling right and $M<d$. When $M=x$, T^* is the length of a trip from d to 1 to x for every value of L, so

$$E(T^*|M=x<d, ☛)=y_r(1)-y_r(d)+y_l(1-x)=a(2-d-x)+\lambda\delta Z_l+\tau\nu\lambda cF^c(x).$$

We analyze T^{**} by dividing it into three parts: the effects of transactions that arrive

(i) during the dictionary look-up of the tagged transaction, (ii) while the entities of the transactions in (i) are read, and (iii) while the entities in the tagged transaction itself are read. The steps required to do this analysis are relegated to the Appendix. The result is

$$E(T^{**}|M=x, \rightarrow) = \alpha\tau\nu F^c(x) + \lambda\psi\beta + \beta\lambda\tau\nu_M,$$

where ν_M is the conditional expectation of the number of entities in a transaction which are to the right of the dictionary when $M = x < d$, and ψ, β and ν_M are given in equations (A.1), (A.2) and (A.3) in the Appendix.

Thus, we have

$$E(T_2) = \pi_r \int_0^d [E(T^*|M=x, \rightarrow) + E(T^{**}|M=x, \rightarrow)]m(x)\, dx. \tag{21b}$$

Case 3. The head is traveling left and $L < d$. In this case, T^* is the length of a trip from d to M so

$$E(T^*|M=x, L<d, \leftarrow) = y_l(1-x) - y_l(1-d) = a(d-x) + \tau\nu\lambda c[F(d) - F(x)].$$

The analysis of T^{**} parallels the analysis in case 1; we obtain

$$E(T^{**}|M=x, L<d, \leftarrow) = [F(d) - F(x)]\nu\tau\alpha.$$

We have

$$P\{M > x \& L < d\} = P\{\text{all entities in } [x, d)\} = \sum_1^\infty [F(d) - F(x)]^n p_n = \hat{P}[F(d) - F(x)].$$

The corresponding density function is

$$-\frac{d}{dx}\hat{P}[F(d) - F(x)] = \hat{p}[F(d) - F(x)]f(x),$$

which, with $\pi_l = 1 - \pi_r$, yields

$$\begin{aligned} E(T_3) = \pi_l \int_0^d & [E(T^*|M=x, L<d, \leftarrow) \\ & + E(T^{**}|M=x, L<d, \leftarrow)]\hat{p}[F(d) - F(x)]f(x)\, dx. \end{aligned} \tag{21c}$$

Case 4. The head is traveling left and $L > d$. In this case, T^* is the length of a trip from d to 0 to L, and so

$$E(T^*|L=x>d, \leftarrow) = y_l(1) - y_l(1-d) + y_r(x) = a(d+x) + \lambda\delta Z_r + \tau\nu\lambda cF(x).$$

The analysis of T^{**} parallels the analysis in case 2 with $F(\cdot)$ replacing $F^c(\cdot)$. The result is

$$E(T^{**}|M=x) = \alpha\tau\nu F(x) + \lambda\psi\beta + \beta\lambda\tau\nu_L$$

where ν_L is the conditional expectation of the number of entities in a transaction which are to the left of the dictionary when $L = x > d$, and ψ, β and ν_L are given in equations (A.4), (A.5) and (A.6) in the Appendix.

Thus we have

$$E(T_4) = \pi_l \int_d^1 [E(T^*|M=x) + E(T^{**}|M=x)]m(x)\, dx. \tag{21d}$$

Cases 1 through 4 are mutually exclusive and collectively exhaustive, so

$$E(T)=\sum_1^4 E(T_i). \tag{21e}$$

Thus,

$$E(S)=\nu\tau+E(T) \tag{22}$$

with $E(T)$ computed from (21e).

4. Numerical results and discussion. In this section we present some of the numerical results that we have obtained with the model for specific distributions of entity locations and the size of transactions. In our calculations we have used the beta distribution with integer parameters for entity locations, i.e.,

$$f(x)=\frac{(\gamma_1+\gamma_2+1)!}{\gamma_1!\,\gamma_2!}x^{\gamma_1}(1-x)^{\gamma_2} \qquad 0<x<1 \tag{23}$$

for parameters γ_1, $\gamma_2>-1$. We have chosen this distribution because of its unimodality and the convenience with which we can control its mean by suitable parameter choice. In our experiments we have used the parameter values $\gamma_1=4$ and $\gamma_2=2$ which place the mean at $x=0.625$. Note that this distribution reduces to the uniform distribution when $\gamma_1=\gamma_2=0$. We have used the geometric distribution for the transaction size (number of entities); i.e.,

$$p_n=p(1-p)^{n-1} \qquad n>0 \tag{24}$$

for a given parameter p. The average transaction size for this distribution is

$$\nu=\sum_{n=1}^{\infty} np_n=1/p. \tag{25}$$

In § 3 we analyzed two factors that determine the system's performance: $E(D)$, the expected time from transaction arrival until the next dictionary visit completion, and $E(S)$, the expected time to read the related transaction entities as well as the entities of other co-existing transactions after the visit. Fig. 3 depicts these quantities as functions of the dictionary location d for four different transaction loads. The transaction-load parameters that we have varied in these cases are λ, the transaction arrival rate, and ν, the average transaction size. In all cases we have kept the scan-speed a as well as τ and δ fixed. From (2), $\lambda_{\max}=(\delta+\nu\tau)^{-1}$ is the upper bound on the arrival rates for which the steady-state exists. In cases (a) and (b), the average cycle time is 202 ms; it is 2000 ms. in cases (c) and (d).

From the results we observe two conflicting trends: The optimal placement is *at the mean* of the data distribution as far as the expected time to reach and read the dictionary, $E(D)$, is concerned. This can be explained by observing that this time is dominated by the time it takes to read the data entities on the way to the dictionary. Therefore, these times become balanced if d is at the mean. On the other hand, the entity read-times *after* dictionary lookup are minimal if the data can be collected in one sweep over the disk surface without having to reverse directions (and, consequently, first visit the dictionary again). Therefore from this aspect the optimal d is at that extreme point which is closest to the mean. In the lightly loaded cases the performance is insensitive to the actual dictionary placement whereas in the heavily loaded cases placement becomes critical and should be at the mean of the data.

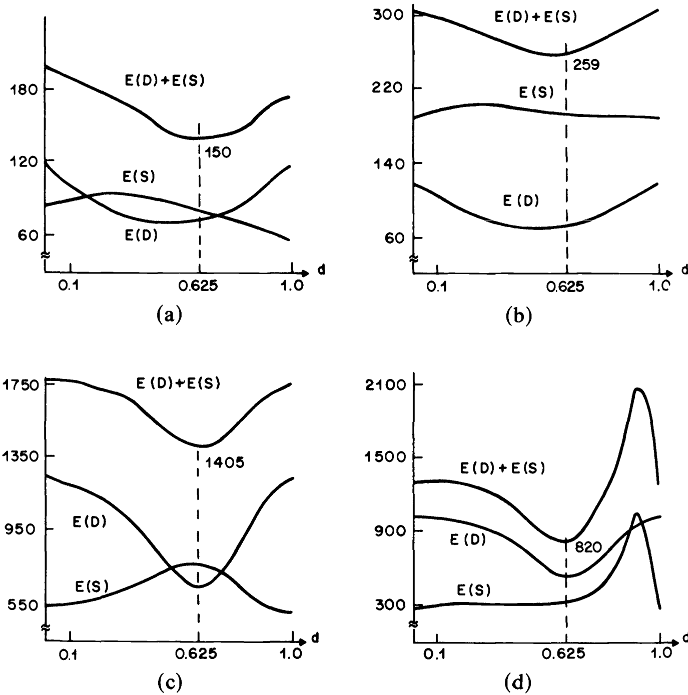


FIG. 3. *Expected transaction waiting times as a function of dictionary location for different transaction-load types.* (a) $\lambda = 0.01\lambda_{\max}$, $\nu = 1$. (b) $\lambda = 0.01\lambda_{\max}$, $\nu = 10$. (c) $\lambda = 0.9\lambda_{\max}$, $\nu = 1$. (d) $\lambda = 0.9\lambda_{\max}$, $\nu = 10$. *In all cases,* $\tau = \delta = 18$ ms, $a = 2$ ms, $\gamma_1 = 4$ *and* $\gamma_2 = 2$.

In all four cases, the largest component of $E(D)$ is due to either (16c) or (16f), according to whether the dictionary is to the left or right of the center of the disk surface. These components include a phase where the arm moves from an extreme to the dictionary without doing any useful work. In cases (a) and (b) which have light loads, these components account for about 70 percent of $E(D)$. In case (c) which has a heavy load of small transactions, these components account for about 60 percent of $E(D)$. The most dramatic case is (d), where these components account for 97 percent of $E(D)$.

We note also the difference in performance between a traffic process of small transactions and one of large transactions. For a transaction load of small transactions the maximal arrival rate for which the system remains stable is much higher than the maximal load of large transactions. Since each new transaction requires a dictionary lookup, the lookup phase becomes a major performance barrier with respect to completion time in heavy traffic streams of small transactions—much more so than in the heavy traffic situations of large transactions.

5. Conclusion. In this paper we have extended the results of disk scheduling in an operating system to include transaction scheduling in a database system. Our main conclusions are that the dictionary placement has a critical influence on the performance, particularly with heavy transaction traffic, and that the use of a dictionary carries a significant overhead compared with systems that can function without it.

It might seem to be easy to relax the assumption that the read/write and dictionary look-up times for each entity are constants. If this were done, the analysis would have to deal with length-biased sampling, as mentioned in the development of (16b). We

have attempted to analytically obtain the optimal placement of the dictionary by differentiating the formula for the expected wait-per-transaction with respect to the dictionary parameter. The complexity of the formula prevented us from completing this task. Perhaps a simpler model can be devised that lends itself to analytic optimization and also preserves the essential dynamics of the system.

We noted in the introduction that the use of a dictionary is a constraint which stems from the functional requirements of the transaction system and therefore, assuming that we cannot dispense with it, we can ask whether the overheads incurred by its use can be reduced. An examination of the transaction processing cycle in Fig. 2 reveals that it contains 2 phases in which the arm moves "blindly" from the extremes to the dictionary without doing any useful work. Intuitively, if these phases could be shortened, the overhead incurred in the dictionary lookup would be reduced. To achieve this goal we could use *multiple dictionaries*; e.g., we could place one dictionary at each extreme of the disk. The result would be that the blind phases would disappear altogether and that the dictionary lookup times would be twice as frequent and therefore shorter.

The main pitfall of such a scheme is that the data must now be distributed: either in a partitioned or a replicated form. Dictionary updating would be more complex in this environment. We suggest this extension as well as other performance problems that would arise from the use of multiple dictionaries as a topic for further research.

Appendix. In this Appendix we give the details of the derivation of $E(T^{**})$ in cases 2 and 4. Recall that for case 2, we analyze T^{**} by dividing it into three parts: the effects of transactions that arrive (i) during the dictionary look-up of the tagged transaction, (ii) while the entities in the transactions in (i) are read, and (iii) while the entities in the tagged transactions itself are read.

(i) For each additional transaction that arrived during the dictionary look-up of the tagged transaction, we read the entities to the right of x. For each transaction, the mean number of such entities is $\nu F^c(x)$, so the mean time to read all the entities is

$$E(T_i^{**}) = \alpha\tau\nu F^c(x).$$

(ii) For all transactions together that are described in (i), the mean time spent reading entities in $(d, 1]$ is

$$\psi = \alpha F^c(d)\nu\tau. \tag{A.1}$$

The mean number of transactions that arrive while this is happening is $\lambda\psi$. Let β be the mean contribution to T^{**} from each of these transactions. Then

$$\beta = E(A) + (\alpha+1)[F(d) - F(x)]\nu\tau. \tag{A.2}$$

The first term reflects the time spent at the dictionary and the second term reflects the time spent reading entities in $[x, d)$. The "one" in the $(\alpha+1)$ term is the effect of the transaction that arrived when the head was in $(d, 1]$. Thus,

$$E(T_{ii}^{**}) = \lambda\psi\beta.$$

(iii) While the entities of the tagged transaction are read, new arrivals may occur which will be felt at the subsequent visit to the dictionary. For each of these arrivals, the tagged transaction will wait for a dictionary look-up as well as for those entities that are in $[M, d)$. For the tagged transaction, let K be the number of entities in $(d, 1]$. Then $\lambda\tau E(K|M=x)$ is the mean number of transactions that arrive while these K

entities are being read. The mean time spent on each of these transactions is β, so

$$E(T_{iii}^{**}) = \beta\lambda\tau E(K|M=x).$$

The formula for $E(K|M=x)$ is obtained as follows. When $N=n$ and $M=x<d$, there is one point at x and $(n-1)$ points that can be placed in $[d,1]$. The probability that a point is placed in $[d,1]$ when $M=x$ is $F^c(d)/F^c(x)$ and all placements are independent so

$$E(K|M=x, N=n) = (n-1)F^c(d)/F^c(x).$$

Now

$$P\{N=n|M=x\} = \frac{P\{M=x|N=n\}P\{N=n\}}{P\{M=x\}} = \frac{n[F^c(x)]^{n-1}p_n}{\hat{p}[F^c(x)]},$$

where (19a) is used to evaluate the denominator. Consequently,

$$E(K|M=x) = \frac{F^c(d)}{F^c(x)} \frac{1}{\hat{p}[F^c(x)]} \sum_{n=1}^{\infty} n(n-1)[F^c(x)]^{n-1}p_n = \frac{F^c(d)}{\hat{p}[F^c(x)]}\hat{P}''[F^c(x)],$$

where $\hat{P}''[F^c(x)]$ is the second derivative of $\hat{P}(\cdot)$ evaluated at $F^c(x)$. It is also the derivative of $\hat{p}(\cdot)$ evaluated at $F^c(x)$ so

$$\text{(A.3)} \qquad E(K|M=x) = \nu_M = \frac{F^c(d)\hat{p}'[F^c(x)]}{\hat{p}[F^c(x)]}.$$

Since

$$E(T_2^{**}|M=x) = E(T_i^{**}) + E(T_{ii}^{**}) + E(T_{iii}^{**})$$

we obtain the unnumbered equation preceding (21b).

Case 4 parallels case 2 with $[0,d)$ replacing $(d,1]$. We obtain immediately

$$\text{(A.4)} \qquad E(T_i^{**}) = \alpha\tau\nu F(x), \qquad \psi = \alpha F(d)\nu\tau,$$

and

$$\text{(A.5)} \qquad \beta = (\alpha+1)[F(x)-F(d)]\nu\tau.$$

Let K be the number of entities in $[0,d)$. Adapting the arguments leading to (A.3) yields

$$\text{(A.6)} \qquad E(K|L=x) = \nu_L = F(d)\hat{p}'[F(x)]/\hat{p}[F(x)].$$

REFERENCES

[1] E. G. COFFMAN, L. A. KLIMKO AND B. RYAN, *Analysis of scanning policies for reducing disk seek times*, this Journal, 1 (1972), pp. 269–279.

[2] E. G. COFFMAN AND M. HOFRI, *On the expected performance of scanning disks*, this Journal, 11 (1982), pp. 60–70.

[3] E. G. COFFMAN, *Analysis of a drum input/output queue under scheduled operation in a paged computer system*, J. ACM, 16 (1969), pp. 73–90.

[4] P. J. DENNING, *Effects of scheduling on file memory operations*, Proc. AFIPS SJCC, Vol. 30, Thompson Books, Washington, DC, 1966, pp. 9–21.

[5] 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. 624–633.

[6] H. FRANK, *Analysis and optimization of disk storage devices for time-sharing systems*, J. ACM, 16 (1969), pp. 602–620.

[7] DANIEL P. HEYMAN AND MATHEW J. SOBEL, *Stochastic Models in Operations Research*, Vol. 1, McGraw-Hill, New York, 1982.

[8] T. J. TEOREY AND T. B. PINKERTON, *A comparative analysis of disk scheduling policies*, Proc. 3rd Symposium on Operating Systems Principles, Stanford University, Stanford, CA 1971, pp. 114–121.

[9] RONALD W. WOLFF, *Poisson arrivals see time averages*, Oper. Res., 31 (1982), pp. 223–231.

SIAM J. COMPUT.
Vol. 13, No. 4, November 1984

© 1984 Society for Industrial and Applied Mathematics
002

EQUIVALENCE RELATIONS, INVARIANTS, AND NORMAL FORMS*

ANDREAS BLASS† AND YURI GUREVICH‡

Abstract. For an equivalence relation E on the words in some finite alphabet, we consider the recognition problem (decide whether two words are equivalent), the invariant problem (calculate a function constant on precisely the equivalence classes), the normal form problem (calculate a particular member of an equivalence class, given an arbitrary member) and the first member problem (calculate the first member of an equivalence class, given an arbitrary member). A solution for any of these problems yields solutions for all earlier ones in the list. We show that, for polynomial time recognizable E, the first member problem is always in the class Δ_2^P (solvable in polynomial time with an oracle for an NP set) and can be complete for this class even when the normal form problem is solvable in polynomial time. To distinguish between the other problems in the list, we construct an E whose invariant problem is not solvable in polynomial time with an oracle for E (although the first member problem is in $\mathrm{NP}^E \cap \text{co-}\mathrm{NP}^E$), and we construct an E whose normal form problem is not solvable in polynomial time with an oracle for a certain solution of its invariant problem.

Key words. polynomial time, Turing reducible, NP-complete, equivalence relation, certificate, normal form

Introduction. This paper was stimulated by [3], where finite structures (e.g. graphs) were considered as inputs for algorithms. Since one is usually interested only in the isomorphism types of structures, it is natural to ask whether these types can be represented in a form suitable for use as inputs in place of the structures themselves. To avoid trivial "solutions", we insist that the isomorphism type of a structure be (rapidly) computable when the structure itself is given. For this to be possible, it is necessary that the isomorphism problem, for the structures considered, be (rapidly) solvable; indeed, to tell whether two structures are isomorphic, one just computes their isomorphism types and checks whether they are equal. Is this necessary condition sufficient as well? That is, does a solution of the isomorphism problem yield a presentation of the isomorphism classes as well? In many cases it does, because the solution of the isomorphism problem proceeds by calculating an invariant (or a system of invariants) such that two structures of the sort considered are isomorphic exactly when their invariants agree. In such a case, the invariants themselves can be used as a presentation of the isomorphism classes. It is not at all obvious, however, that every solution of an isomorphism problem must yield such invariants. For example, the solution in [2] for trivalent graphs does not appear to yield invariants. We shall show, in the more general setting of arbitrary equivalence relations rather than isomorphism, that one cannot, in general, calculate invariants even if one can tell when two objects are equivalent.

We shall also consider two problems more difficult than the invariant problem. One is obtained by insisting that the invariant be a member of the equivalence class it represents. In the isomorphism-type situation discussed above, this amounts to insisting that one can (rapidly) compute from an isomorphism type some structure in it. An even more difficult problem arises if one insists that the invariant be the first (in some suitable ordering) member of the equivalence class. We shall prove that each of these problems is strictly more difficult than its predecessor.

---

* Received by the editors December 22, 1982, and in final form July, 1983.

† Department of Mathematics, University of Michigan, Ann Arbor, Michigan 48109.

‡ Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, Michigan 48109.

To make these ideas precise, we consider an equivalence relation E over Σ^*, where Σ is a finite alphabet, and we consider the following four problems:

Recognition problem. Given x and y in Σ^*, determine whether xEy.

Invariant problem. Calculate, for x in Σ^*, an invariant $F(x)$ (in Σ_1^* for some finite alphabet Σ_1) such that, for all x and y in Σ^*, xEy if and only if $F(x)=F(y)$.

Normal form problem. Calculate, for x in Σ^*, a "normal form" $F(x)$ (in Σ^*) such that, for all x and y in Σ^*, $xEF(x)$ and if xEy then $F(x)=F(y)$.

First member problem. Calculate, for x in Σ^*, the first y such that xEy.

In the first member problem, "first" refers to the following standard ordering of Σ^*: y precedes z if either y is shorter than z or they have the same length and y lexicographically precedes z. For many of the equivalence relations we consider, equivalent words have the same length, so "first" can be understood as "lexicographically first".

It is clear that any solution of the first member problem solves the normal form problem and that any solution to the normal form problem solves the invariant problem. Furthermore, given a solution of the invariant problem, we can solve the recognition problem by computing and comparing $F(x)$ and $F(y)$.

In the context of ordinary recursion theory, all four problems are equivalent, for, if we know how to solve the recognition problem, then we can solve the first member problem by testing each y in Σ^* in turn (in standard order) until we find one equivalent to x. Since the search is bounded by x, the problems remain equivalent if we consider only primitive recursive computations. Our objective is to prove that all four problems are inequivalent in the context of polynomial time computability. (They are also inequivalent in higher-type recursion theory, but that does not concern us here.) Unfortunately, this objective cannot be achieved without proving $\mathrm{P}\neq\mathrm{NP}$.

Indeed, the search procedure indicated above implies that, when the recognition problem is in P, the first member problem is in the polynomial hierarchy of Stockmeyer [4] and is therefore in P if $\mathrm{P}=\mathrm{NP}$. More precisely, the first member problem is in Stockmeyer's class Δ_2^{P}, that is, it is computable in polynomial time with an oracle for an NP set, if E is in P. The computation proceeds as follows, using an oracle for the NP set

$$\{(x, y)\,|\,\text{some member of the equivalence class of } x \text{ precedes } y\}.$$

Given x, we begin by querying the oracle about $(x, 1^l)$, for various values of $l\leq \text{length}(x)$, to determine by binary search the length m of the first y in the equivalence class of x; this takes approximately log (length (x)) steps. Then we query the oracle about $(x, 10^{m-1})$ to determine the first digit y_1 of the desired y. Then we query about $(x, y_1 10^{m-2})$ to determine the next digit y_2, etc. In m such steps we determine all of y.

We shall, in § 1, show that the complexity estimate just obtained is optimal, by constructing an E whose first member problem is Δ_2^{P}-complete while the normal form problem has a solution in P (and therefore so do the invariant and recognition problems). This result implies that, if $\mathrm{P}\neq\mathrm{NP}$, the first member problem is strictly harder than the normal form problem.

In § 2, we shall show that no problem in our list is polynomial time Turing reducible to an earlier one. Specifically, we shall show that the invariant problem is strictly harder than the recognition problem by constructing an equivalence relation E whose invariant problem has no solution computable in polynomial time with an oracle for E. We improve this result by showing that we can simultaneously arrange for the first member problem to be in $\mathrm{NP}^E\cap\text{co-NP}^E$. Then we show that the normal form problem is strictly harder than the invariant problem by constructing an equivalence relation

E and a solution F of its invariant problem such that no solution of the normal form problem is computable in polynomial time with an oracle for F. The corresponding result for the normal form and first member problems, namely the existence of an equivalence relation E and a solution F of its normal form problem such that the first member problem is not solvable in polynomial time with an oracle for F, follows immediately from Theorem 1, relativized to an oracle A such that $\mathrm{P}^A \neq \mathrm{NP}^A$ [1]. Nevertheless, we present a direct and somewhat simpler construction of such E and F using some of the same methods as the other constructions in § 2.

It is natural to ask whether the inequivalence of each pair of consecutive problems in our list can be proved for absolute (without oracles) polynomial time computability. As indicated above, this cannot be done without proving $\mathrm{P} \neq \mathrm{NP}$. Still, in view of Theorem 1, one could hope to show that these questions are equivalent to natural questions such as $\mathrm{NP} \stackrel{?}{=} \text{co-NP}$. Results of this sort will be presented in a subsequent paper (to appear in the proceedings of the Recursive Combinatorics Symposium, Münster, 1983, Springer Lecture Notes).

Finally, in § 3, we formulate some open problems.

1. The first member problem. We show, in this section, that the upper bound of Δ_2^{P}, for the complexity of the first member problem associated with a polynomial time computable equivalence relation, is optimal, even if we require a polynomial time solution for the normal form problem rather than just the recognition problem.

THEOREM 1. *There is an equivalence relation E on $\{0, 1\}^*$ for which the normal form problem is solvable in polynomial time, but the first member problem is Δ_2^{P}-complete.*

Remark 1. When we say that the problem of computing a certain function F from Σ^* to Σ_1^* is complete for (or hard for, or in) a certain complexity class, we mean that the following decision problem is complete for (or hard for, or in) that class: Given a word x in Σ^*, a letter a in Σ_1 and a positive integer n (in unary notation), decide whether the nth letter in $F(x)$ is a.

Remark 2. In the statement and proof of Theorem 1, completeness refers to log-space reductions. (The proof of the weaker result, where completeness refers to polynomial time reduction, would be no easier, although the words w in the proof could then be assumed to be empty by coding them into m.)

Remark 3. For the equivalence relation E that we shall construct, the last member problem is solvable in polynomial time.

Proof of Theorem 1. We shall work with the alphabet $\{0, 1, 2\}$; the reduction to $\{0, 1\}$ is routine. Let A be a fixed NP-complete set in $\{0, 1\}^*$, accepted by a particular nondeterministic Turing machine N in time bounded by a polynomial. Let p be a monotone increasing polynomial (essentially the square of the time bound) such that every computation of N with an input of length l can be coded (in a standard way) by some c in $\{0, 1\}^*$ of length $p(l)$. Henceforth, we shall not distinguish between c and the computation it codes.

Consider the following auxiliary problem. An instance of the problem is $m2w2^k$, where m is a word in $\{0, 1\}^*$ coding, in some standard way, a query machine M (a deterministic Turing machine with alphabet $\{0, 1\}$ that is to interact with an as yet unspecified oracle), w is a word in $\{0, 1\}^*$, and k is a positive integer. The question in the instance $m2w2^k$ of the auxiliary problem is whether machine M with oracle A, henceforth written M^A, with input w, halts in fewer than k steps. It is routine to check that this problem is Δ_2^{P}-complete.

To prove the theorem, we shall, after some preliminary work, define an equivalence relation E on $\{0, 1, 2\}^*$, give a polynomial time computable solution for its normal

form problem, and give a log-space computable reduction of the auxiliary problem to the first member problem for E. For each instance of the auxiliary problem, we define its *plausible computations* to be the words in $\{0, 1, 2\}^*$ that consist of

(i) $m2w2^k$, followed by
(ii) a string $a_1, \cdots, a_k$ of length k, followed by
(iii) a single digit h, followed by
(iv) a string of length k^2, which we think of as consisting of k blocks $q_1, \cdots, q_k$ each having length k, followed by
(v) a string of length $k \cdot p(k)$, which we think of as consisting of k blocks $c_1, \cdots, c_k$ each having length $p(k)$,

subject to the following requirements:

(α) When the query machine M coded by m is started with input w and allowed to run for $k-1$ steps (or until it halts if this occurs earlier), its first query (if any) to the oracle concerns the unique string q_1' such that $q_1 = q_1'10^r$ for some r. (Note that, by the time bound, length $(q_1') < k =$ length (q_1), so this makes sense.) If the answer to this first query is a_1, under the convention that 0 means "Yes" and 1 means "No", then the next query, if any, concerns the q_2' such that $q_2 = q_2'10^r$ for some r. If the answer to the second query is a_2, the next query (if any) concerns the q_3' such that $q_3 = q_3'10^r$ for some r, etc. (Note that, by the time bound, there will be fewer than k queries altogether.)

(β) If there are exactly j queries in the computation described in (α), then $a_i = 1$ and $q_i = 0^k$ and $c_i = 0^{p(k)}$ for $j < i \leq k$.

(γ) If the computation in (α) reaches a halting state, then $h = 0$; otherwise $h = 1$.

(δ) For each i such that $a_i = 0$, c_i has the form $c_i'10^r$, where c_i' is a computation showing that N accepts q_i'. If $a_i = 1$ then c_i is a string of 0's.

Observe that the property of being a plausible computation is decidable in polynomial time.

For orientation and future reference, we construct, for each $m2w2^k$, a particular plausible computation, called the correct computation. Let the machine M^A with input w run for $k-1$ steps. Let $q_1', \cdots, q_j'$ be its queries to the oracle, and let $a_1, \cdots, a_j$ be the oracle's answers, with 0 representing "Yes". By the time bound, j and each length (q_i') must be smaller than k. For each $i \leqq j$, define q_i to be $q_i'10^r$, where r is chosen (depending on i) so that length $(q_i) = k$. For $j < i \leqq k$, let $a_i = 1$, $q_i = 0^k$, $c_i = 0^{p(k)}$ as required by (β). Note that, with our choices of a's and q's, the computation of M with oracle A is exactly the computation described in (α) above. Let h be 0 if this computation enters a halting state, and 1 otherwise. Finally, define c_i to be $0^{p(k)}$ if $a_i = 1$ and to be $c_i'10^r$ if $a_i = 0$, where c_i' is a computation of N accepting q_i', where r is chosen to make length $(c_i) = p(k)$, and where c_i' is chosen, among all computations accepting q_i', so that c_i is lexicographically as early as possible. It is immediate that $s = m2w2^k a_1 \cdots a_k h q_1 \cdots q_k c_1 \cdots c_k$ is a plausible computation for $m2w2^k$. We call it the *correct computation* for $m2w2^k$.

The important fact about this correct computation is that it lexicographically precedes all other plausible computations for $m2w2^k$. Indeed, suppose $s^* = m2w2^k a_1^* \cdots a_k^* h^* q_1^* \cdots q_k^* c_1^* \cdots c_k^*$ were a lexicographically earlier plausible computation for the same $m2w2^k$. If the first difference between these two plausible computations occurs at a_i, then we would have $a_i^* = 0$ and $a_i = 1$. The computations described in (α) for s and s^* are identical up to, but not including, the oracle's response to the ith query. In particular, that query itself is the same: $q_i = q_i^*$. Since $a_i^* = 0$, condition (δ) for s^* guarantees that c_i^* contains a computation $c_i^{*\prime}$ of N accepting q_i^*. So $q_i \in A$. But this contradicts the fact, expressed by $a_i = 1$, that the A oracle responded

negatively to query q_i. Therefore, the first difference between s and s^* must come later than all the a_i's. The computations described in (α) for s and s^* are therefore the same. Thus $h = h^*$ (by (γ)) and $q_i = q_i^*$ for all i (by (α) and (β)). The first difference between s and s^* must therefore be at some c_i, and, by (δ), we must have $a_i = 0$ for this i. But then we cannot have c_i^* lexicographically preceding c_i, because of the way we chose c_i for the correct computation.

We are, at last, ready to define E. For each $m2w2^k$, we put into one equivalence class all its plausible computations together with the extra word $m2w2^k1^l$, where $l = k(p(k)+k+1)+1$. This value of l was chosen so that the extra word has the same length as the plausible computations for $m2w2^k$. Note that the extra word is always the (lexicographically) last member of its equivalence class. To complete the definition of E, we declare that any word that is neither a plausible computation nor an extra word shall be equivalent only to itself.

The normal form problem for E is solved by the following polynomial time algorithm. Given x, check whether it is a plausible computation for its maximal initial segment ending with a 2. If not, give x as output. If so, give the extra word for that initial segment as output. This algorithm always produces the last member of the equivalence class of x.

The auxiliary problem is reduced to the first member problem for E by the following algorithm. Given an instance $m2w2^k$ of the auxiliary problem, calculate $l = k(p(k)+k+1)+1$, and then write $m2w2^k1^l$. The answer to this instance is given by the digit h in position length$(m2w)+2k+1$ of the first member of the equivalence class of $m2w2^k1^l$. This algorithm works because the equivalence class of the extra word $m2w2^k1^l$ contains all the plausible computations for $m2w2^k$ and its first member is, as we saw above, the correct computation for $m2w2^k$; the h digit in this correct computation encodes the answer to the auxiliary problem for $m2w2^k$. □

2. Nonreducibility. The main result of this section is that, in the context of polynomial time computability, the recognition problem, the invariant problem, the normal form problem and the first member problem are of strictly increasing complexity. None of these problems can be reduced in general to the previous problem on the list.

THEOREM 2. (a) *The invariant problem is not polynomial time Turing reducible to the recognition problem. Indeed, there is an equivalence relation E on* $\{0, 1\}^*$ *such that the invariant problem for E cannot be solved by a polynomial time algorithm with an oracle for E.*

(b) *The normal form problem is not polynomial time Turing reducible to the invariant problem. Indeed, there are an equivalence relation E on* $\{0, 1\}^*$ *and a polynomially bounded solution* $F: \{0, 1\}^* \to \{0, 1\}^*$ *of its invariant problem, such that the normal form problem for E cannot be solved by a polynomial time algorithm with an oracle for F.*

(c) *The first member problem is not polynomial time Turing reducible to the normal form problem. Indeed, there are an equivalence relation E on* $\{0, 1\}^*$ *and a polynomially bounded solution F of its normal form problem, such that the first member problem for E cannot be solved by a polynomial time algorithm with an oracle for F.*

Remark 4. To say that a function F is polynomially bounded means that the length of $F(x)$ is bounded by a polynomial in the length of x. We think of computation with an oracle for a function F (rather than a predicate) as follows. The machine can write a word x on its query tape and receive from the oracle, in a single computation step, the value of $F(x)$, printed on a special tape. A query of this sort can be simulated by a sequence of queries concerning the predicate "The nth letter in $F(x)$ is a".

Remark 5. Theorem 1 is relativizable to any oracle. By taking an oracle relative to which P$\neq$NP, as in [1], we immediately obtain part (c) of Theorem 2. Nevertheless we shall give another, more direct proof of (c).

Proof of Theorem 2. Fix an enumeration $M_0, M_1, \cdots$ of all polynomial time bounded query machines with alphabet $\{0, 1\}$, and let p_n be the polynomial clock bound of M_n.

We construct the desired equivalence relation E in stages. Initially E is the equality relation on $\{0, 1\}^*$. At each stage n we may update E by putting into it two pairs (x, y) and (y, x) for some binary strings x, y of the same length d_n. Here $d_0 < d_1 < \cdots$. Thus the resulting equivalence relation has the special properties that each equivalence class has at most two members, the two members of any nontrivial equivalence class are words of the same length, and there is at most one nontrivial equivalence class for each length.

In part (b) we construct the desired invariant function F simultaneously with E. Initially $F(x) = x1$ for every binary word x. At a stage n we may update F by stipulating that $F(x) = 0^{d_n+1}$ for one or two words x of length d_n. In part (c) we construct the desired normal form function F simultaneously with E. Initially F is the identity function $F(x) = x$. At each stage n we may update F by stipulating that $F(x) = 1^{d_n}$ for one word of length d_n.

The sequence $d_0 < d_1 < d_2 < \cdots$ is chosen in such a way that for each n, $p_n(d_n) < d_{n+1}$ and $p_n(d_n) < 2^{d_n-1}$. Thus, computing on an input of length d_n, M_n asks fewer than 2^{d_n-1} queries and each query is shorter than d_{n+1}.

In the rest of the proof we consider a stage n of the construction. Let x and y range over the binary words of length $d = d_n$. Let $M = M_n$ and $p = p_n$.

(a) For each x let x' be the result produced by M on the input x, when it uses the current E as an oracle. If there are distinct x, y with $x' = y'$ go to the next stage. M^E fails to solve the invariant problem for E because $M^E(x) = x' = y' = M^E(y)$, whereas $(x, y) \notin E$. (Note that later stages of the construction will not affect the values of $M^E(x)$ and $M^E(y)$, because they alter E only on words longer than any query involved in the computation of these values.)

Suppose, on the other hand, that $x' \neq y'$ for all $x \neq y$. We say that x *affects* y if M queries E about (x, y) or (y, x) in the computation of y'. Each y is affected by at most $p(d) < 2^{d-1}$ elements x.

LEMMA 1. *Let R be a binary relation on a nonempty set S of cardinality $2k$. Suppose that for each $u \in S$ there are fewer than k elements $v \in S$ with $(u, v) \in R$. Then there exist distinct $u, v \in S$ such that neither (u, v) nor (v, u) is in R.*

Proof of Lemma 1. Otherwise, the set of all $k(2k-1)$ two-element subsets of S would be the union of the $2k$ sets

$$R_u = \{\{u, v\}: (u, v) \in R\}$$

each of which has cardinality $\leqq k-1$. This is a contradiction. The lemma is proved. □

By the lemma (with $S = \{0, 1\}^d$ and R being the relation "is affected by") there are distinct x, y such that neither of x and y affects the other. Put (x, y) and (y, x) into E. This does not alter the computations of x' and y'. Go to the next stage. M^E fails to solve the invariant problem for E because $(x, y) \in E$, whereas $M^E(x) = x' \neq y' = M^E(y)$.

(b) For each x let F_x by the current F with the following change: $F_x(x) = 0^{d+1}$. Let x' be the result computed by M with input x and the oracle F_x.

If there is an x with $x' \neq x$, then choose one such x, and update F by making it equal to F_x. This does not affect the computation of x'. Go to the next stage. Note

that the updated F is one-to-one on $\{0,1\}^d$ so that there is no need to update E. Since $(x, x') \notin E$, M^F fails to solve the normal problem for E.

Suppose, on the other hand, that $x' = x$ for all x. Choose distinct x and y such that M does not query F_x about y in the computation of x', and M does not query F_y about x in the computation of y'. This choice is possible, by Lemma 1, because each computation involves at most $p(d) < 2^{d-1}$ queries. Update F by stipulating $F(x) = F(y) = 0^{d+1}$, put pairs (x, y) and (y, x) into E, and go to the next stage. M^F fails to solve the normal form problem for E because $M^F(x) = x \neq y = M^F(y)$, whereas $(x, y) \in E$.

(c) For each x let F_x be the current F with the following change: $F_x(x) = 1^d$. Let x' be the result computed by M with oracle F_x on input x. If there is an x with $x' \neq x$, then choose one such x, update F by making it equal to F_x, put $(x, 1^d)$ and $(1^d, x)$ into E, and go to the next stage. M^F fails to solve the first member problem for E because $M^F(x) = x' \neq x$ whereas x is the first member in its equivalence class.

Suppose, on the other hand, that $x' = x$ for every x. In particular $(1^d)' = 1^d$. Choose $x < 1^d$ such that M does not query the oracle about x in the computation of $(1^d)'$. This choice is possible because the computation involves at most $p(d) < 2^{d-1}$ queries. Update F by making it equal to F_x. This does not change the computations of x' and $(1^d)'$. Put $(x, 1^d)$ and $(1^d, x)$ into E, and go to the next stage. M^F fails to solve the first member problem for E because $M^F(1^d) = 1^d$, whereas 1^d is not the first member in its equivalence class. □

Our last result is an improvement of part (a) in Theorem 2. It asserts that the invariant problem can be intractable even when the recognition problem is tractable and the first member problem is nearly tractable.

THEOREM 3. *There is an equivalence relation E on $\{0,1\}^*$ such that*

(i) *the invariant problem for E is not solvable by a polynomial time algorithm with an oracle for E; and*

(ii) *the first member problem for E is in* NP *and* co-NP *relative to E.*

Proof. In the proof of part (a) of Theorem 2 we constructed an equivalence relation E such that every equivalence class has at most two elements, the two members of any nontrivial equivalence class are words of the same length, and there is at most one nontrivial equivalence class for each length. We shall modify the construction so that for each length $l > 0$ there is exactly one nontrivial equivalence class.

First make sure that $4p_n(d_n) < 2^{d_n} - 3$. Then turn to a stage n of the construction. Here we first update E by adding to it all pairs $(0^l, 1^l)$ and $(1^l, 0^l)$ such that $0 < l < d_0$ if $n = 0$, and $d_{n-1} < l < d_n$ otherwise. We use the notation of the proof of Theorem 2. Let x' be the result produced by M on input $x \in \{0,1\}^d$ with an oracle for the current E. If there is a pair $x \neq y$ with $x' = y'$ pick one such pair x, y. Computing x' and y', M queried E about $\leq 2p(d)$ pairs (u, v) altogether. Hence at most $4p(d) < 2^d - 3$ words were involved in all those queries. Choose $u, v \in \{0,1\}^d$ such that u, v, x, y are different and u, v were not involved in the queries of the computations of x', y'. Update E by adding (u, v) and (v, u) to E. This does not alter the computations of x', y'. Go to the next stage. If, on the other hand, $x' \neq y'$ for all $x \neq y$ proceed as in the proof of part (a) of Theorem 2.

To show that the function sending each x to the first member of its equivalence class is in $NP^E \cap co\text{-}NP^E$, we first observe that its graph, i.e. the binary relation "y is the first member of the E-class of x," is NP^E by the following procedure. If $y < x$ and $(x, y) \in E$, then accept; if $y = x$, then guess two distinct but E-related words of the same length as x, and accept if x is not the later of these two words. To finish the proof, we make the general remark that any polynomially bounded function F whose

graph is in NP^E is itself in $\mathrm{NP}^E \cap \text{co-NP}^E$. The NP^E procedure for determining that the nth letter in $F(x)$ is (resp. is not) a is to guess the value y of $F(x)$ and a computation showing that (x, y) belongs to the graph of F and then to accept if the nth letter of y is (resp. is not) a.

3. Problems. The results in this paper suggest several natural problems. Two that strike us as particularly interesting are the following.

1. Can results similar to ours be obtained if, instead of considering arbitrary equivalence relations, one restricts attention to the relation of isomorphism between (standard representations of) structures?

2. Are there analogues of Theorem 1 for the invariant and normal form problems? That is, is there a polynomial time computable E such that every polynomially bounded solution of its invariant problem is Δ_2^{P}-hard? And is there an E such that the invariant problem is solvable in polynomial time but every polynomially bounded solution of the normal form problem is Δ_2^{P}-hard?

REFERENCES

[1] T. BAKER, J. GILL AND R. SOLOVAY, *Relativizations of the* P = ?NP *question*, this Journal, 4 (1975), pp. 431–442.

[2] Z. GALIL, C. M. HOFFMANN, E. M. LUKS, C. P. SCHNORR AND A. WEBER, *An* $O(n^3 \log n)$ *deterministic and an* $O(n^3)$ *probabilistic isomorphism test for trivalent graphs*, Proc. 23rd Annual IEEE Symposium on Foundations of Computer Science, 1982, pp. 118–125.

[3] Y. GUREVICH, *Toward logic tailored for computational complexity*, Proc. European Logic Colloquium, Aachen, 1983, Springer Lecture Notes, to appear.

[4] L. J. STOCKMEYER, *The polynomial-time hierarchy*, Theoret. Comp. Sci., 3 (1977), pp. 1–22.

SIAM J. COMPUT.
Vol. 13, No. 4, November 1984

© 1984 Society for Industrial and Applied Mathematics
003

PREEMPTIVE SCHEDULING OF A MULTIPROCESSOR SYSTEM WITH MEMORIES TO MINIMIZE MAXIMUM LATENESS*

TEN-HWANG LAI†‡ AND SARTAJ SAHNI†

Abstract. We develop an $O(q^2n+n \log n)$ algorithm to obtain a preemptive schedule that minimizes maximum lateness when n jobs with given due dates and memory requirements are to be scheduled on m processors ($n \geqq m$) of given memory sizes q is the number of distinct due dates. The value of the minimum maximum lateness can itself be found in $O(qn+n \log n)$ time.

Key words. preemptive scheduling, maximum lateness, memory requirements

1. Introduction. The problem of scheduling n jobs on a multiprocessor system consisting of m processors, each having its own independent memory of size μ_i, has been considered by Kafura and Shen [2]. Associated with each job are a processing time t_j and a memory requirement m_j. Job j can be processed on processor i if and only if $m_j \leqq \mu_i$. No job can be simultaneously processed on two different processors and no processor can process more than one job at any given instant of time. In a preemptive schedule, it is possible to interrupt the processing of a job and resume it later on a possibly different processor. In a nonpreemptive schedule, each job is processed without interruption on a single processor.

Obtaining minimum finish time nonpreemptive schedules is NP-hard even when $m=2$ and $\mu_1=\mu_2$ [1]. Hence, Kafura and Shen [2] study the effectiveness of several heuristics for nonpreemptive scheduling. For the preemptive case, they develop an $O(n \log n)$ algorithm that obtains minimum finish time schedules (without loss of generality, we may assume $n \geqq m$). Their algorithm begins by first computing the finish time, f^*, of a minimum finish time schedule. This is done as follows. First, the jobs and processors are reordered such that $\mu_1 \geqq \mu_2 \geqq \cdots \geqq \mu_m$ and $m_1 \geqq m_2 \geqq \cdots \geqq m_n$. This reordering takes $O(n \log n)$ time (again, we assume $n \geqq m$). Let F_i be the set of all jobs that can be processed only on processors $1, 2, \cdots, i$ because of their memory requirements. Let X_i be the sum of the processing requirements of the jobs in F_i. $X_i=0$ if and only if $F_i=\phi$. Kafura and Shen [2] show that

$$f^* = \max\{\max_j \{t_j\}, \max_i \{X_i/i\}\}. \tag{1.1}$$

The jobs may now be scheduled in the above order ($m_1 \geqq m_2 \geqq \cdots \geqq m_n$) using f^* and McNaughton's rule [4].

In this paper, we extend the work of [2] to the case where each job has a due time d_j associated with it. Every job is released at a common release time. We are interested in first determining whether or not the n jobs can be preemptively scheduled in such a way that every job is complete by its due time. A schedule that has this property is called a *feasible schedule.*

The existence of a feasible schedule can be determined in polynomial time using network flow techniques [3]. The complexity of the algorithm that results from this approach is $O(qn(n+qs) \log^2 (n+qs))$ where q is the number of distinct due dates

* Received by the editors August 19, 1981, and in revised form June 23, 1983. This research was supported in part by the Office of Naval Research under contract N00014-80-C-0650 and in part by the National Science Foundation under grant MCS80-005856.

† Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455.

‡ Present address Department of Computer and Information Science, The Ohio State University, Columbus, Ohio 43210.

and s the number of different memory sizes. In fact, a feasible schedule (whenever one exists) may be obtained in this much time. In § 2 we develop another algorithm for this problem. This algorithm is considerably harder to prove correct, but has a complexity that is only $O(qn+n\log n)$. A feasible schedule can be constructed in $O(q^2n+n\log n)$ time. In arriving at the algorithm of § 2, we develop a necessary and sufficient condition for the existence of a feasible schedule. With the help of this condition, in § 3, we develop an algorithm to obtain a schedule that minimizes the maximum lateness. This algorithm is also of complexity $O(q^2n+n\log n)$.

Sahni [5] and Sahni and Cho [6], [7] have done work related to that reported here. They have considered preemptive scheduling of n jobs with due dates when $\mu_1=\mu_2=\cdots=\mu_m$. For the special case when all memory sizes are the same, Sahni [5] has developed an $O(n\log mn)$ algorithm to obtain a feasible schedule (when one exists). Sahni and Cho [6], [7] have obtained efficient algorithms for the case when $\mu_1=\mu_2=\cdots=\mu_m$ and the processors run at different speeds.

2. A fast feasibility algorithm. In this section, we first derive a necessary and sufficient condition for the existence of a feasible schedule. This condition is used to obtain a fast algorithm to construct a feasible schedule whenever such a schedule exists. In § 3, this necessary and sufficient condition is used to obtain a fast algorithm to minimize the maximum lateness.

Each job is characterized by a triple (t_j, d_j, m_j) where t_j is the task time of job j, d_j is its due time, and m_j, its memory requirement. Let $\delta_1, \delta_2, \cdots, \delta_q$, $\delta_1<\delta_2<\cdots<\delta_q$, denote the distinct due times in the multiset $\{d_1, d_2, \cdots, d_n\}$. Let δ_0 be the common release time for the n jobs. Without loss of generality, we may assume that $\delta_0<\delta_1$.

Let $\mu(1), \mu(2), \cdots, \mu(s)$, $\mu(1)>\mu(2)>\cdots>\mu(s)$ be the distinct memory sizes in the multiset $\{\mu_1, \mu_2, \cdots, \mu_m\}$. Let $\mu(s+1)=0$ for convenience. Let P_k denote the set of all processors with memory size equal to $\mu(k)$; i.e., $P_k=\{i|\mu_i=\mu(k)\}$, $1\leqq k\leqq s$. Let J_k be the set of all jobs that can be processed only on processors in $P_1, P_2, \cdots, P_k$ because of their memory requirements; i.e., $J_k=\{j|\mu(k)\geqq m_j>\mu(k+1)\}$, $1\leqq k\leqq s$. We shall refer to P_k as processor class k and J_k as job class k.

2.1. A necessary and sufficient condition. It is easy to see that in every feasible schedule for the n jobs, at least the amount

$$b(j,d)=\begin{cases} t_j & \text{if } d_j\leqq\delta_d,\\ t_j-\min\{t_j, d_j-\delta_j\} & \text{otherwise}\end{cases}$$

of job j must be completed by δ_d, $1\leqq j\leqq n$, $0\leqq d\leqq q$. Observe that if there exist j and d such that $b(j,d)>\delta_d-\delta_0$, then there is no feasible schedule.

Of the minimum amount $b(j,d)$ that must be completed before δ_d, at most

$$a(j,d)=\min\{b(j,d), \delta_1-\delta_0\}$$

can be completed by δ_1.

Define $B(k,d)$ to be the sum of the $b(j,d)$s for those jobs j in J_k. Define $A(k,d)$ in a similar manner. Specifically,

$$B(k,d)=\sum_{j\in J_k} b(j,d), \qquad 1\leqq k\leqq s, \quad C\leqq d\leqq q$$

and

$$A(k,d)=\sum_{j\in J_k} a(j,d), \qquad 1\leqq k\leqq s, \quad 0\leqq d\leqq q.$$

$B(k, d)$ gives a lower bound on the amount of jobs in J_k that must be completed by δ_d. $A(k, d)$ gives the maximum amount of $B(k, d)$ that can be done by δ_1.

Define a capacity function $C(k, d)$ such that

$$C(k, d) = |P_k|(\delta_d - \delta_0), \qquad 1 \leqq k \leqq s, \quad 0 \leqq d \leqq q.$$

$C(k, d)$ gives the available processing capacity in processor class k from the release time δ_0 to the due time δ_d.

One readily observes that if a feasible schedule exists, then

$$\sum_{i=1}^{k} B(i, d) \leqq \sum_{i=1}^{k} C(i, d) \tag{2.0}$$

for all k, d, $1 \leqq k \leqq s$, $0 \leqq d \leqq q$. While (2.0) provides a necessary condition for the existence of a feasible schedule, it does not provide a sufficient condition. We leave it to the reader to construct an instance that satisfies (2.0) but for which no feasible schedule exists.

This necessary condition can be strengthened by using the notion of a profile function. Let π be the set of all nonincreasing functions σ with domain $\{0, 1, 2, \cdots, s\}$ and range $\{0, 1, \cdots, q\}$. Recall that s is the number of processor classes and q the number of distinct due times. Thus

$$\pi = \{\sigma | \sigma: \{0, 1, \cdots, s\} \to \{0, 1, \cdots, q\} \text{ and } \sigma(k) \geqq \sigma(k+1), 0 \leqq k < s\}.$$

π defines the set of *profile functions.* Each profile function σ defines a *profile* in a timing diagram (see [8]), i.e., the curve of $t = \delta_{\sigma(i)}$, $i = 1, 2, \cdots, s$. We shall refer to the profile defined by σ simply as the profile σ. For example, consider the case $s = 4$, $q = 5$ and the profile function σ such that $\sigma(0) = \sigma(1) = \sigma(2) = 4$, $\sigma(3) = 2$, and $\sigma(4) = 1$. Figure 2.1 displays σ pictorially.

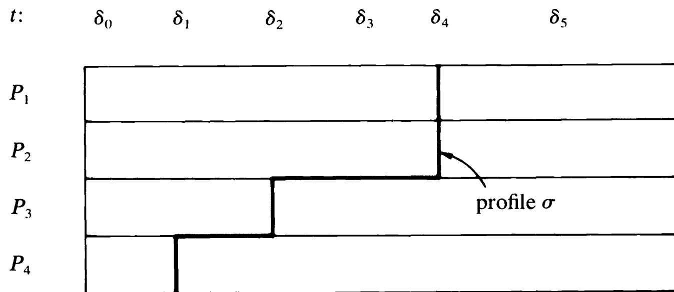


FIG. 2.1. *Example profile.*

Let σ be a profile function. In any feasible schedule, at least the amount $B(i, \sigma(i))$ of processing from J_i must be done on $P_1, P_2, \cdots, P_i$ by time $\delta_{\sigma(i)}$. Since σ is nonincreasing, $B(i, \sigma(i))$ is a lower bound on the amount of processing from J_i that must be scheduled between δ_0 and the profile σ (see Fig. 2.2). Since the J_is are pairwise disjoint, it follows that $\sum_{i=1}^{s} B(i, \sigma(i))$ is a lower bound on the amount of processing that must be scheduled between δ_0 and the profile σ. Since $\sum_{i=1}^{s} C(i, \sigma(i))$ is the total processing capacity of $P_1, \cdots, P_s$ between δ_0 and the profile σ, it follows that if there is a feasible schedule, then

$$\sum_{i=1}^{s} B(i, \sigma(i)) \leqq \sum_{i=1}^{s} C(i, \sigma(i)) \tag{2.1}$$

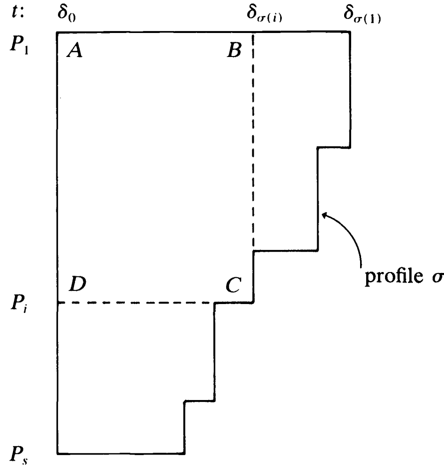


FIG. 2.2. *$B(i, \sigma(i))$ must be scheduled in the rectangle ABCD and hence to the left of the profile σ.*

for every $\sigma \in \pi$. We shall show in Theorem 1 that there is a feasible schedule if and only if (2.1) holds.

2.2. Obtaining a feasible schedule. We are now ready to introduce the ideas that lead to the feasibility algorithm of § 2.3. Our algorithm begins by computing the amount w_j of each job j that is to be scheduled between δ_0 and δ_1. These w_js are determined so that they can be scheduled in the interval δ_0 to δ_1 and the remaining processing requirements $\{t_j - w_j\colon 1 \leqq j \leqq n\}$ can be feasibly met from δ_1 to δ_q.

Once the w_js are known, the schedule from δ_0 to δ_1 may be obtained. The schedule for the remaining $q-1$ intervals is similarly obtained. The w_js are computed starting with jobs in J_1 and proceeding to J_2, etc. Observe that jobs in $J_{k+1} \cup \cdots \cup J_s$ compete with the jobs in J_k for the processing time available on $P_1, \cdots, P_k$ from δ_0 to δ_1. Hence, while determining $w_j, j \in J_k$, we must also determine a value R_k that represents the amount of $P_1, \cdots, P_k$s processing capacity in the interval δ_0 to δ_1 that is to be reserved for the jobs in $J_{k+1} \cup \cdots \cup J_s$.

Considering the definition of $a(j, d)$, it seems plausible to compute the w_js using the greedy method:

> **for** $d \leftarrow 0, 1, 2, \cdots$ **until** satisfied **do**
> set w_j to $a(j, d)$ for every $j \in J_k$
> **end**.

We would like to compute R_k in a similar manner from a yet to be defined quantity $Y(k, d)$. We shall define $Z(k, d)$, $Y(k, d)$, and $X(k, d)$, $0 \leqq k \leqq s$, $0 \leqq d \leqq q$, so that:

(i) In any feasible schedule, at least $Z(k, d)$ much of $J_{k+1} \cup \cdots \cup J_s$ must be done on $P_1, \cdots, P_k$ by time δ_d.

(ii) In any feasible schedule, at least $X(k, d)$ much of $J_{k+1} \cup \cdots \cup J_s$ must be processed on $P_1, \cdots, P_k$ between δ_1 and δ_d.

(iii) $Y(k, d) = Z(k, d) - X(k, d)$. Hence, one may think of $Y(k, d)$ as representing the maximum amount of $Z(k, d)$ that can be done between δ_0 and δ_1.

It is important to note that $(R_k, Z(k, d), Y(k, d))$, when regarded as an attribute of $J_{k+1} \cup \cdots \cup J_s$ is the counterpart of $(w_j, b(j, d), a(j, d))$ if the latter is considered an attribute of job j, $j \in J_k$.

We now proceed to define $Z(k, d)$. Define π_d as below:

$$\pi_d = \{\sigma | \sigma \in \pi \text{ and } \sigma(i) \leqq d,\ 0 \leqq i \leqq s\}.$$

From the discussion of § 2.1, it follows that for any profile $\sigma \in \pi_d$,

$$\sum_{i=k+1}^{s} B(i, \sigma(i)) - \sum_{i=k+1}^{s} C(i, \sigma(i))$$

gives a lower bound on the amount of $J_{k+1} \cup \cdots \cup J_s$ that must be done on $P_1, \cdots, P_k$ by time $\delta_{\sigma(k+1)}$ (and hence by time δ_d as $\sigma(k+1) \leqq d$). Hence.

$$Z(k, d) = \max_{\sigma \in \pi_d} \left\{ \sum_{i=k+1}^{s} B(i, \sigma(i)) - \sum_{i=k+1}^{s} C(i, \sigma(i)) \right\} \tag{2.2}$$

is also a lower bound on the amount of $J_{k+1} \cup \cdots \cup J_s$ that must be done on $P_1, \cdots, P_k$ by δ_d.

From (2.2) we may obtain a simple recurrence for $Z(k, d)$. Let $\sigma' \in \pi_d$ be the σ at which

$$\sum_{i=k+1}^{s} B(i, \sigma(i)) - \sum_{i=k+1}^{s} C(i, \sigma(i))$$

is maximum. Assume that $k < s$. If $\sigma'(k+1) \neq d$, then $Z(k, d) = Z(k, d-1)$. If $\sigma'(k+1) = d$, then

$$\begin{aligned} Z(k, d) &= \sum_{i=k+1}^{s} B(i, \sigma'(i)) - \sum_{i=k+1}^{s} C(i, \sigma'(i)) \\ &= \sum_{i=k+2}^{s} B(i, \sigma(i)) - \sum_{i=k+2}^{s} C(i, \sigma'(i)) + B(k+1, d) - C(k+1, d) \\ &= Z(k+1, d) + B(k+1, d) - C(k+1, d). \end{aligned}$$

This yields

$$Z(k, d) = \begin{cases} 0 & \text{if } k = s, \\ Z(k, 0) & \text{if } d = 0, \\ \max\{Z(k, d-1), Z(k+1, d) + B(k+1, d) - C(k+1, d)\} & \text{otherwise.} \end{cases} \tag{2.3}$$

Define $D(k, d)$ by

$$D(k, d) = \begin{cases} 0 & \text{if } d = 0, \\ C(k, 1) & \text{otherwise.} \end{cases}$$

When $d \neq 0$, $D(k, d)$ is nothing but the processing capacity of P_k from δ_0 to δ_1.

To arrive at a formula for $X(k, d)$, we note that for any i and d, $B(i, d) - A(i, d)$ is a lower bound on the amount of J_i's processing that must be done between δ_1 and δ_d. The processing capacity of P_i in this interval is $C(i, d) - D(i, d)$. So , for any profile function $\sigma \in \pi_d$,

$$\sum_{i=k+1}^{s} [B-A](i, \sigma(i)) - \sum_{i=k+1}^{s} [C-D](i, \sigma(i))$$

is a lower bound on the amount of $J_{k+1} \cup \cdots \cup J_s$ that must be done on $P_1, \cdots, P_k$ between δ_1 and δ_d. Consequently $X(k, d)$, $0 \leqq k \leqq s$, $0 \leqq d \leqq q$, as defined by

$$X(k, d) = \max_{\sigma \in \pi_d} \left\{ \sum_{i=k+1}^{s} [B - A - C + D](i, \sigma(i)) \right\} \tag{2.4}$$

is a lower bound on the amount of $J_{k+1} \cup \cdots \cup J_s$ that must be processed on $P_1, \cdots, P_k$ between δ_1 and δ_d.

From (2.4) the recurrence

$$X(k, d) = \begin{cases} 0 & \text{if } k = s, \\ X(k, 0) & \text{if } d = 0, \\ \max \{X(k, d-1), [X + B - A - C + D](k+1, d)\} & \text{otherwise} \end{cases} \tag{2.5}$$

may be obtained in the same way as (2.3) was obtained from (2.2).

Define

$$Y(k, d) = Z(k, d) - X(k, d), \qquad 0 \leqq k \leqq s, \quad 0 \leqq d \leqq q.$$

Some of the identities that we shall use in § 2.3 are stated below.

LEMMA 1. *If* $\sum_{i=1}^{s} B(i, \sigma(i)) \leqq \sum_{i=1}^{s} C(i, \sigma(i))$ *for every* $\sigma \in \pi$, *then*

(1a) $A(k, 0) = B(k, 0) = 0, \qquad 1 \leqq k \leqq s.$

(1b) $A(k, 1) = B(k, 1), \qquad 1 \leqq k \leqq s.$

(2a) $X(k, 1) = 0, \qquad 0 \leqq k \leqq s.$

(2b) $X(k, 0) = Y(k, 0) = Z(k, 0) = 0, \qquad 0 \leqq k \leqq s.$

(2c) $X(0, d) = Y(0, d) = Z(0, d) = 0, \qquad 0 \leqq d \leqq q.$

(3) *The functions a, b, A, B, C, D, X, Y and Z all have nonnegative values and are nondecreasing in the second variable*; *i.e.*,

$$0 \leqq f(k, d) \leqq f(k, d+1) \quad \textit{for } f = a, b, A, B, C, D, X, Y \textit{ and } Z.$$

Proof. See Appendix.

2.3. The algorithm. We are now ready to describe our preemptive scheduling algorithm. The jobs will be scheduled in q phases. In phase d we determine the amount of each job j that is to be scheduled from δ_{d-1} *to* δ_d. Once this amount has been determined the actual schedule from δ_{d-1} *to* δ_d is constructed using the Kafura–Shen algorithm.

```
line    PROCEDURE COMPUTE_W
        //w_j is the amount of job j to be processed from δ_0 to δ_1//
 1      R_0 → 0
 2      for k ← 1 to s do//consider jobs by classes//
 3         Q_k ← {d | A(k, d) + Y(k, d) ≦ R_{k-1} + C(k, 1)}
 4      if Q_k = φ then print ('infeasible job set')
 5                    stop endif
 6      h_k ← max {d | d ∈ Q_k}
 7      case
 8         :(1) h_k = q: w_j ← a(j, q), j ∈ J_k
 9         :(2) h_k < q, A(k, h_k + 1) + Y(k, h_k) ≧ R_{k-1} + C(k, 1):
10         set w_j for j ∈ J_k such that
```

```
11            a(j, h_k) ≦ w_j ≦ a(j, h_k + 1) and
12            Σ_{j∈J_k} w_j + Y(k, h_k) = R_{k-1} + C(k, 1)
13         :(3) else: w_j ← a(j, h_k + 1), j ∈ J_k
14      end case
15      R_k ← R_{k-1} + C(k, 1) − Σ_{j∈J_k} w_j
16      end for
17      end COMPUTE_W.
```

Procedure COMPUTE_W determines the amount w_j of job j that is to be scheduled from δ_0 *to* δ_1. The w_js are determined in a way such that $\{w_j: 1 \leqq j \leqq n\}$ can be scheduled from δ_0 to δ_1 and the remaining processing requirements $\{t_j - w_j: 1 \leqq j \leqq n\}$ can be feasibly scheduled from δ_1 to δ_q. Once the w_js are determined, the amount w_j of job j, $1 \leqq j \leqq n$, that is to be scheduled from δ_1 to δ_2 is determined by applying COMPUTE_W to $\{t_j - w_j: 1 \leqq j \leqq n\}$. Repeatedly applying COMPUTE_W in this way, one may successfully determine the w_js for each interval.

In procedure COMPUTE_W, R_k denotes the amount of idle time remaining on processor classes $1, 2, \cdots, k$ following the scheduling of the w_js corresponding to jobs in job classes $1, 2, \cdots, k$. (One may also think of R_k as the amount of processing time on processor classes $1, 2, \cdots, k$ that is to be reserved for jobs in job classes $k+1$, $k+2, \cdots, s$.) Roughly speaking, COMPUTE_W computes the w_js (job) class by class. In determining the w_js for job class k, the processing capacity available is equal to $R_{k-1} + C(k, 1)$. Initially, let $w_j \leftarrow a(j, 0)$ for $j \in J_k$ and $R_k \leftarrow Y(k, 0)$. If $\sum_{j\in J_k} w_j + R_k < R_{k-1} + C(k\ 1)$, then w_j, $j \in J_k$, is incremented to $a(j, 1)$ and R_k incremented to $Y(k, 1)$. If it is still the case that $\sum_{j\in J_k} w_j + R_k < R_{k-1} + C(k, 1)$, then w_j, $j \in J_k$ is incremented to $a(j, 2)$ and R_k to $Y(k, 2)$. This procedure continues until $R_{k-1} + C(k, 1)$ is used up (i.e., until $\sum_{j\in J_k} w_j + R_k = R_{k-1} + C(k, 1)$).

When actually implementing COMPUTE_W, the subscripts on h, R, and Q may be omitted. We have kept them in the version given so that we may easily refer to the values of h, R, and Q during different interations of the *for* loop. One should also note that in case (2), since $A(k, h_k + 1) + Y(k, h_k) \geqq R_{k-1} + C(k, 1)$ and $A(k, h_k) + Y(k, h_k) \leqq R_{k-1} + C(k, 1)$, there exist w_j, $a(j, h_k) \leqq w_j \leqq a(j, h_k + 1)$, such that

$$\sum_{j\in J_k} w_j + Y(k, h_k) = R_{k-1} + C(k, 1).$$

These w_js are easily determined by first setting all $w_j = a(j, h_k)$, $j \in J_k$, and then incrementing the w_js one by one (up to at most $a(j, h_k + 1)$) until the desired equality is satisfied.

2.4. Correctness and complexity. We now proceed to prove the correctness of the above algorithm and analyze its complexity. We have pointed out in § 2.1 that if there exists a feasible schedule, then $\sum_{i=1}^{s} B(i, \sigma(i)) \leqq \sum_{i=1}^{s} C(i, \sigma(i))$ for every $\sigma \in \pi$. We shall show in the following that if $\sum_{i=1}^{s} B(i, \sigma(i)) \leqq \sum_{i=1}^{s} C(i, \sigma(i))$ for every $\sigma \in \pi$, then the above algorithm generates a feasible schedule.

DEFINITION. For convenience in proving Lemmas 2 and 4, we arbitrarily define $Q_0 = \{0, 1\}$ and $h_0 = 1$.

We first show that if $\sum_{i=1}^{s} B(i, \sigma(i)) \leqq \sum_{i=1}^{s} C(i, \sigma(i))$ for every $\sigma \in \pi$, then procedure COMPUTE_W will not terminate in line 5.

LEMMA 2. *If* $\sum_{i=1}^{s} B(i, \sigma(i)) \leqq \sum_{i=1}^{s} C(i, \sigma(i))$ *for every* $\sigma \in \pi$, *then*

(1) $\qquad Q_k \neq \varnothing$ *and* $h_k \geqq 1, \quad 0 \leqq k \leqq s;$

(2) $\qquad R_k \geqq Y(k, h_k), \quad 0 \leqq k \leqq s;$

(3) $$R_k \leqq Y(k, h_k+1) \quad \text{if } h_k \neq q, \quad 0 \leqq k \leqq s.$$

Proof. We shall show (1), (2) and (3) by induction on k.

I.B. When $k=0$, $Q_0 \neq \phi$, $h_0=1$ and $R_0=Y(0, h_0)=Y(0, h_0+1)=0$ (either by definition or by Lemma 1).

I.H. Assume that (1), (2) and (3) are true for $k-1$ where $1 \leqq k-1 < s$.

I.S. We shall show that (1), (2) and (3) are true for k. To show (1), we see that

$$\begin{aligned} R_{k-1} &\geqq Y(k-1, h_{k-1}) && \text{(induction hypothesis)} \\ &\geqq Y(k-1, 1) && \text{(Lemma 1)} \\ &= Z(k-1, 1) && \text{(Lemma 1)} \\ &\geqq [Z+B-C](k, 1) && (2.3) \\ &= [Y+A-C](k, 1) && \text{(Lemma 1).} \end{aligned}$$

Hence, $1 \in Q_k$ and so $Q_k \neq \varnothing$ and $h_k \geqq 1$.

To prove (2) and (3), consider the three cases of COMPUTE_W (lines 7–14).

Case 1. In this case, $\sum w_j = A(k, h_k)$. From lines 3 and 6, we observe that $A(k, h_k) + Y(k, h_k) \leqq R_{k-1} + C(k, 1)$. Combining these two facts with the definition of R_k (line 15), we obtain $Y(k, h_k) \leqq R_k$.

Case 2. From lines 12 and 15, we obtain $R_k = Y(k, h_k) \leqq Y(k, h_k+1)$.

Case 3. In this case, $A(k, h_k+1) + Y(k, h_k) < R_{k-1} + C(k, 1) < A(k, h_k+1) + Y(k, h_k+1)$ and $\sum w_j = A(k, h_k+1)$. From these and line 15, we immediately obtain $Y(k, h_k) < R_k < Y(k, h_k+1)$. □

Before establishing the correctness of our scheme to compute the w_js, we obtain some relationships concerning the amount of processing t'_j of job j that remains to be done following time δ_1. Note that $t'_j = t_j - w_j$, $1 \leqq j \leqq n$.

DEFINITION. Define $b'(j, d)$, $a'(j, d)$, $B'(k, d)$, and $A'(k, d)$ to be the values obtained for b, a, B and A when t'_j is used in place of t_j. Let $C'(k, d) = |P_k|(\delta_d - \delta_1)$, $1 \leqq k \leqq s$, $1 \leqq d \leqq q$, and $W(k) = \sum_{j \in J_k} w'_j$, $1 \leqq k \leqq s$.

LEMMA 3. *If* $\sum_{i=1}^{s} B(i, \sigma(i)) \leqq \sum_{i=1}^{s} C(i, \sigma(i))$ *for every* $\sigma \in \pi$, *then*

(1) $$B'(k, d) \leqq B(k, d) - A(k, d), \qquad d \leqq h_k,$$

(2) $$B'(k, d) = B(k, d) - W(k), \qquad d > h_k \neq q.$$

Proof. It is easy to see that for any job j, $1 \leqq j \leqq n$,

$$b'(j, d) = \max\{0, b(j, d) - w_j\}.$$

When $d \leqq h_k$, from Lemma 1 and lines 8–13 of COMPUTE_W, we have $a(j, d) \leqq a(j, h_i) \leqq w_j$. Hence,

$$b'(j, d) = \max\{0, b(j, d) - w_j\} \leqq b(j, d) - a(j, d).$$

Hence,

$$B'(k, d) \leqq B(k, d) - A(k, d).$$

When $d > h_k \neq q$, from cases (2) and (3) of COMPUTE_W, Lemma 1 and the definition of $a(j, d)$, we see that

$$a(j, h_k) \leqq w_j \leqq a(j, h_k+1) \leqq a(j, d) \leqq b(j, d).$$

So,

$$b'(j, d) = \max\{0, b(j, d) - w_j\} = b(j, d) - w_j.$$

Hence,

$$B'(k, d) = B(k, d) - W(k). \qquad \square$$

LEMMA 4. *If* $\sum_{i=1}^{s} B(i, \sigma(i)) \leqq \sum_{i=1}^{s} C(i, \sigma(i))$ *for every* $\sigma \in \pi$, *then the following are true for every* k, $0 \leqq k \leqq s$ *and every* σ *such that* $\sigma(k) \geqq 1$:

$$(1) \qquad \sum_{i=1}^{k} B'(i, \sigma(i)) + Z(k, \sigma(k)) - R_k \leqq \sum_{i=1}^{k} C(i, \sigma(i)).$$

(2) *If* $\sigma(k) \leqq h_k$, *then*

$$\sum_{i=1}^{k} B'(i, \sigma(i)) + X(k, \sigma(k)) \leqq \sum_{i=1}^{k} C(i, \sigma(i)).$$

Proof. The proof is by induction on k.

I.B. When $k=0$, $\sum_{i=1}^{0} B'(i, \sigma(i)) = \sum_{i=1}^{0} C(i, \sigma(i)) = R_0 = 0$ by definitions, and $Z(0, \sigma(0)) = X(0, \sigma(0)) = 0$ by Lemma 1. Hence, (1) and (2) hold.

I.H. Assume that (1) and (2) are true for $k-1$, where $k-1$ is in the range $0 \leqq k-1 < s$.

I.S. We proceed to establish (1) and (2) for k by considering the three cases: (1) $\sigma(k) > h_k$, (2) $\sigma(k) \leqq h_k$ and $\sigma(k-1) \leqq h_{k-1}$, and (3) $\sigma(k) \leqq h_k$ and $\sigma(k-1) > h_{k-1}$.

Case 1. $\sigma(k) > h_k$. We first obtain the following

$$\sum_{i=1}^{k-1} B'(i, \sigma(i)) + Z(k-1, \sigma(k-1)) - R_{k-1} \leqq \sum_{i=1}^{k-1} C(i, \sigma(i)) \qquad \text{(I.H.)},$$

$$R_{k-1} - R_k = W(k) - C(k, 1) \qquad \text{(def. of } R_k\text{)},$$

$$B'(k, \sigma(k)) + W(k) = B(k, \sigma(k)) \qquad \text{(Lemma 3)},$$

$$\begin{aligned} &Z(k, \sigma(k)) + B(k, \sigma(k)) - C(k, \sigma(k)) \\ &\quad \leqq Z(k-1, \sigma(k)) && ((2.3) \text{ and } \sigma(k) \geqq 1) \\ &\quad \geqq Z(k-1, \sigma(k-1)) && (\sigma(k-1) \geqq \sigma(k) \text{ and } (2.3)). \end{aligned}$$

Adding these four equalities and inequalities yields

$$\sum_{i=1}^{k} B'(i, \sigma(i)) + Z(k, \sigma(k)) - R_k \leqq \sum_{i=1}^{k} C(i, \sigma(i)).$$

Case 2. $\sigma(k) \leqq h_k$ and $\sigma(k-1) \leqq h_{k-1}$. From the induction hypothesis, we have

$$(2.6) \qquad \sum_{i=1}^{k-1} B'(i, \sigma(i)) + X(k-1, \sigma(k-1)) \leqq \sum_{i=1}^{k-1} C(i, \sigma(i)).$$

From (2.5) and the fact that $\sigma(k-1) \geqq \sigma(k) \geqq 1$, we get

$$X(k-1, \sigma(k-1)) \geqq X(k-1, \sigma(k)) \geqq [X + B - A - C + D](k, \sigma(k)).$$

Using Lemma 3, this reduces to

$$X(k-1, \sigma(k-1)) \geqq X(k, \sigma(k)) + B'(k, \sigma(k)) - C(k, \sigma(k)).$$

Combining with (2.6) yields

$$(2.7) \qquad \sum_{i=1}^{k} B'(i, \sigma(i)) + X(k, \sigma(k)) \leqq \sum_{i=1}^{k} C(i, \sigma(i)).$$

Since $R_k \geqq Y(k, h_k) \geqq Y(k, \sigma(k))$ (Lemmas 2 and 1), we conclude that $Z(k, \sigma(k)) - R_k \leqq X(k, \sigma(k))$. Substituting into (2.7) yields

$$\sum_{i=1}^{k} B'(i, \sigma(i)) + Z(k, \sigma(k)) - R_k \leqq \sum_{i=1}^{k} C(i, \sigma(i)).$$

Case 3. $\sigma(k) \leqq h_k$ and $\sigma(k-1) > h_{k-1}$. From the induction hypothesis, we get

$$\sum_{i=1}^{k-1} B'(i, \sigma(i)) + Z(k-1, \sigma(k-1)) - R_{k-1} \leqq \sum_{i=1}^{k-1} C(i, \sigma(i)). \tag{2.8}$$

Since $h_{k-1} \neq q$, we obtain from Lemma 2

$$R_{k-1} \leqq Y(k-1, h_{k-1}+1) \leqq Y(k-1, \sigma(k-1)). \tag{2.9}$$

From Lemma 3, (2.5), and the inequality $\sigma(k-1) \geqq \sigma(k) \geqq 1$, we get

$$[X + B' - C](k, \sigma(k)) \leqq X(k-1, \sigma(k)) \leqq X(k-1, \sigma(k-1)). \tag{2.10}$$

Adding (2.8), (2.9) and (2.10) yields

$$\sum_{i=1}^{k} B'(i, \sigma(i)) + X(k, \sigma(k)) \leqq \sum_{i=1}^{k} C(i, \sigma(i)).$$

Using the same reasoning as in Case 2, we may now conclude the truth of (1) for k. □

THEOREM 1. *There exists a feasible preemptive schedule for the given n jobs if and only if*

$$\sum_{i=1}^{s} B(i, \sigma(i)) \leqq \sum_{i=1}^{s} C(i, \sigma(i)) \quad \textit{for every } \sigma \in \pi_q.$$

Proof. We have already pointed out that if a feasible schedule exists, then the above inequality is satisfied for every $\sigma \in \pi_q$. So, we need only show that when the above inequality is satisfied for every $\sigma \in \pi_q$, there is a feasible schedule. Assume that

$$\sum_{i=1}^{s} B(i, \sigma(i)) \leqq \sum_{i=1}^{s} C(i, \sigma(i)) \text{ for every } \sigma \in \pi_q. \tag{2.11}$$

From (2.11) it is clear that when $q = 1$, the t_js and (1.1) yield $f^* \leqq \delta_1 - \delta_0$, and so a feasible schedule exists.

For the induction hypothesis, we assume that there exists a feasible schedule when (2.11) is satisfied and $q = r$ for some r, $1 \leqq r$. We show that if (2.11) is satisfied when $q = r+1$, then there is a feasible schedule. From Lemma 2, we see that $Q_k \neq \phi$ for any k. Hence procedure COMPUTE_W successfully computes the w_js. Let $P(k) = \bigcup_{i=1}^{k} P_i$. It is clear from COMPUTE_W that $w_j \leqq a(j, d) \leqq \delta_1 - \delta_0$ where $d = q$ or $h_k + 1$ and that $\sum_{j \in J_1 \cup \cdots \cup J_k} w_j \leqq |P(k)|(\delta_1 - \delta_0) - R_k \leqq |P(k)|(\delta_1 - \delta_0)$ for every k. Hence, the w_js satisfy (1.1) (i.e., $f^* \leqq \delta_1 - \delta_0$) and may be scheduled from δ_0 to δ_1 using the Kafura–Shen algorithm.

Now, consider the t_j's. We know that $X(s, d) = Z(s, d) = 0$, $0 \leqq d \leqq r+1$. If $h_s < r+1$, then from Lemma 2 we obtain $0 \leqq R_s \leqq Y(s, h_s + 1) = 0$ or $R_s = 0$. Using this in Lemma 4 yields

$$\sum_{i=1}^{s} B'(i, \sigma(i)) \leqq \sum_{i=1}^{s} C(i, \sigma(i)) \quad \text{for every } \sigma \in \pi_{r+1} \text{ such that } \sigma(s) \geqq 1. \tag{2.12}$$

If $h_s = r+1$ then $\sigma(s) \leqq h_s$ and from Lemma 4 we once again obtain (2.12). One readily

sees that (2.12) is equivalent to

$$(2.13)\qquad \sum_{i=1}^{s} B'(i,\sigma(i)+1) \leqq \sum_{i=1}^{s} C(i,\sigma(i)+1) \quad \text{for every } \sigma\in\pi_r.$$

Following the scheduling from δ_0 to δ_1, we are left with the problem of scheduling the t_j's from δ_1 to δ_{r+1}. The number of distinct due times is now r (note that $t_j'=0$ for every j such that $d_j=\delta_1$). Relabel the start time δ_1 as δ_0' and the due times $\delta_2,\cdots,\delta_{r+1}$ as $\delta_1',\cdots,\delta_r'$. Define $b''(j,d)$, $B''(k,d)$, and $C''(k,d)$ to be the values obtained for b, B, and C when t_j' is used in place of t_j and δ_d' is used in place of δ_d. We immediately see that $B''(k,d)=B'(k,d+1)$, and $C''(k,d)=C'(k,d+1)$, for every k, d, $1\leqq k\leqq s$, $0\leqq d\leqq r$. Substituting into (2.13) yields

$$\sum_{i=1}^{s} B''(i,\sigma(i)) \leqq \sum_{i=1}^{s} C''(i,\sigma(i)) \quad \text{for every } \sigma\in\pi_r.$$

It now follows from the induction hypothesis that the t_j's can be scheduled. □

From Theorem 1, it is clear that by repeatedly using COMPUTE_W to determine the amount to be scheduled in each interval, a feasible schedule can be obtained whenever such a schedule exists. Each time COMPUTE_W is used, we need to recompute b, a, Z, X and Y. The time needed for this is $O(nq)$ (note that recurrences (2.3) and (2.5) will be used to compute Z and X). The *for* loop may be executed in $O(qs+n)$ time. We may assume that $s\leqq n$ and so the complexity of COMPUTE_W is $O(qn)$. The Kafura–Shen algorithm is of complexity $O(n)$. Hence, the overall computing time for the q phases of our scheduling algorithm is $O(q^2n)$. An additional $O(n\log n)$ time is needed to sort the jobs by memory size m_i. Hence, the overall complexity of our preemptive scheduling algorithm is $O(q^2n+n\log n)$. As for preemptions, since each job may be preempted at most twice in each interval $[\delta_d, \delta_{d+1}]$, $0\leqq i\leqq q-1$, the total number of preemptions is $O(nq)$.

3. Minimizing maximum lateness. Let S be a preemptive schedule for (t_j, d_j, m_j), $1\leqq j\leqq n$. Let f_j be the finish time of job j in S. If $f_j\leqq d_j$, $1\leqq j\leqq n$, then S is a feasible schedule and no job is late. The *lateness* of job j is f_j-d_j and the *maximum lateness* of the n jobs is $L_{\max}=\max\{f_j-d_j: 1\leqq j\leqq n\}$. Note that $L_{\max}\leqq 0$ if and only if all jobs finish by their due times. Also, note that if $L_{\max}\leqq 0$ then $\delta_0-\delta_1\leqq L_{\max}$.

From the definition of $L_{\max}$, it follows that by changing the release time from δ_0 to $\delta_0-L_{\max}$ we obtain a job set that can be scheduled such that no job finishes after its due time. Hence, to determine the minimum $L_{\max}$, we need to determine the least x such that the condition of Theorem 1 is satisfied when a release time of δ_0-x is used. This x may be obtained from a form equivalent to that of Theorem 1. We observe that $\sum_{i=1}^{s} B(i,\sigma(i))\leqq\sum_{i=1}^{s} C(i,\sigma(i))$ for every $\sigma\in\pi$ if and only if $\max_{\sigma\in\pi}\{\sum_{i=1}^{s} B(i,\sigma(i))-\sum_{i=1}^{s} C(i,\sigma(i))\}\leqq 0$. It is helpful to rewrite this form separating out the case when $\sigma(i)=0$, $1\leqq i\leqq s$. For this σ, we see that $\sum_{i=1}^{s} B(i,\sigma(i))-\sum_{i=1}^{s} C(i,\sigma(i))=\sum_{j=1}^{n} b(j,0)$. For every other σ, there is a k, $1\leqq k\leqq s$, such that $\sigma(k)\geqq 1$.

Define H_k by

$$H_k=\max_{\sigma\in\pi,\sigma(k)\geqq 1}\left\{\sum_{i=1}^{k} B(i,\sigma(i))-\sum_{i=1}^{k} C(i,\sigma(i))\right\},\qquad 1\leqq k\leqq s.$$

We immediately see that

$$\max_{\sigma\in\pi}\left\{\sum_{i=1}^{s} B(i,\sigma(i))-\sum_{i=1}^{s} C(i,\sigma(i))\right\}=\max\left\{\sum_{i=1}^{n} b(j,0), H_1, H_2,\cdots,H_s\right\}.$$

Let $x_1 = \max_{1 \leqq j \leqq n} \{t_j - d_j + \delta_0\}$ and let $x_2 = \max_{1 \leqq k \leqq s} \{H_k/|P(k)|\}$, where $|P(k)|$ is the cardinality of $P(k) = \bigcup_{i=1}^{k} P_i$.

Clearly, if we change the release time to $\delta_0 - \max\{x_1, x_2\}$ then $\max\{\sum b'(j, 0), H'_1, H'_2, \cdots, H'_s\} = 0$ (the b', H'_i values are computed with respect to the new release time $\delta_0 - \max\{x_1, x_2\}$). Hence, $\max_{\sigma \in \pi} \{\sum B'(i, \sigma(i)) - \sum C'(i, \sigma(i))\} \leqq 0$ and $\sum B'(i, \sigma(i)) \leqq \sum C'(i, \sigma(i))$ for every $\sigma \in \pi$. Moreover, $x = \max\{x_1, x_2\}$ is the least value of x for which this happens. Hence,

$$(L_{\max})_{\min} = \max\{x_1, x_2\}.$$

The H_ks may be computed in $O(qs)$ time as follows. Define H_k^d by

$$H_k^d = \max_{\substack{\sigma \in \pi \\ \sigma(k) \geqq d}} \left\{ \sum_{i=1}^{k} B(i, \sigma(i)) - \sum_{i=1}^{k} C(i, \sigma(i)) \right\}, \qquad 1 \leqq k \leqq s, \quad 1 \leqq d \leqq q.$$

Hence, $H_k = H_k^1$, $1 \leqq k \leqq s$. We immediately obtain the following recurrence for H_k^d:

$$H_k^d = \begin{cases} \max_{i \geqq d} \{B(1, i) - C(1, i)\} & \text{if } k = 1, \\ \sum_{i=1}^{s} B(i, q) - \sum_{i=1}^{s} C(i, q) & \text{if } d = q, \\ \max\{H_{k-1}^d + B(k, d) - C(k, d), H_k^{d+1}\} & \text{otherwise.} \end{cases}$$

Using this recurrence, all the H_k^ds may be obtained in $O(qs)$ time (excluding the time needed to determine the $b(j, d)$s, $B(k, d)$s etc.). The additional time needed to compute the $B(k, d)$s and $C(k, d)$s is $O(qn + n \log n)$ (assuming $n \geqq m$). Hence, the minimum $L_{\max}$ may be determined in $O(qn + n \log n)$ time. Having determined the minimum $L_{\max}$, a schedule having this $L_{\max}$ value can be obtained by changing δ_0 to $\delta_0 - (L_{\max})_{\min}$ and using procedure COMPUTE_W.

4. Conclusions. We have developed an $O(q^2 n + n \log n)$ algorithm to obtain a preemptive schedule for n jobs (t_j, d_j, m_j), $1 \leqq j \leqq n$, on m processors with given memory sizes. This schedule minimizes $L_{\max}$ and contains at most $O(qn)$ preemptions. The minimum value of $L_{\max}$ can itself be obtained in only $O(qn + n \log n)$ time.

Appendix. Proof of Lemma 1. Assume that

$$\text{(A.1)} \qquad \sum_{i=1}^{s} B(i, \sigma(i)) \leqq \sum_{i=1}^{s} C(i, \sigma(i))$$

for every $\sigma \in \pi$. Using $\sigma(i) = 0$, $1 \leqq i \leqq s$, in (A.1), we obtain $B(i, 0) = 0$ for $1 \leqq i \leqq s$. From this, we have $b(j, 0) = 0$, $1 \leqq j \leqq n$, and, therefore,

$$\text{(A.2)} \qquad t_j \leqq d_j - \delta_0, \qquad 1 \leqq j \leqq n.$$

From (A.1) and (A.2), it is easy to verify (1), (2) and (3) (except the case when $f = Y$) of Lemma 1.

The rest of this Appendix is devoted to proving (3) for $f = Y$; i.e.,

$$0 \leqq Y(k, d) \leqq Y(k, d+1), \qquad 0 \leqq k \leqq s, \quad 0 \leqq d \leqq q-1.$$

To prove this inequality, since Y is defined as $Z - X$, we need to know the relation between X and Z (see Claim 2 below). But X and Z are in turn defined through A and B, so we first establish the relationship concerning A and B:

CLAIM 1.

$$\frac{B(k,d)-A(k,d)}{\delta_d-\delta_1} \leqq \frac{B(k,d)}{\delta_d-\delta_0}, \qquad 1 \leqq k \leqq s, \quad 1 < d \leqq q.$$

Proof. Since $a(j,d)=\min\{b(j,d),\ \delta_1-\delta_0\}$, $a(j,d)=b(j,d)$ or $a(j,d)=\delta_1-\delta_0$. If $a(j,d)=b(j,d)$, then

$$\frac{\delta_1-\delta_0}{\delta_d-\delta_0} b(j,d) \leqq a(j,d)$$

as $\delta_1 \leqq \delta_d$. If $a(j,d)=\delta_1-\delta_0$, then since $t_j \leqq d_j-\delta_0$ implies $b(j,d) \leqq \delta_d-\delta_0$, we get

$$\frac{\delta_1-\delta_0}{\delta_d-\delta_0} b(j,d) \leqq \delta_1-\delta_0 = a(j,d).$$

So, in both cases we have

$$\frac{\delta_1-\delta_0}{\delta_d-\delta_0} b(j,d) \leqq a(j,d).$$

Hence,

$$\frac{\delta_1-\delta_0}{\delta_d-\delta_0} B(k,d) \leqq A(k,d) \quad \text{or} \quad \frac{\delta_1-\delta_0}{\delta_d-\delta_1} B(k,d) \leqq \frac{\delta_d-\delta_0}{\delta_d-\delta_1} A(k,d)$$

or

$$\frac{\delta_d-\delta_0}{\delta_d-\delta_1}[B(k,d)-A(k,d)] \leqq B(k,d) \quad \text{or} \quad \frac{B(k,d)-A(k,d)}{\delta_d-\delta_1} \leqq \frac{B(k,d)}{\delta_d-\delta_0}. \qquad \square$$

CLAIM 2.

$$(\delta_d-\delta_0)X(k,d) \leqq (\delta_d-\delta_1)Z(k,d), \qquad 0 \leqq k \leqq s, \quad 0 \leqq d \leqq q.$$

Proof. The proof is by induction on k and d.

I.B. on k. When $k=s$ and $0 \leqq d \leqq q$, $X(k,d)=Z(k,d)=0$.

I.H. on k. Assume that the inequality is correct when $0 \leqq k = k'+1 \leqq s$ and $0 \leqq d \leqq q$.

I.S. on k. When $k=k'$ and $0 \leqq d \leqq q$, the inequality may be shown correct by induction on d.

I.B. on d. When $d=0$ or 1, $X(k',d)=0$ and $Z(k',d) \geqq 0$.

I.H. on d. Assume that the inequality is correct when $q \geqq d = d'-1 \geqq 1$.

I.S. on d. We need to show the inequality is correct for $d=d'$ (and $k=k'$). From (2.5), we see that there are two possibilities for $X(k',d')$:

Case (i). $X(k',d')=X(k',d'-1)$. In this case,

$$\begin{aligned}(\delta_{d'-1}-\delta_0)X(k',d') &= (\delta_{d'-1}-\delta_0)X(k',d'-1) \\ &\leqq (\delta_{d'-1}-\delta_1)Z(k',d'-1) \qquad \text{(I.H. on } d) \\ &\leqq (\delta_{d'-1}-\delta_1)Z(k',d') \qquad \text{(by (2.3)).}\end{aligned}$$

Since $\delta_{d'-1}-\delta_0 > \delta_{d'-1}-\delta_1 \geqq 0$, we get

$$X(k',d') \leqq Z(k',d') \quad \text{or} \quad (\delta_{d'}-\delta_{d'-1})X(k',d') \leqq (\delta_{d'}-\delta_{d'-1})Z(k',d').$$

Adding this inequality to the previous one yields:

$$(\delta_{d'}-\delta_0)X(k',d') \leqq (\delta_{d'}-\delta_1)Z(k',d').$$

Case (ii). $X(k', d') = [X+B-A-C+D](k'+1, d')$. Now we obtain

$$\frac{X(k', d')}{\delta_{d'}-\delta_1} = \frac{[X+B-A-C+D](k'+1, d')}{\delta_{d'}-\delta_1}.$$

Using I.H. on k, Claim 1 and

$$\frac{C(k'+1, d')-D(k'+1, d')}{\delta_{d'}-\delta_1} = |P_{k'+1}| = \frac{C(k'+1, d')}{\delta_{d'}-\delta_0},$$

we obtain

$$\frac{X(k', d')}{\delta_{d'}-\delta_1} \leqq \frac{Z(k'+1, d')+B(k'+1, d')-C(k'+1, d')}{\delta_{d'}-\delta_0}$$

$$\leqq \frac{Z(k', d')}{\delta_{d'}-\delta_0} \qquad \text{(by (2.5))}. \qquad \square$$

We are now ready to show $Y(k, d) \geqq Y(k, d-1) \geqq 0$, $0 \leqq k \leqq s$, $1 \leqq d \leqq q$.

CLAIM 3. $Y(k, d) \geqq 0$, $0 \leqq k \leqq s$, $0 \leqq d \leqq q$.

Proof. For $k \geqq 1$, this follows from Claim 2 and the fact $\delta_k - \delta_0 > \delta_k - \delta_1 \geqq 0$. For $k = 0$, this follows from part (2b) of Lemma 1. $\square$

CLAIM 4. $Y(k, d) \geqq Y(k, d-1)$, $0 \leqq k \leqq s$, $1 \leqq d \leqq q$.

Proof. The proof is by induction on k.

I.B. When $k = s$ and $1 \leqq d \leqq q$, $Y(k, d) = Y(k, d-1) = 0$.

I.H. Assume that $Y(k, d) \geqq Y(k, d-1)$ for $1 \leqq k' < k \leqq s$ and $1 \leqq d \leqq q$.

I.S. We need to show that $Y(k', d) \geqq Y(k', d-1)$ for $1 \leqq d \leqq q$.

Let d be in the range $1 \leqq d \leqq q$.

Case (i). $Z(k', d) \geqq Z(k', d-1)$ and $X(k', d) = X(k', d-1)$. In this case, it is readily seen that $Y(k', d) \geqq Y(k', d-1)$.

Case (ii). $Z(k', d) = Z(k', d-1)$ and $X(k', d-1) < X(k', d) = [X+B-A-C+D](k'+1, d)$. This case is not possible. To see this, suppose that this case is possible. Let $k'' \geqq k'$ be the largest k'' which

(A.3) $\quad Z(k'', d) = Z(k'', d-1)$ and $X(k'', d-1) < [X+B-A-C+D](k''+1, d)$

for some d. Let d'' be the smallest d for which (A.3) holds. Note that $d'' > 0$. So

(A.4) $Z(k'', d'') = Z(k'', d''-1)$ and $X(k'', d''-1) < [X+B-A-C+D](k''+1, d'')$.

Since $X(k'', d''-1) \geqq 0$, it follows from (2.5) and (A.4) that $X(k'', d'') > 0$. Assume that

$$Z(k'', d'') = Z(k'', d''-1) = Z(k'', d''-2) = \cdots = Z(k'', d^*) \neq Z(k'', d^*-1).$$

Then it follows from our choice of k'' and d'' that

$$X(k'', d''-1) = X(k'', d''-2) = \cdots = X(k'', d^*).$$

If $d^* = 0$, then $0 = Z(k'', d^*) = Z(k'', d'') \geqq X(k'', d'') > 0$. Hence, $d^* \neq 0$. Now, from the choice of d^*, we get

$$\begin{aligned} Y(k'', d^*) &= Z(k'', d^*) - X(k'', d^*) \\ &\leqq [Z+B-C](k''+1, d^*) \\ &\quad -[X+B-A-C+D](k''+1, d^*) \\ &= [Y+A-D](k''+1, d^*). \end{aligned} \tag{A.5}$$

Also,

$$X(k'', d''-1) < [X+B-C-A+D](k''+1, d'')$$

and

$$Z(k'', d''-1) \geqq [Z+B-C](k''+1, d'').$$

So

$$Y(k'', d^*) = Y(k'', d''-1) > [Y+A-D](k''+1, d'').$$

Substituting into (A.5) yields

$$[Y+A-D](k''+1, d^*) > [Y+A-D](k''+1, d'')$$

or

$$\begin{aligned} Y(k''+1, d^*) &> [Y+A-D](k''+1, d'') - [A-D](k''+1, d^*) \\ &\geqq Y(k''+1, d'') \qquad \text{(Lemma 1 and definition of } D). \end{aligned}$$

But $k''+1 > k'$ and so from I.H. it follows that

$$Y(k''+1, d'') \geqq Y(k''+1, d''-1) \geqq \cdots \geqq Y(k''+1, d^*).$$

So Case (ii) is not possible.

Case (iii). $Z(k', d) = [Z+B-C](k'+1, d)$ and $X(k', d) = [X+B-A-C+D](k'+1, d)$. Now $Y(k', d) = [Y+A-D](k'+1, d)$. Suppose that

$$\text{(A.6)} \qquad Z(k', d-1) = Z(k', d-2) = \cdots = Z(k', d^*) \neq Z(k', d^*-1).$$

From the proof of Case (ii), it follows that $X(k', d-1) = X(k', d-2) = \cdots = X(k', d^*)$. So $Y(k', d-1) = Y(k', d^*)$. If $d^* = 0$, then $Y(k', d-1) = Y(k', 0) = 0 \leqq Y(k', d)$. If $d^* \neq 0$, then

$$\begin{aligned} Y(k', d-1) &= Y(k', d^*) \\ &\leqq [Y+A-D](k'+1, d^*) && \text{(by (A.6), (2.3), (2.5))} \\ &\leqq Y(k'+1, d) + [A-D](k'+1, d^*) && \text{(I.H.)} \\ &\leqq [Y+A-D](k'+1, d) && \text{(Lemma 1)} \\ &\equiv Y(k', d). \end{aligned}$$

□

REFERENCES

[1]. M. GAREY AND D. JOHNSON, *Computers and Intractability, A Guide to the Theory of* NP-*Completeness*, W. H. Freeman, San Francisco, 1979.

[2] D. KAFURA AND V. SHEN, *Task scheduling on a multiprocessor system with independent memories*, this Journal, 6 (1977), pp. 167–187.

[3] T. LAI AND S. SAHNI, *Preemptive scheduling of a multiprocessor system with memories to minimize maximum lateness*, Tech. Report 81-20, Computer Science Department, Univ. of Minnesota, Minneapolis, MN, 1981.

[4] R. MCNAUGHTON, *Scheduling with deadlines and loss functions*, Management Sci., 6 (1959), pp. 1–12.

[5] S. SAHNI, *Preemptive scheduling with due dates*, Oper. Res., 27 (1979), pp. 925–934.

[6] S. SAHNI AND Y. CHO, *Scheduling independent tasks with due times on a uniform processor system*, J. Assoc. Comput. Mach., 27 (1980), pp. 550–568.

[7] ———, *Nearly on line scheduling of a uniform processor system with release times*, this Journal, 8 (1979), pp. 275–285.

[8] E. COFFMAN, JR., *Computer and Job Shop Scheduling Theory*, John Wiley, New York, 1976.

SIAM J. COMPUT.
Vol. 13, No. 4, November 1984

© 1984 Society for Industrial and Applied Mathematics
004

SCHEDULING INDEPENDENT TASKS ON UNIFORM PROCESSORS*

GREGORY DOBSON†

Abstract. A worst-case analysis is given for the LPT (longest processing time) heuristic applied to the problem of scheduling independent tasks on uniform processors. A bound of $\frac{19}{12}$ is derived on the ratio of the heuristic to the optimal makespan and an example is given where the error is greater than $\frac{3}{2}$. A generalization of the classic result of Graham for the case of identical processors is given. Here tight bounds are derived for the ratio of the heuristic to the optimal makespan which depends on the ratio of the longest task to the makespan.

Key words. scheduling independent tasks, uniform processors, finish time, approximate algorithm

1. Introduction. This paper addresses the problem of scheduling n independent tasks, T, with processing times $t_1 \geqq \cdots \geqq t_n$ on m processors with speeds $s_1 \geqq \cdots \geqq s_m$. The objective is to find a schedule, i.e. a partition $P = (P_1, \cdots, P_m)$, of the tasks which minimizes the maximum finishing time. Formally,

$$z^* \equiv \underset{P}{\text{minimum}} \underset{1 \leqq i \leqq m}{\text{maximum}} \frac{t(P_i)}{s_i}$$

where $t(P_i) = \sum_{j \in P_i} t_j$ and the minimization is over all partitions of P of T.

This problem is well known to be NP-hard since it contains 3-Partition (see Garey and Johnson (1979)). To see this observe that if N were an instance of 3-Partition with items of weight $t_1, \cdots, t_n$ and there were m bins of size $s_1 = \cdots = s_m = K$, a constant, then a packing of the items into the bins exists if and only if $z^* \leqq K$.

The paper considers the heuristic which schedules the tasks, in order of processing time, placing the longest remaining task on the processor which would complete that task first. Ties are broken arbitrarily. The heuristic is commonly referred to as the LPT (longest processing time) heuristic in the literature. Let $\bar{P} = (\bar{P}_1, \cdots, \bar{P}_m)$ be the partition given by the LPT heuristic and define

$$\bar{z} \equiv \max_{1 \leqq i \leqq m} t(\bar{P}_i).$$

There are numerous other results for this and similar problems where there is a precedence structure on the tasks. The reader is referred to Kunde (1981) and Graham (1969) for results where there is a precedence constraint. For the problem where the tasks are independent, Graham (1969) gave the first analysis of the case where the processors have identical speeds. In Coffman et al. (1978) another heuristic for this problem, called MULTIFIT, is given which uses bin packing to give a bound of 1.220. Special cases of the problem that are solved optimally by this heuristic are discussed in Coffman et al. (1977) and a slight generalization of some of their results is given in Dobson (1981). Recently Friesen (1984) has improved the bound on the MULTIFIT heuristic to 1.2 and Friesen and Langston have shown that a variation on the MULTIFIT heuristic gives a bound of 1.4 when applied to the problem with uniform processors.

* Received by the editors December 7, 1982, and in revised form July 22, 1983. This research was partially supported by Department of Energy Contract DE-AC03-76SF00326, PA# DE-AT03-76ER72018, Office of Naval Research contract N00014-75-C-0267, and National Science Foundation grants MCS76-81259, MCS-79260009 and ENG77-06761.

† Graduate School of Management, University of Rochester, Rochester, New York 14627.

The previous best known bound for the LPT heuristic was $\bar{z}/z^* \leqq 2$ and worst known examples had $\bar{z}/z^* \geqq \frac{3}{2} - \varepsilon$ for arbitrary $\varepsilon > 0$ (Gonzalez, Ibarra and Sahni (1977)) (Ibarra and Kim (1977)). The theorem in § 4 shows that $\bar{z}/z^* \leqq \frac{19}{12}$ and gives an example where there is an error greater than $\frac{3}{2}$. In the case where the processors all have identical speeds, $s_1 = \cdots = s_m$, Graham (1969) showed that $\bar{z}/z^* \leqq \frac{4}{3} - (1/3m)$. The results in §§ 2 and 3 generalize this for the case where the task lengths are $t_j \leqq (1/k)z^*$ for some integer $k \geqq 2$. The bounds are $(k+3)/(k+2)$ and $(k+2)/(k+1) - 1/(m(k+1))$.

We have often found it easier to discuss this problem in the language of bin packing rather than that of multiprocessor scheduling. Here items take the place of tasks, size of an item for processing time of a task, bins for processors, size of bin for speed of processor, level of bin for finishing time of processor. The problem is then to pack the n items into m bins in order to minimize the maximum level in any bin. This terminology will be used freely throughout the rest of this paper.

2. The case of identical processors. This section considers the special case of the scheduling problem where the processors' speeds are identical. Graham's result for the LPT heuristic, ignoring the term for the number of processors, gives a bound of $\frac{4}{3}$. One would suspect that the worst-case error, $\bar{z}/z^*$, would be smaller if all the processing times of the tasks were relatively small.

THEOREM 2.1. *If for all $j \in T$, $t_j \in [0, (1/k)z^*]$ where k is a positive integer and if the* LPT *heuristic gives a partition $\bar{P}$ with value $\bar{z}$ then*

$$\frac{\bar{z}}{z^*} \leqq \frac{k+3}{k+2}$$

and the bound is tight.

Note that $t_j \leqq z^*$ so that $k \geqq 1$ always holds and thus the ratio $\bar{z}/z^* \leqq \frac{4}{3}$. Thus the theorem reduces to Graham's when $k=1$. To apply the theorem to obtain a better than $\frac{4}{3}$ bound requires prior knowledge of z^*. Note though, that $(k'+3)/(k'+2)$ overestimates the ratio $(k+3)/(k+2)$ when $k' = \lfloor (t(T)/m) \max_{j \in T} (1/t_j) \rfloor$ since $t(T)/m$ is a lower bound on z^*.

Scaling. We may assume without loss of generality that $z^* = 1$. By scaling each weight by $1/z^*$ (i.e. $t_j \leftarrow t_j/z^*$) we have the new scaled value of $z^* = 1$ and $0 \leqq t_j \leqq 1$ for all $j \in T$.

Proof of Theorem 2.1. The proof is by contradiction. Assume there is an example in which the heuristic places an item in a bin and the resulting level exceeds $(k+3)/(k+2)$. Assume this is the example with the fewest number of items. The first item to exceed level $(k+3)/(k+2)$ in some bin is item n. If this were not the case then item n could be removed producing a smaller counterexample.

First observe that $t_n \notin [0, 1/(k+2)]$. To see this observe that if $t_n \in [0, 1/(k+2)]$ and if it were placed in bin i to give a level greater than $(k+3)/(k+2)$ then the level in bin i (as well as every other bin) must be greater than 1. Thus $t(T) > t(\{1, \cdots, n-1\}) \geqq m$, contradicting $z^* = 1$. Second, if $t_n \in (1/(k+1), 1/k]$ and it overfills a bin to a level $(k+3)/(k+2)$, then the level in every bin must be at least

$$\frac{k+3}{k+2} - \frac{1}{k} > 1 - \frac{1}{k} = \frac{k-1}{k}.$$

Every bin must have k items in it. Certainly no bin can hold more than k such items and have a level under z^*. Hence there is no way to pack items T so that the maximum level is at most z^*, contradicting the definition of z^*.

It remains to show that $t_n \notin (1/(k+2), 1/(k+1)]$.

Case $k=1$. Here we assume item n is the first item to exceed level z^*. If $t_n \in (\frac{1}{3}, \frac{1}{2}]$ then there are either one or two items in each bin (after packing $\{1, \cdots, n-1\}$). Say the heuristic places item n in bin i. If there is only one item, k, in the bin i, so $t_n + t_k > 1$, then clearly $t_l + t_k > 1$ for every $l \in T$. Hence any optimal packing of T must have item k in a bin by itself. Since the heuristic placed item n in the most empty bin, all other items stored one to a bin are larger than k, they too must be by themselves in any optimal packing of T. Every other bin has 2 items in it. If n is to be placed without exceeding the z^* level, then 3 items must go in some bin but $t_j > \frac{1}{3}$ for all j, a contradiction. □

Case $k=2$. Item n is the first to exceed level $\frac{5}{4}$. If $t_n \in (\frac{1}{4}, \frac{1}{3}]$ then we can write $t_n = \frac{1}{3} - \varepsilon$ where $0 \leqq \varepsilon < \frac{1}{12}$. Each bin is filled to a level at least $\frac{5}{4} - \frac{1}{3} + \varepsilon = \frac{11}{12} + \varepsilon$. There are at least 2 other items in a bin. If there are 2 then the larger one weighs at most $\frac{1}{2}$ so the smaller, r, weighs at least $\frac{5}{12} + \varepsilon$. Item r can only be packed in a bin with 2 items in an optimal packing since if we compute the total weight of item r and the 2 two smallest items we have

$$t_r + t_{n-1} + t_n \geqq \tfrac{5}{12} + \varepsilon + 2(\tfrac{1}{3} - \varepsilon) = \tfrac{13}{12} - \varepsilon > 1.$$

Here every item that is packed in a bin with 2 items can only fit in a bin with 2 items in an optimal packing of T. Every bin can only take 3 so there is at least one too many items. □

Case $k \geqq 3$. We assume that $t_n \in (1/(k+2), 1/(k+1)]$ and that n is the first item to exceed a level of at least $(k+3)/(k+2)$. We will show that the items T could not be packed in m bins of size 1.

CLAIM 1. *Let* $t_n = 1/(k+1) - \varepsilon$ *where* $0 \leqq \varepsilon < 1/((k+2)(k-1))$. *The level in the most empty bin is at least*

$$1 - \frac{1}{(k+1)(k+2)} + \varepsilon.$$

Proof.

$$\frac{k+3}{k+2} - \frac{1}{k+1} + \varepsilon = \frac{(k+3)(k+1) - (k+2)}{(k+1)(k+2)} + \varepsilon$$

$$= \frac{k^2 + 3k + 2 - 1}{(k+1)(k+2)} + \varepsilon = 1 - \frac{1}{(k+1)(k+2)} + \varepsilon. \qquad \square$$

CLAIM 2. *Every bin has* k, $k+1$ *or* $k+2$ *items in it.*

Proof. If some bin had $k+3$ items then since each item weighs strictly more than $1/(k+2)$, the current level would be strictly greater than $(k+3)/(k+2)$, contradicting the definition of n being the first item to exceed that level. If it had $k-1$ items then the current level would be at most

$$\frac{k-1}{k} < 1 - \frac{1}{(k+1)(k+2)} + \varepsilon,$$

contradicting Claim 1. □

CLAIM 3. *There is at least one bin*, r, *with* k *elements.*

Proof. If this were not the case then every bin has at least $k+1$ items and there would be at least $(k+1)m+1$ items all of weight greater than $1/(k+2)$, but only $k+1$ of these items could possibly fit in a bin of capacity $z^* = 1$. This contradicts the definition of z^*. □

CLAIM 4. *If $x_1 \geqq \cdots \geqq x_k$ are weights of items in bin r, the one with k items, then*

$$x_k > \frac{1}{k+1} + \varepsilon.$$

Proof.

$$\begin{aligned} x_k &\geqq 1 - \frac{1}{(k+1)(k+2)} + \varepsilon - x_1 - \cdots - x_{k-1} \\ &\geqq 1 - \frac{1}{(k+1)(k+2)} + \varepsilon - \frac{k-1}{k} \\ &= \frac{(k+1)(k+2)-k}{k(k+1)(k+2)} + \varepsilon = \frac{k^2+2k+2}{k(k+1)(k+2)} + \varepsilon > \frac{1}{k+1} + \varepsilon. \end{aligned}$$ □

CLAIM 5. *Some bin contains at least $k+1$ items.*

Proof. Otherwise every bin would contain exactly k items each at least of size $1/(k+1)+\varepsilon$. Including item n there would be $km+1$ items. At least $k+1$ would have to go in some bin, but

$$k\left(\frac{1}{k+1} + \varepsilon\right) + \left(\frac{1}{k+1} - \varepsilon\right) \geqq 1 + \varepsilon,$$

contradicting the definition of $z^* = 1$. □

CLAIM 6. *Let $y_1 \geqq \cdots \geqq y_{k+1}$ be the weights of items in a bin q containing $k+1$ items. Then*

$$y_{k-1} > \frac{1}{k+1} + \varepsilon.$$

Proof. If not, then by Claim 4 and the fact that the items are packed by decreasing weight, the $(k-1)$st item was placed after the kth item in bin r (the one with k items). Thus the level in the q bin is at least

$$\begin{aligned} x_1 + \cdots + x_{k-1} + y_{k-1} + y_k + y_{k+1} &\geqq 1 - \frac{1}{(k+1)(k+2)} + \varepsilon - \frac{1}{k} + 3\left(\frac{1}{k+1} - \varepsilon\right) \\ &= 1 + \frac{3}{k+1} - \frac{1}{k} - \frac{1}{(k+1)(k+2)} - 2\varepsilon \\ &> 1 + \frac{3k-(k+1)}{k(k+1)} - \frac{3}{(k+1)(k+2)} \\ &= 1 + \frac{(2k-1)(k+2)-3k}{k(k+1)(k+2)} \\ &= 1 + \left(\frac{1}{k+2}\right)\left(\frac{2k^2-2}{k(k+1)}\right) \geqq \frac{k+3}{k+2}, \end{aligned}$$

since

$$\begin{aligned} \frac{2k^2-2}{k(k+1)} \geqq 1 \quad &\text{iff} \quad 2k^2 - 2 - k(k+1) \geqq 0 \\ &\text{iff} \quad k^2 - k - 2 \geqq 0 \\ &\text{iff} \quad k(k-1) - 2 \geqq 0 \\ &\text{iff} \quad k \geqq 2 \end{aligned}$$

contradicting that n was the first to exceed the level $(k+3)/(k+2)$. □

It is clear that if bin q had had $k+2$ items of weights $y_1 \geqq \cdots \geqq y_{k+2}$ then

$$y_{k-1} > \frac{1}{k+1} + \varepsilon.$$

CLAIM 7. *There are no bins with $k+2$ items in an optimal solution.*

Proof. Assume there were; then the level in such a bin would be

$$(k+2)\left(\frac{1}{k+1} - \varepsilon\right) = 1 + \frac{1}{k+1} - (k+2)\varepsilon > 1. \qquad \square$$

CLAIM 8. *Let u be the number of bins with k items. Let s be the number of bins with at least $k+1$ items. Then we have by the previous claims that there are at least $ku+(k-1)s$ "large" items which weigh strictly more than $1/(k+1)+\varepsilon$, and at least $2s+1$ "small" items which weigh at least $1/(k+1)-\varepsilon$. The claim is that any repacking of the bins so that some bin q has $k+1$ items and a level of at most $z^*=1$ must have at least 3 small items.*

Proof. Assume you can do it with only 2. The level in such a bin with $k-1$ large and 2 small items is strictly greater than

$$(k-1)\left(\frac{1}{k+1} + \varepsilon\right) + 2\left(\frac{1}{k+1} - \varepsilon\right) = 1 + (k-3)\varepsilon \geqq 1,$$

provided $k \geqq 3$. $\square$

To finish the proof of the upper bound in Theorem 2.1 observe by Claim 7 that at least $s+1$ bins must have $k+1$ items. The only way to place $k+1$ items into bins of capacity 1 is to have at least 3 small items. There are at most $3s+1$ small items available. We can create at most $\lfloor (3s+1)/3 \rfloor < s+1$ bins with $k+1$ items by claim 8. This contradiction of the definition of z^* shows that $t_n \notin ((1/k+2), 1/(k+1)]$.

To see that the bound is tight consider the following examples parameterized by $k=1,2,3,\cdots$. We will produce an example that has an error of at least $(k+3)/(k+2)-\delta$. Let p be large enough so that $\delta \equiv (1/p)(1/(k+1)-1/(k+2))$ is "small" enough. We defined δ so that $1/(k+2)+p\delta = 1/(k+1)$. The list of items is

$k+1$	of weight $1/(k+2)+(2p-1)\delta$,
$\vdots$	$\vdots$ $\vdots$
$k+1$	of weight $1/(k+2)+(p+1)\delta$,
$k+1$	of weight $1/(k+2)+p\delta$,
$(k-1)(k+1)$	of weight $1/(k+2)+p\delta = 1/(k+1)$,
$k+1$	of weight $1/(k+2)+(p-1)\delta$,
$\vdots$	$\vdots$ $\vdots$
$k+1$	of weight $1/(k+2)+1\delta$,
$k+2$	of weight $1/(k+2)+0\delta$.

There are $(k+1)p$ bins. The LPT heuristic places the items in the bins so that there are $k+1$ identical sets of p bins as displayed in Fig. 1. Here $\alpha = 1/(k+2)$ and each bin contains $k+1$ items. The $k-1$ middle ones are all of weight $1/(k+2)+p\delta = 1/(k+1)$.

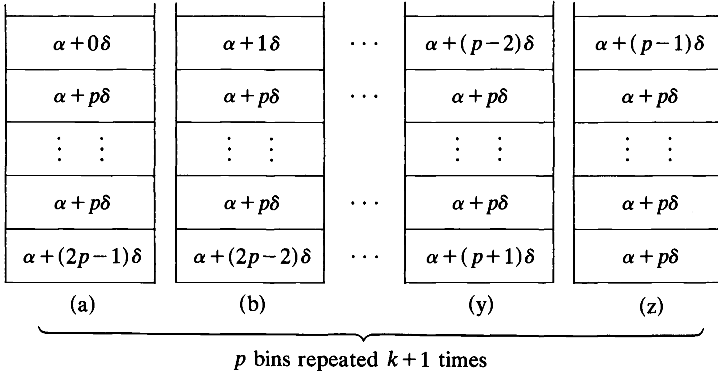


FIG. 1.

The $((k+1)^2p+1)$st item of weight $1/(k+2)$ is not shown. Observe that each bin is filled to a level $1-\delta$. To see that all the items can be stored in $(k+1)p$ bins of capacity 1 we move the item of weight $1/(k+2)+0\delta$ out of bin (a) and place it aside. Next move the item which weighs $1/(k+2)+1\delta$ out of bin (b) and into bin (a). Move the item which weighs $1/(k+2)+2\delta$ out of bin (c) and into bin (b) and so on. At this point all the bins labeled (a), (b), $\cdots$, (y) are filled to level 1, there are $k+2$ items of weight $1/(k+2)$ on the side and all the (z) bins have k items of weight $1/(k+2)+p\delta=1/(k+1)$. It is now possible to place all the items of weight $1/(k+2)$ in one bin and all the remaining items of weight $1/(k+1)$ (there are $k(k+1)$ of them) in the remaining k bins. The new packing looks like Fig. 2. The remaining bins labelled (z) look like Fig. 3.

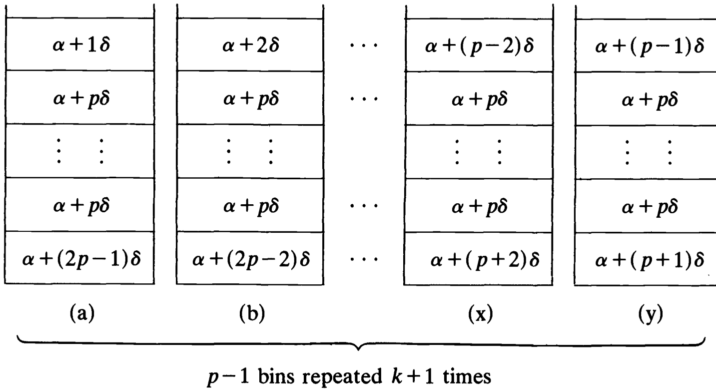


FIG. 2.

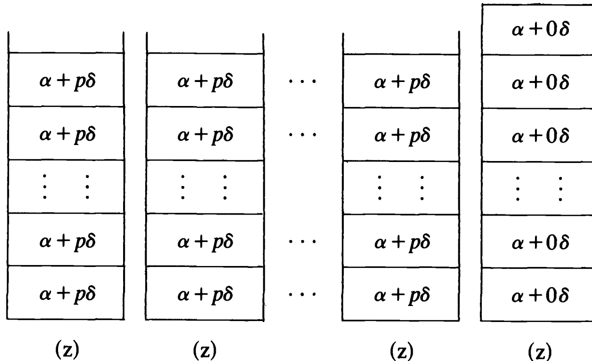


FIG. 3.

3. A bound for a small number of processors. Apparently an improvement of the bound analogous to Graham's (1969) result is not possible. One might conjecture a bound of

$$\frac{k+3}{k+2}-\frac{1}{m(k+2)},$$

but the example with $m=2$ bins, k even, k items of size $1/k$ and $k+1$ items of size $1/(k+1)$, gives a heuristic solution value of

$$\frac{k}{2}\frac{1}{k}+\frac{k+2}{2}\frac{1}{k+1}=1+\frac{1}{2(k+1)}>1+\frac{1}{2(k+2)}=\frac{k+3}{k+2}-\frac{1}{2(k+2)}$$

for every k. The optimal solution has a value of 1. For small m, a slightly tighter bound that can be proved is

THEOREM 3.1. *Let $k \geqq 2$. If for all $j \in T$, $t_j \in [0, (1/k)z^*]$, and if the* LPT *heuristic gives a partition $\bar{P}$ with value $\bar{z}$ then*

$$\frac{\bar{z}}{z^*}\leqq\frac{k+2}{k+1}-\frac{1}{m(k+1)}.$$

Proof. As in the proof of Theorem 2.1 we assume the result is not true and thus there must be a smallest counterexample. First using a calculation analogous to Graham (1969) we show that $t_n > 1/(k+1)$. To see this observe that item n was placed in a bin whose level was currently at most

$$\bar{z}-t_n \leqq \frac{1}{m}\sum_{i=1}^{n-1} t_i$$

since it was placed in the most empty bin. Thus

$$\bar{z}\leqq\frac{1}{m}\sum_{i=1}^{n} t_i+\frac{m-1}{m}t_n.$$

Since the example at hand exceeds the alleged bound

$$\frac{1}{k+1}-\frac{1}{m(k+1)}<\frac{\bar{z}-z^*}{z^*}\leqq\frac{1}{z^*}\frac{m-1}{m}t_n,$$

which implies

$$m-1<(m-1)(k+1)\frac{t_n}{z^*}$$

or

$$t_n>\frac{1}{k+1}z^*.$$

Thus, $t_n \in (1/(k+1), 1/k]$. Before item n is placed the level of every bin must be at least

$$\frac{k+2}{k+1}-\frac{1}{m(k+1)}-\frac{1}{k}>1-\frac{1}{k}.$$

Since the weight of every item is at most $1/k$ there are at least k items in every bin. At most k can fit since $t_n > 1/(k+1)$ and thus there is no way to store the $mk+1$ items in the m bins of size 1, a contradiction. □

4. Different speed processors. This section considers the problem of scheduling independent tasks on uniform processors with different speeds $s_1, \cdots, s_m$. The heuristic considered orders the tasks by time, i.e. $t_1 \geqq \cdots \geqq t_n$, then starting with an empty partition it repeatedly executes the following for $j = 1, \cdots, n$.

1) Compute

$$\frac{t(P_r)+t_j}{s_r} = \min_{1 \leqq i \leqq m} \frac{t(P_i)+t_j}{s_i}$$

where $P = (P_1, \cdots, P_m)$ is the current partition of the tasks $\{1, \cdots, j-1\}$.

2) Add item j to P_r.

The following guarantee exists on the performance of the heuristic.

THEOREM 4.1. *Assume $\bar{P}$ is the partition given by the heuristic with a value $\bar{z} = \max_{1 \leqq i \leqq m} t(\bar{P}_i)/s_i$ and P^* is an optimal partition with a value z^* then*

$$\frac{\bar{z}}{z^*} \leqq \frac{19}{12}.$$

Proof. Without loss of generality we may scale the task times so that $t_n = 1$ and the processors speeds so that $z^* = 1$. Assume to the contrary that there is an example with $\bar{z}/z^* > \frac{19}{12}$. As in the proof of Theorem 2.1 assume that this is the smallest example, that is (a) it is the one with the fewest number of bins, (b) for the example with the fewest number of bins it has the least number of items and (c) that the first item to exceed a level of $\frac{19}{12}$ is item n. It is possible to make several observations about this example. In what follows the partition $\bar{P}$ will refer to the partition produced by the heuristic by packing only items $\{1, \cdots, n-1\}$.

LEMMA 4.2. *$s_i \geqq 1$ for all i.*

Proof. If not then for some i, $t_n > s_i$. No item may be placed in bin i in an optimal packing since $z^* = 1$. Removal of this bin will create a smaller counterexample. □

LEMMA 4.3. *If $s_i > \frac{12}{7}$ then $t(\bar{P}_i)/s_i > 1$.*

This says that if a bin is large enough then in any counterexample it must be overfilled.

Proof. If item n were placed in bin i then

$$\frac{t(\bar{P}_i)+t_n}{s_i} > \frac{19}{12},$$

so

$$\frac{t(\bar{P}_i)}{s_i} > \frac{19}{12} - \frac{1}{s_i} > 1$$

if and only if $s_i > \frac{12}{7}$. □

The bins will be classified as either "large" if $s_i > \frac{12}{7}$ or "small" if $s_i \leqq \frac{12}{7}$. Let L be the set of large bins. The next lemma shows that the average size of the large bins cannot be "too large" in any counterexample to the theorem.

LEMMA 4.4. *Let $\bar{s}$ be the average size of the large bins, i.e. $\sum_{i \in L} s_i/|L|$. Then $\bar{s} < 6$.*

Proof. First observe that since every large bin is overfilled, $t(\bar{P}_i)/s_i > 1$, the only way the set of items in L can be packed so that $z^* = 1$ is of the items with sizes less than $\frac{12}{7}$ are placed in small bins and the smaller items displaced from the small bins are returned to the large bins. The maximum reduction that can occur is from an item

of size $\frac{12}{7}$ to one of size 1. This is possible if the following situation occurred:

processor:	12/7	12/7−ε	12/7−2ε	⋯	1
processor number:	i	$i+1$	$i+2$	⋯	m
speed:		12/7	12/7−ε	⋯	1+ε

and every item is moved over one slot. One cannot do better than this since every item is at least as large as $t_n = 1$.

Second, observe that small bins contain at most one item. To see this note that the size of every item is at least t_n. If there were two items in a small bin then the bin and its contents could be removed resulting in a smaller counterexample. Third, if an item, say p, is packed into a large bin with the result that once item p is included the bin becomes overfull, $t(P_i) > s_i$, then that item, p, cannot be replaced with a smaller one by exchanging it with any item that was originally placed by the heuristic in a small bin. To see this, assume item p is as above and a smaller item q is in a small bin, say j. This implies that $t_q < t_p$ and thus $p < q$ which in turn implies that bin j was empty when p was placed and it would have been placed in bin j, a contradiction.

Finally, the total capacity available in large bins is $\sum_{i\in L} s_i$. Every item placed below level 1 of size less than $\frac{12}{7}$ may at best be reduced to an item of size 1. All items placed above level 1 cannot be reduced. Thus a necessary condition for our example to have a heuristic value exceeding $\frac{19}{12}$ and yet still have an optimal solution value of 1, is

$$\sum_{i\in L}\left(\frac{7}{12}s_i - 1\right) + \sum_{i\in L}\frac{7}{12}s_i < \sum_{i\in L} s_i. \tag{4.1}$$

The first term comes from the fact that in this example every large bin must have a level strictly greater than $(\frac{19}{12}s_i - 1)/s_i$ and the items stored above level 1 cannot be reduced. The second sum is the total size of all items stored below level 1 assuming they were all of size $\frac{12}{7}$ and all could be reduced to size 1. The inequality states that if there is to be a solution then all the items must be in the large bins below level 1. This inequality reduces to $\frac{1}{6}\sum_{i\in L} s_i < |L|$, or $\bar{s} < 6$. □

The next lemma shows that if there is a bin of capacity $s_i \in [2, 6)$ then it must contain a "large" item of size at least $\frac{12}{7}$. Such an item must be packed in a large bin. □

LEMMA 4.5. *If $s_i \in [2, 6)$ then bin i contains an item of size greater than $\frac{12}{7}$.*

Proof. Assume to the contrary that every item is of size at most $\frac{12}{7}$. There are at most $\lfloor s_i - 1\rfloor$ items packed into this bin since otherwise this bin and its contents could be removed creating a smaller counterexample. Adding one item of size 1 should overfill the bin to a level of at least $\frac{19}{12}$. Thus

$$\lfloor s_i - 1\rfloor\left(\frac{12}{7}\right) + 1 \geqq \frac{19}{12}s_i.$$

The reader may check that for $s_i \in [2, 6)$ that the above inequality is not satisfied and thus some item must be larger than $\frac{12}{7}$. In any optimal solution it must be packed into a large bin. □

LEMMA 4.6. *In a smallest counterexample there can be no bins of size $s_i \in [\frac{12}{7}, 2)$.*

Proof. Let $s_i \in [\frac{12}{7}, 2)$. Observe that such a bin is overfull. In any optimal solution such a bin can hold at most one item. There are currently either two items or one item that is too large and in either case the bin and its contents can be removed providing a smaller counterexample. □

LEMMA 4.7. *There are no large bins with $s_i < 6$.*

Proof. If there were then by the last two lemmas $s_i \in [2, 6)$ and each bin must contain an item of size at least $\frac{12}{7}$. This must be packed in a large bin in any optimal solution. A necessary condition for there to be an optimal schedule is

$$\begin{aligned} 1 + \sum_{s_i \geqq 6} \left(\frac{s_i}{\frac{12}{7}} + \frac{7}{12} s_i - 1 \right) + \sum_{s_i \in [2,6)} \left(\frac{12}{7} + \frac{s_i - \frac{12}{7}}{\frac{12}{7}} + \frac{7}{12} s_i - 1 \right) \\ = 1 + \frac{14}{12} \sum_{s_i \geqq 6} s_i - \sum_{s_i \geqq 6} 1 + \sum_{s_i \in [2,6)} \left(\frac{14}{12} s_i - \frac{2}{7} \right) \\ \leqq \sum_{i \in L} s_i. \end{aligned}$$

The terms on the first line are a lower bound on what must be stored in the large bins. The size of item n is 1. The first sum is the total of all items stored in bins with $s_i \geqq 6$, after the items below level 1 have been reduced from $\frac{12}{7}$ to 1 and the items from level 1 to level $(\frac{19}{12} s_i - 1)/s_i$ remain unchanged. The second sum is for the bins of size $s_i \in [2, 6)$. Here we assume there is at least one item of size $\frac{12}{7}$ which cannot be reduced by trading with items in the small bins. The total space available in the large bins is $\sum_{i \in L} s_i$. The above inequality holds if and only if

$$\frac{1}{6} \sum_{i \in L} s_i \leqq \sum_{s_i \geqq 6} 1 + \sum_{s_i \in [2,6)} \frac{2}{7} - 1$$

if and only if

$$\frac{1}{6} \sum_{s_i \geqq 6} s_i + \frac{1}{6} \sum_{s_i \in [2,6)} s_i < \sum_{s_i \geqq 6} 1 + \sum_{s_i \in [2,6)} \frac{2}{7}$$

if and only if

$$\sum_{s_i \geqq 6} 1 + \frac{1}{3} \sum_{s_i \in [2,6)} 1 < \sum_{s_i \geqq 6} 1 + \frac{2}{7} \sum_{s_i \in [2,6)} 1.$$

Assuming the sum over $s_i \in [2, 6)$ is not empty one concludes

$$\frac{1}{3} < \frac{2}{7},$$

a contradiction. □

From the previous lemmas we see that for $\{s_i\}_{i \in L}$, $\bar{s} < 6$ but $s_i \geqq 6$ for all large bins in any counterexample. Thus no large bins exist and the theorem follows. □

The following example shows that it is possible for the heuristic to do as poorly as $\bar{z}/z^* = 1.512$. The items are as follows

28	of size $2 - \delta$,
28	of size $2 - \delta - \varepsilon$,
28	of size $2 - \delta - 2\varepsilon$,
⋮	⋮
28	of size $1 + \gamma + \varepsilon$,
14+28	of size $1 + \gamma$,
8	of size 1.

The bins are

7	of size 8,
28	of size $2-\delta$,
28	of size $2-\delta-\varepsilon$,
⋮	⋮ ⋮
28	of size $1+\gamma+\varepsilon$,

where ε is picked as small as you like but divides $(2-\delta)-(1+\gamma)$ evenly. The heuristic places the items in the first 7 large bins as in Fig. 4.

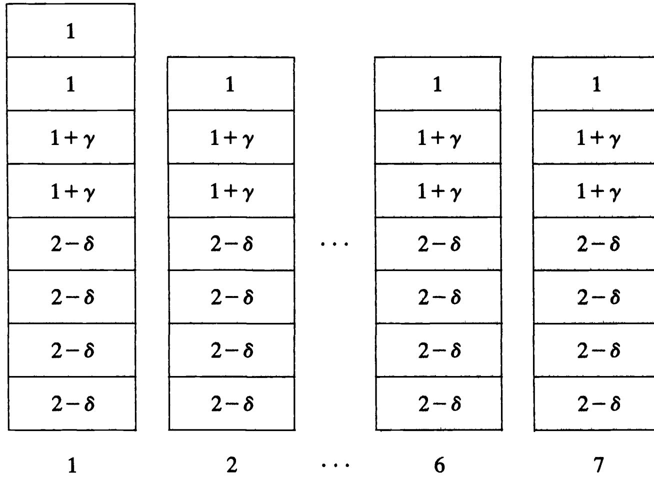


FIG. 4.

As for the remaining bins, the 28 items of size $2-\delta-k\varepsilon$ are placed in bins of size $2-\delta-(k-1)\varepsilon$. The optimal solution packs the largest 7 bins as in Fig. 5.

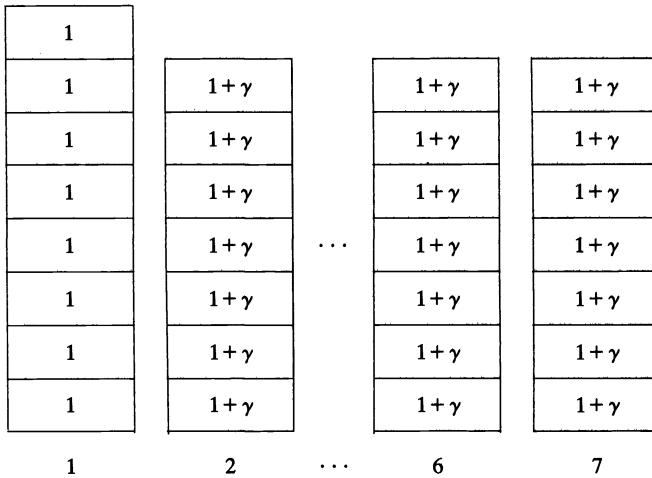


FIG. 5.

The remaining items are placed in bins of the same size. Setting $\gamma=\frac{1}{7}$, $\delta=\frac{4}{85}$ the example has an error of $\frac{251}{166}$.

REFERENCES

E. G. COFFMAN JR., ed. (1976), *Computer and Job-Shop Scheduling Theory*, John Wiley, New York.

E. G. COFFMAN JR., J. Y. T. LEUNG AND D. SLUTZ (1977), *On the optimality of first-fit and level algorithms for parallel machine assignment and sequencing*, Proc. 1977 International Conference on Parallel Processing, Jean-Loup Baer, ed.

E. G. COFFMAN JR., M. GAREY AND D. JOHNSON (1978), *An application of bin-packing to multiprocessor scheduling*, this Journal, 7, pp. 1–17.

S. A. COOK (1971), *The complexity of theorem-proving procedures*, Proc. Third Annual ACM Symposium on Theory of Computing, pp. 151–158.

G. DOBSON (1981), *Some exact and approximation algorithms for packing and covering problems*, Ph.D. thesis, Operations Research Dept., Stanford University, Stanford, CA.

D. K. FRIESEN (1984), *Tighter bounds for the* MULTIFIT *processor scheduling algorithm*, this Journal, 13, pp. 170–181.

D. K. FRIESEN AND M. A. LANGSTON (1983), *Bounds for multifit scheduling on uniform processors*, this Journal, 12, pp. 60–70.

M. GAREY, R. GRAHAM AND D. JOHNSON (1978), *Performance guarantees for scheduling algorithms*, Oper. Res., 26, pp. 3–21.

M. R. GAREY, R. L. GRAHAM, D. S. JOHNSON AND A. C. YAO (1976), *Resource constraint scheduling as generalized bin packing*, J. Combin. Theory A., 21, pp. 257–298.

M. R. GAREY AND D. S. JOHNSON (1979), *Computers and Intractability: A Guide to the Theory of* NP-*Completeness*, W. H. Freeman, San Francisco.

T. GONZALEZ, O. IBARRA AND S. SAHNI (1977), *Bounds for* LPT *schedules on uniform processors*, this Journal, 6, pp. 155–166.

T. GONZALEZ AND S. SAHNI (1978), *Flowshop and jobshop schedules: complexity and approximation*, Oper. Res., 26, pp. 36–52.

R. L. GRAHAM (1969), *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17, pp. 416–425.

——— (1966), *Bounds on certain multiprocessing timing anomalies*, SIAM J. Appl. Math., 45, pp. 1563–1581.

E. HOROWITZ AND S. SAHNI (1976), *Exact and approximate algorithms for scheduling nonidentical processors*, J. Assoc. Comput. Mach., 23, pp. 317–327.

O. IBARRA AND C. KIM (1977), *Heuristic algorithms for scheduling independent tasks on nonidentical processors*, J. Assoc. Comput. Mach., 24, pp. 280–289.

D. JOHNSON, A. DEMERS, J. ULLMAN, M. GAREY AND R. GRAHAM (1974), *Performance bounds for simple one-dimensional bin packing algorithms*, this Journal, 3, pp. 299–325.

R. KARP (1972), *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, pp. 85–104.

——— (1975), *On the complexity of combinatorial problems*, Networks, 5, pp. 45–68.

B. KORTE (1979), *Approximative algorithms for discrete optimization problems*, Ann. Discr. Math., 4, pp. 85–120.

S. SAHNI (1976), *Algorithms for scheduling independent tasks*, J. Assoc. Comput. Mach., 23, pp. 116–127.

S. SAHNI AND E. HOROWITZ (1978), *Combinatorial problems: reducibility and approximation*, Oper. Res., 26, pp. 718–759.

SIAM J. COMPUT.
Vol. 13, No. 4, November 1984

© 1984 Society for Industrial and Applied Mathematics
005

ON RELATIVIZED POLYNOMIAL AND EXPONENTIAL COMPUTATIONS*

HANS HELLER†

Abstract. The relationship between polynomial and exponential time bounded relativized computations is investigated. It is shown that there exists an oracle X such that relative to X nondeterministic exponential computations yield only the languages in $\Sigma_2^{P,X}$ from the polynomial hierarchy. More informally, relative to that X two polynomial quantifiers are as powerful as one exponential quantifier.

Key words. **P**, **NP**, relativization, polynomial hierarchy, exponential-time bounded Turing machines

Introduction. The difficulty of several problems in complexity theory such as the $\mathbf{P}=\mathbf{NP}$ question led to the investigation of relativized computations. An increasing number of publications about this kind of computations shows the interest in this topic. Nevertheless—as far as I know—relativized results did not lead to the solution of any nonrelativized problem. R. Book and C. Wrathall showed in [5] and [6] the equivalence of statements about relativized complexity classes with statements about nonrelativized complexity classes. Therefore there is some hope that relativized results may help solve nonrelativized problems.

The insight we gain from relativized results is mostly of the kind that (1) shows us the difficulty of a problem and (2) exhibits possible inclusion relations between certain complexity classes.

1. The existence of oracles X and Y such that $\mathbf{P}(X)=\mathbf{NP}(X)$ and $\mathbf{P}(Y)\neq \mathbf{NP}(Y)$ (see [1]) tells us that a solution of the $\mathbf{P}=\mathbf{NP}$ question must be done by techniques which do not uniformly apply to all relativized $\mathbf{P}(Z)=\mathbf{NP}(Z)$ questions and therefore may be intricate.

2. The existence of an oracle X such that $\mathbf{P}(X)=\mathbf{PSpace}(X)$ (see [1]) shows that **P** may not only equal **NP** but also **PSpace** in contrast to the more commonly accepted assumption that $\mathbf{P}\subset\mathbf{PSpace}$.

The results of this paper show inclusion relations between polynomial and exponential complexity classes relativized to the same X.

1. Basic concepts. The notation used in the following is similar to that of [1]. The reader is assumed to be familiar with the following concepts not explained in detail: (non-)deterministic Turing machines, oracle machines, time bounded computations, reducibility ($\leqq_m^P$ denotes polynomial many-one reducibility, $\leqq_T^P$ denotes polynomial Turing reducibility), and completeness with respect to a given reducibility.

The underlying alphabet is $\Sigma=\{0,1\}$. A Turing machine M can be converted to a polynomial-time bounded machine by attaching a clock to M which stops the computation of M on an input x after $p(|x|)$ steps for a polynomial p. A recursive enumeration of all polynomial-time bounded machines, deterministic or nondeterministic, is generated by attaching a clock for all polynomials to all oracle machines.

Let $p_i(n)=i+n^i$ and P_i (NP_i) be an enumeration of the polynomial-time bounded deterministic (nondeterministic) oracle machines. Without loss of generality it is assumed that P_i (NP_i) runs in time p_i, $P_i^X(NP_i^X)$ denotes the language accepted by P_i

* Received by the editors January 5, 1983.

† Institut für Informatik, Technische Universität München, Arcisstr. 21, 8 München 2, West Germany. Present address, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

(NP_i) with oracle X. P_i^X (NP_i^X) also denotes machine P_i (NP_i) with oracle X. This should not cause any confusion.

There is a coding $\langle \cdot, \cdot \rangle$ of finite sequences of words over Σ into Σ^* for which encoding and decoding can be done in polynomial time. The coding of oracle machines is such that the set

$$K(X) = \{\langle i, x, 0^l \rangle : NP_i^X \text{ accepts } x \text{ in fewer than } l \text{ steps}\}$$

can be computed by a polynomial-time bounded nondeterministic oracle machine with oracle X. Each of the usual codings satisfies this condition.

For $\mathscr{C}$ a class of languages over Σ define the operators **P** and **NP** by

$$\mathbf{P}(\mathscr{C}) = \{P_i^X : i \in N, X \in \mathscr{C}\}, \qquad \mathbf{NP}(\mathscr{C}) = \{NP_i^X : i \in N, X \in \mathscr{C}\}.$$

We write $\mathbf{P}(X)$ for $\mathbf{P}(\{X\})$ and accordingly for the operator **NP**. **P** (**NP**) also denotes the class of languages accepted deterministically (nondeterministically) in polynomial time without oracle X. co $\mathscr{C}$ is the class of complements of sets in $\mathscr{C}$:

$$\text{co } \mathscr{C} = \{\bar{X} : C \in \mathscr{C}\}.$$

Define the polynomial hierarchy relative to X by

$$\Sigma_0^{P,X} = \Pi_0^{P,X} = \Delta_0^{P,X} = \mathbf{P}(X);$$

for $i \geqq 0$,

$$\Sigma_{i+1}^{P,X} = \mathbf{NP}(\Sigma_i^{PX}),$$

$$\Pi_{i+1}^{P,X} = \text{co } \Sigma_{i+1}^{P,X},$$

$$\Delta_{i+1}^{P,X} = \mathbf{P}(\Sigma_i^{P,X}).$$

Obviously the following inclusions hold:

$$\Sigma_i^{P,X} \cup \Pi_i^{P,X} \subseteq \Delta_{i+1}^{P,X} \subseteq \Sigma_{i+1}^{P,X} \cap \Pi_{i+1}^{P,X}.$$

$\mathbf{PH}(X) = \{(\Sigma_i^{P,X}, \Pi_i^{P,X}, \Delta_i^{P,X}) : i \in N\}$ is the polynomial hierarchy relativized to X. The hierarchy $\mathbf{PH}(X)$ can also be defined by polynomial bounded quantification:

$A \in \Sigma_{i+1}^{P,X}$ iff there are a $B \in \mathbf{P}(X)$ and a polynomial p such that

$$x \in A \leftrightarrow \exists y_1 \forall y_2 \cdots Q y_{i+1} [|y_1|, \cdots, |y_{i+1}| \leqq p(|x|) \wedge \langle x, y_1, \cdots, y_{i+1} \rangle \in B]$$

(where Q is $\exists$ if i is even otherwise $\forall$).

Accordingly, sets in $\Pi_{i+1}^{P,X}$ can be defined by $i+1$ alternating quantifiers starting with $\forall$.

Let us consider the above mentioned set $K(X)$ as image of X under the function K (see also the analogous jump operation in the theory of recursive functions [11]). It therefore makes sense to write $K^2(X)$ for $K(K(X))$. Since $K(X) \in \mathbf{NP}(X)$, we have $K^i(X) \in \Sigma_i^{P,X}$. Obviously $K^i(X)$ is $\leqq_m^P$-complete in $\Sigma_i^{P,X}$ and $\overline{K^i(X)}$ is $\leqq_m^P$-complete in $\Pi_i^{P,X}$. Thereby we achieve one more representation for $\Sigma_{i+1}^{P,X}$, $\Pi_{i+1}^{P,X}$, and $\Delta_{i+1}^{P,X}$:

$$A \in \Sigma_{i+1}^{P,X} \text{ iff } A \in \mathbf{NP}(K^i(X)),$$

$$A \in \Pi_{i+1}^{P,X} \text{ iff } A \in \text{co } \mathbf{NP}(K^i(X)),$$

$$A \in \Delta_{i+1}^{P,X} \text{ iff } A \in \mathbf{P}(K^i(X)).$$

Relativization of a lemma of Meyer and Stockmeyer (see [10, Lemma 3.1]) yields the following result:

LEMMA 1. *$\Sigma_i^{P,X} \cup \Pi_i^{P,X} = \Delta_i^{P,X}$ implies $\Sigma_k^{P,X}$ for all $k > i$.*

We say hierarchy **PH** (X) extends 0 levels if $\Sigma_0^{P,X}=\Sigma_1^{P,X}$. Hierarchy **PH** (X) extends $i+1$ levels, for $i\geqq 0$, if

$$\Sigma_i^{P,X}\subset\Sigma_{i+1}^{P,X} \text{ and } \Sigma_{i+1}^{P,X}=\Pi_{i+1}^{P,X}.$$

2. The exponential hierarchy. Let P be the set of polynomials over N and lin be the set of linear functions from N to N. 2^P (2^{lin}) is the set of functions from N to N where the exponent is a polynomial (linear function). 2^{2^P} and $2^{2^{\text{lin}}}$ are explained accordingly. For notational reasons 2^P (2^{lin}) is also denoted by EP (EL).

If F is a set of functions (time bounds), then DTime (F, X) (NTime (F, X)) is the set of languages accepted within a time bound from F by deterministic (nondeterministic) oracle machines with oracle X. DTime (EP, X) and DTime (EL, X) are denoted by **EP** (X) and **EL** (X), respectively. **NEP** (X) and **NEL** (X) are used for the nondeterministic classes.

NEP (X) is the set of languages definable by exponential bounded (bounds from EP) existential quantification over sets in **P** (X). EP_i (EL_i) is the ith deterministic oracle machine running in time 2^{p_i} (2^{in}). NEP_i (NEL_i) are the nondeterministic analogues.

Time separation results showing that computation within a greater time bound is more powerful than computation within a smaller time bound are well known from the literature (see also [12] and for the relativized case [14]). Here we are only interested in the kind of proper inclusions which are listed in the next lemma.

LEMMA 2 (time hierarchy).

1. **P** $(X)\subset$ **EL** $(X)\subset$ **EP** $(X)\subset$ DTime $(2^{2^{\text{lin}}}, X)\cdots$.
2. **NP** $(X)\subset$ **NEL** $(X)\subset$ **NEP** $(X)\subset$ NTime $(2^{2^{\text{lin}}}, X)\cdots$.

We will now describe two methods (called substitution and translation) which have been applied many times. Each of the two methods is introduced by a single application. We are more interested in the methods than in the special results exhibited in Lemmas 3 and 4. Both methods depend on a padding technique (see also [12, Lemma 4]).

LEMMA 3 (substitution). **P** (**EL** (X)) $=$ **EP** (X).

Proof. We must show that **EP** $(X)\subseteq$ **P** (**EL** (X)); the other direction is obvious. Let $A=EP_i^X$. Set $A^*=\{x0^{p_i(|x|)}: x\in A\}$. A^* is the padded version of A. A^* is in **EL** (X). Consider the oracle machine P_j which on input x generates $y=x0^{p_i(|x|)}$ and accepts iff y is in the oracle. P_j with oracle A^* accepts A. Since P_j runs in polynomial time and the oracle A^* is in **EL** (X), we yield $A\in$ **P** (**EL** (X)). QED

By the method of substitution we can also prove the following two assertions:

1. **EP** (**EP** (X)) $=$ **EP** (**EL** (X)) $=$ DTime $(2^{2^P}, X)$.
2. **EL** (**EL** (X)) $=$ **EL** (**EP** (X)) $=$ DTime $(2^{2^{\text{lin}}}, X)$.

From these assertions and Lemma 2 (time hierarchy) it follows that Turing reducibility in exponential time ($\leqq_T^{EP}$ and $\leqq_T^{EL}$) is not transitive. The reason for this is that EP and EL are not closed under composition.

Lemma 4 shows how certain equalities achieved for small time bounds can be transferred to greater time bounds. Translational methods are also used in [4] and [5].

LEMMA 4 (translation). **NP** $(X)=$ co **NP** (X) *implies* **NEL** $(X)=$ co **NEL** (X).

Proof. Let $A=NEL_i^X$. Set $A^*=\{x0^{2^{i|x|}}: x\in A\}$. A^* is in co **NP** (X). Since co **NP** $(X)=$ **NP** (X) there is a j such that $A^*=NP_j^X$. The nondeterministic machine NEL_k which on input x generates $y=x0^{2^{i|x|}}$ and simulates NP_j^X on y accepts A with oracle X in exponential time. The time bound is in EL since $p_j(2^{in})$ is bounded above by a function in EL. QED

How can an exponential analogue of the polynomial hierarchy be defined? We must be careful because of the difference between exponential and polynomial computation. To maintain the analogy the exponential hierarchy should not exceed DTime $(2^{2^P}, X)$. Since DTime $(2^{2^P}, X) \subset \mathbf{NEP}(\mathbf{NEP}(\mathbf{NEP}(X)))$ by substitution and time hierarchy, we cannot define the exponential hierarchy by repeated application of the operator **NEP**. Nevertheless, the exponential hierarchy can be represented by exponential bounded quantification. Let

$$\Sigma_0^{EP,X} = \Pi_0^{EP,X} = \Delta_0^{EP,X} = \Delta_1^{EP,X} = \mathbf{EP}(X).$$

$\Sigma_i^{EP,X}$, for $i \geqq 1$, is given by

$A \in \Sigma_i^{EP,X}$ iff there is a set $B \in \mathbf{P}(X)$ and a j such that

$$x \in A \leftrightarrow \exists y_1 \forall y_2 \cdots Qy_i[|y_1|, \cdots, |y_i| \langle 2^{p_j(|x|)} \wedge \langle x, y_1, \cdots, y_i\rangle \in B]$$

(where Q is $\exists$ if i is odd otherwise $\forall$).

$$\Pi_i^{EP,X} = \mathrm{co}\,\Sigma_i^{EP,X}.$$

It is easy to see that

$$\Sigma_{i+1}^{EP,X} = \mathbf{NEP}(K^i(X)).$$

Therefore $\Delta_i^{EP,X}$ can be defined by

$$\Delta_{i+1}^{EP,X} = \mathbf{EP}(K^i(X)).$$

As usual we have

$$\Sigma_i^{EP,X} \cap \Pi_i^{EP,X} \subseteq \Delta_{i+1}^{EP,X} \subseteq \Sigma_{i+1}^{EP,X} \cap \Pi_{i+1}^{EP,X}.$$

Warning. The exponential hierarchy defined in this way seems to be very similar to the polynomial hierarchy. We must, however, state a remarkable difference: If $\Sigma_i^{EP,X}$ is closed under complements, then $\Sigma_i^{EP,X}$ is not necessarily equal to all $\Sigma_k^{EP,X}$ for $k > i$, in contrast to the polynomial case (see Lemma 1). The reason for this is that 2^P is not closed under composition whereas the polynomials are. An example will be given below. Some simple facts about exponential-time bounded computations are established next.

(F1) *$K^i(X)$ is $\leqq_m^{EP}$-complete in $\Sigma_i^{EP,X}$*,

(F2) $\mathbf{NEP}(\mathbf{EP}(X)) = \mathbf{EP}(\mathbf{EP}(X)) = \mathrm{DTime}(2^{2^P}, X)$.

Since $\mathbf{EP}(X)$ is closed under polynomial quantification, we have $\mathbf{NP}(\mathbf{EP}(X)) = \mathbf{P}(\mathbf{EP}(X))$ from which we get the first equation of (F2) by translation. The second equation follows by substitution.

Define $K_{\log}(X)$ by

$$K_{\log}(X) = \{\langle i, x, l\rangle : \langle i, x, 0^l\rangle \in K(X)\}.$$

(F3) *$K_{\log}(X)$ is $\leqq_m^P$-complete in* $\mathbf{NEP}(X)$.

Obviously every set in $\mathbf{NEP}(X)$ is $\leqq_m^P$-reducible to $K_{\log}(X)$. Since $K_{\log}(X)$ is many-one-reducible to $K(X)$ in time 2^n and $K(X) \in \mathbf{NP}(X)$, we yield $K_{\log}(X) \in \mathbf{NEP}(X)$.

LEMMA 5. $\mathbf{P}(K_{\log}(X)) = \mathbf{NEP}(X)$ iff $\mathbf{NEP}(X) = \mathrm{co}\,\mathbf{NEP}(X)$.

Proof. $\mathbf{P}(K_{\log}(X)) = \mathbf{NEP}(X)$ implies obviously $\mathbf{NEP}(X) = \mathrm{co}\,\mathbf{NEP}(X)$.

Suppose $\mathbf{NEP}(X)$ is closed under complements. Let $K_{\log}(X) = NEP_k$ and $\overline{K_{\log}(X)} = NEP_l$. By (F3) it is sufficient to show that $P_i^{K_{\log}(X)} \in \mathbf{NEP}(X)$ for arbitrary i. An oracle machine M_i is described which with oracle X accepts the set $P_i^{K_{\log}(X)}$.

M_i^X: On input x simulate P_i. For each string y queried select nondeterministically

a computation of NEP_k^X and a computation of NEP_l^X on y. If in both computations y is rejected, then reject x. Else, if NEP_k^X accepts y, then continue the simulation of P_i in the yes state. If NEP_l^X accepts y, then continue in the no state. (Only one of NEP_k^X and NEP_l^X can accept y.)

M_i^X runs obviously nondeterministically in exponential time. QED

In the proof of Lemma 5 a well-known method is applied: a set and its complement are simultaneously computed. This method is also useful in other cases. For instance, we can prove the following:

1. $\mathbf{P}(\Sigma_i^{P,X} \cap \Pi_i^{P,X}) = \Sigma_i^{P,X} \cap \Pi_i^{P,X}$.
2. $\mathbf{NP}(\mathbf{NEP}(X) \cap \mathrm{co}\,\mathbf{NEP}(X)) = \mathbf{NEP}(X) \cap \mathrm{co}\,\mathbf{NEP}(X)$.

3. Main result. The investigation presented here was initiated by the following result of M. I. Dekhtyar [7]:

THEOREM 6. *There exists a recursive oracle X for which* $\mathbf{NP}(X) = \mathbf{EP}(X)$.

Since no proof appears in [7] the proof is presented here for the ease of the reader. The construction is similar to that of [1, Thm. 5] where an oracle X is constructed such that $\mathbf{PH}(X)$ extends 1 level; i.e., $\mathbf{P}(X) \subset \mathbf{NP}(X) = \mathrm{co}\,\mathbf{NP}(X)$.

Proof. Define $D_1(X)$ by

$$D_1(X) = \{\langle i, x, l\rangle : EP_i^X \text{ accepts in } < l \text{ steps}\}.$$

$D_1(X)$ *is* $\leqq_m^P$-complete in $\mathbf{EP}(X) = \Delta_1^{EP,X}$ since

$$x \in EP_i^X \leftrightarrow \langle i, x, 2^{P_i(|x|)}\rangle$$

and the string $\langle i, x, 2^{P_i(|x|)}\rangle$ can be computed in polynomial time for fixed i. We shall construct an X such that

$$x \in D_1(X) \leftrightarrow \exists y[|y| = 3|x| \wedge xy \in X]$$

for all x and therefore $D_1(X) \in \mathbf{NP}(X)$. The $\leqq_m^P$-completeness of $D_1(X)$ in $\mathbf{EP}(X)$ implies then $\mathbf{NP}(X) = \mathbf{EP}(X)$.

For the beginning set $X = \varnothing$.

Stage m. For each x of length m such that x is the encoding of a triple—i.e. $x = \langle i, a, l\rangle$—consider the computation of EP_i^X on a for l steps. Reserve for $\bar{X}$ all strings asked during this computation which are not in X. If EP_i^X accepts a in fewer than l steps, then select a y of length $3m$ such that xy is not reserved for $\bar{X}$. Add xy to X.

We must verify that we can always choose an appropriate y. If $x = \langle i, a, l\rangle$ has length m then at most l strings are reserved for $\bar{X}$ with respect to the computation of EP_i^X on a for l steps. Since $l < 2^m$ and at most 2^m different strings are considered at stage m, it follows that fewer than 2^{2m} strings are reserved for $\bar{X}$ at stage m. At stages before m including m fewer than $2^2 + \cdots + 2^{2m} < 2^{3m}$ are reserved. There are 2^{3m} different y of length $3m$ and therefore at least one string xy, where $|y| = 3m$, is not reserved for $\bar{X}$. QED

Define inductively the following classes of functions $e(0, P) = P$, $e(n+1, P) = 2^{e(n,P)}$. $e(n, \mathrm{lin})$ is defined accordingly. Let X be as in Theorem 6. For this X we can state the following simple propositions:

(P1) $\mathbf{P}(X) \subset \mathbf{NP}(X)$ *because* $\mathbf{P}(X) \subset \mathbf{EP}(X)$ *by time hierarchy.*

(P2) $\mathbf{NP}(X) = \mathrm{co}\,\mathbf{NP}(X)$.

Note that (P1) and (P2) imply that the polynomial hierarchy relative to this X extends exactly one level.

(P3) **NEL** $(X) = \text{co}\,$**NEL** (X) *and likewise for greater time bounds by translation of* (P2)

(P4) NTime $(e(i, P), X) = \text{DTime}\,(e(i+1, P), X)$ *for* $i \geqq 0$ *by translation.*

(P5) DTime $(e(i, P), X) \subset \text{NTime}\,(e(i, P), X)$ *by* (P4) *and time hierarchy.*

(P6) DTime $(e(i, \text{lin}), X) \subset \text{NTime}\,(e(i, \text{lin}), X)$ *by translation and* (P5).

Let us express these facts more informally: polynomial bounded quantification (one existential or universal quantifier) over sets in $\mathbf{P}(X)$ is as powerful as deterministic exponential-time bounded computation relative to oracle X. Exponential bounded quantification over $\mathbf{P}(X)$ (one exponential quantifier) yields more than the polynomial hierarchy. The same holds for greater time bounds: for example, exponential quantification over $\mathbf{P}(X)$ is as powerful as deterministic computation relative to X within time bounds from $e(2, P) = 2^{2^P}$. In the following theorem one more possibility for the relationship of polynomial and exponential computation is established.

THEOREM 7. *There is a recursive oracle X for which* $\Sigma_2^{P,X} = \mathbf{EP}\,(K(X)) = \Delta_2^{EP,X}$.

Proof. Define $D_2(X)$ by

$$D_2(X) = \{\langle i, x, l\rangle : EP_i^{K(X)} \text{ accepts } x \text{ in} < l \text{ steps}\}$$

$D_2(X) \leqq_m^P$-complete in $\Delta_2^{EP,X}$ because

$$x \in EP_i^{K(X)} \leftrightarrow \langle i, x, 2^{p_i(|x|)}\rangle \in D_2(X)$$

and for fixed i, $2^{p_i(|x|)}$ can be written down in time $p_i(|x|)$. The construction of the oracle will guarantee that

$$x \in D_2(X) \leftrightarrow \exists y \forall z[|y| = |z| = 4|x| \rightarrow xyz \in X].$$

Thereby $D_2(X) \in \Sigma_2^{P,X}$ and $\mathbf{EP}\,(K(X)) \subseteq \Sigma_2^{P,X}$. Since $\Sigma_2^{P,X} \subseteq \mathbf{EP}\,(X) \subseteq \mathbf{NEP}\,(X) \subseteq \mathbf{EP}\,(K(X))$ we have

$$\Sigma_2^{P,X} = \mathbf{EP}\,(X) = \mathbf{NEP}\,(X) = \mathbf{EP}\,(K(X)) = \Delta_2^{EP,X}.$$

For the encoding we must determine what $EP_i^{K(X)}$ does and therefore we must determine $K(X)$, not only X. For that reason we will in the sequel apply a method which we call settling. Consider a string s, where s represents the acceptance of NP_i^X on x for l steps relative to an oracle X. To settle string s we try to bring s into $K(X)$ by adding strings to X and thereby generating an accepting computation of NP_i^X on x within l steps. Distinguish two cases:

1. It is possible to yield $s \in K(X)$—i.e., to make NP_i^X accept x in fewer than l steps—by adding strings to X. Add these strings to X and select an accepting computation of NP_i^X on x. We can manipulate X, that means delete or add strings, without yielding $s \notin K(X)$ if we do not change those fewer than l many strings asked during the chosen accepting computation.

2. It is not possible to yield $s \in K(X)$. NP_i^X does not accept x in few than l steps, for any extension of X which does not contain strings reserved for $\bar{X}$. In this case we can add to X arbitrary strings, not reserved for $\bar{X}$, without changing $s \notin K(X)$ to $s \in K(X)$.

In both cases X can be manipulated with certain restrictions without yielding undesired effects.

For the beginning set $X = \varnothing$.

Stage m. For each string x of length m do the following: If x is of the form $\langle i, a, l\rangle$—i.e. x is the encoding of a triple—then consider the computation of $EP_i^{K(X)}$ on a for l steps. Settle one after the other the strings asked by $EP_i^{K(X)}$ on a by adding to X strings of length $\geqq 4m$. To be more precise, for the first string s (first with respect

to the order in which the strings are asked by $EP_i^{K(X)}$ on a) such that s is not yet settled and s is asked by $EP_i^{K(X)}$ on a, try to bring s into $K(X)$ by adding strings of length $\geqq 4m$ to X which are not reserved for $\bar{X}$. If this is possible then consider an accepting computation represented by s: since $s \in K(X)$, s is of the form $\langle j, b, 0^k \rangle$ and the computation under consideration is an accepting computation of NP_j^X on b for k steps. Reserve for $\bar{X}$ the strings asked in this computation which are not in X and are of length $\geqq 4m$. After that we say s is settled irrespective if $s \in K(X)$ or $s \notin K(X)$. Continue in the same way with the next such s asked by $EP_i^{K(X)}$ on a. After all strings asked in the computation of $EP_i^{K(X)}$ on a for l steps are settled, test if $x = \langle i, a, l \rangle \in D_2(X)$. If x is in $D_2(X)$, then select a y_0 of length $4m$ such that no string xy_0z where $|z| = 4m$ is reserved for $\bar{X}$ and add all strings xy_0z to X, where z varies over all strings of length $4m$. Otherwise, if $x \notin D_2(X)$, then do nothing.

Let us verify the construction:

1. We must show that there are enough strings left—not reserved for $\bar{X}$—such that an appropriate y_0 can be selected. First we estimate the number of strings which are reserved for $\bar{X}$ at stage m and at stages before m. There are 2^m strings of length m. For those strings of length m which are of the form $\langle i, a, l \rangle$ the computation of $EP_i^{K(X)}$ on a is considered. This computation is bounded by $l < 2^m$. That means fewer than 2^m different strings are asked and the length of these strings is less than 2^m. During the settling of one of these strings fewer than 2^m strings are reserved for $\bar{X}$. It turns out that fewer than 2^{3m} are reserved at stage m. Therefore at stages before m including stage m fewer than $2^3 + (2^3)^2 + \cdots + (2^3)^m < 2^{4m}$ strings are reserved. Since there are 2^{4m} different y_0 of length $4m$, we can always select an appropriate y_0.

2. We must ensure that

$$x \in D_2(X) \leftrightarrow \exists y \forall z[|y| = |z| = 4|x| \rightarrow xyz \in X]$$

holds. A similar estimation to the above shows that settling of strings at stages up to and including m adds fewer than 2^{4m} strings to X. Therefore for a fixed x and exactly one y_0 the 2^{4m} different strings xy_0z where z varies over strings of length $4m$ are added to X if and only if this is done at stage m because $x \in D_2(X)$. QED

Consequences of Theorem 7 are listed in the following (X as in Theorem 7).

(C1) $\Delta_2^{P,X} \subset \Sigma_2^{P,X} = \Pi_2^{P,X} = \mathbf{EP}(X)$

because $\Delta_2^{P,X} = \mathbf{P}(K(X)) \subset \mathbf{EP}(K(X)) = \Delta_2^{EP,X} = \Sigma_2^{P,X}$ by time hierarchy.

Therefore the polynomial hierarchy relativized to this X extends exactly two levels. Further polynomial hierarchies extending two levels have been independently constructed in [8] and [9]. Contrast the hierarchy of Theorem 7 also with the result of T. Baker and A. Selman in [2], where they have shown the existence of an oracle Y such that $\Sigma_2^{P,Y} \neq \Pi_2^{P,Y}$. This hierarchy extends more than two levels but it is not known exactly how many levels it extends.

(C2) $\mathbf{EP}(X) = \Delta_2^{EP,X} \subset \Sigma_2^{EP,X} = \mathrm{DTime}(2^{2^P}, X)$.

By translation of $\Sigma_2^{P,X} = \mathbf{EP}(X)$ we have $\Sigma_2^{EP,X} = \mathrm{DTime}(2^{2^P}, X)$.

(C3) The proposition of (C2) can be translated to greater time bounds. (See also the propositions following Theorem 6.)

(C4) $\mathbf{NEP}(X) = \mathbf{EP}(X)$.

That means, $\Sigma_1^{EP,X}$ is closed under complements but $\Sigma_1^{EP,X} = \Delta_2^{EP,X} \subset \Sigma_2^{EP,X}$ by (C2). Therefore the exponential hierarchy does not collapse into $\Sigma_1^{EP,X}$ (remember the warning).

Let us summarize the result in an informal way:

Theorem 7 is optimal in the sense that $\Delta_2^{EP,X}$ cannot be encoded into a class below $\Sigma_2^{P,X}$ (such as $\Delta_2^{P,X}$) and no class above $\Delta_2^{EP,X}$ (such as $\Sigma_2^{EP,X}$) can be encoded into

$\Sigma_2^{P,X}$. We yield for the X of Theorem 7: Relative to X two polynomial quantifiers are as powerful as one exponential quantifier. Two exponential quantifiers are more powerful. The same can be stated for greater time bounds.

4. Open problems. According to Theorems 6 and 7 we would like to achieve a result which seems to lie between these two results: we would like to encode $\mathbf{NEP}(X) = \Sigma_1^{EP,X}$ into $\mathbf{P}(K(X)) = \Delta_2^{P,X}$. Unfortunately the known methods seem not to work for this case and the problem seems to be difficult. Encoding $\mathbf{NEP}(X)$ into $\mathbf{P}(K(X))$ would yield an X for which $\mathbf{EP}(X) = \mathbf{NEP}(X) \subset \mathbf{EP}(K(X)) = \text{DTime}(2^{2^P}, X)$ (similar to the consequences of Theorems 6 and 7). That means $\mathbf{NEP}(X)$ is closed under complements and properly contained in $\mathbf{EP}(K(X))$. We can ask for an X such that $\mathbf{EP}(X) = \mathbf{NEP}(X) \subset \mathbf{EP}(K(X))$ independent of the polynomial hierarchy. The problem is unsolved. Even the following less hard problems are not settled.

1. Does there exist an X such that

   $\mathbf{NEP}(X) = \text{co}\,\mathbf{NEP}(X) \subset \mathbf{EP}(K(X))$

   or equivalently (see Lemma 5)

   $\mathbf{P}(K_{\log}(X)) = \mathbf{NEP}(X) \subset \mathbf{EP}(K(X))$?

2. Does there exist an X such that

   $\mathbf{P}(K_{\log}(X)) \subset \mathbf{EP}(K(X))$

   or equivalently $\mathbf{P}(\mathbf{NEP}(X)) \subset \mathbf{EP}(\mathbf{NP}(X))$?

The inclusion $\mathbf{P}(\mathbf{NEP}(X)) \subseteq \mathbf{EP}(\mathbf{NP}(X))$ holds for all X, since $\mathbf{NEP}(X) \subseteq \mathbf{EP}(\mathbf{NP}(X))$. The positive solution of problem 1 solves also problem 2. Additionally some new problems arise. If question 2 has an affirmative answer, then we can try to find an X such that

$$\mathbf{P}(K_{\log}(X)) \subset \mathbf{NP}(K_{\log}(X)).$$

More generally, how many levels can the polynomial hierarchy relativized to $K_{\log}(X)$ contain? The inclusion $\Sigma_i^{K_{\log}(X)} \subseteq \mathbf{EP}(\mathbf{NP}(X))$ holds for all X. Besides this, it is near at hand to ask for higher levels of the hierarchies. If ever it is possible to construct polynomial hierarchies extending i levels, for $i > 2$, then we can ask the following:

1. Does there exist an oracle X such that

   $\Sigma_i^{P,X} = \Delta_i^{EP,X}$ for $i > 2$?

2. Does there exist an oracle X such that

   $\Delta_{i+1}^{P,X} = \Sigma_i^{EP,X}$ for $i \geqq 1$?

REFERENCES

[1] T. BAKER, J. GILL AND R. SOLOVAY, *Relativizations of the* P = NP? *question*, this Journal, 4 (1975), pp. 431–442.

[2] T. BAKER AND A. SELMAN, *A second step toward the polynomial hierarchy*, 17th Annual IEEE Symposium on the Foundations of Computer Science (1976), pp. 71–75; Theoret. Comput. Sci., 8 (1979), pp. 177–187.

[3] R. V. BOOK, *Comparing complexity classes*, J. Comput. System Sci., 9 (1974), pp. 177–187.

[4] ———, *Translational lemmas, polynomial time, and* $(\log n)^j$*-space*, Theoret. Comput. Science, 1 (1976), pp. 215–226.

[5] ———, *Bounded query machines: on* NP *and* Pspace, Theoret. Comput. Sci., 15 (1981), pp. 27–39.

[6] R. V. BOOK AND C. WRATHALL, *Bounded query machines: on* NP( ) *and* NPquery( ), Theoret. Comput. Sci., 15 (1981), pp. 41–50.

[7] M. I. DEKHTYAR, *On the relation of deterministic and nondeterministic complexity classes*, Mathematical Foundations of Computer Science 1976, Lecture Notes in Computer Science 45, Springer-Verlag, New York, 1977, pp. 282–287.

[8] H. HELLER, *Relativized polynomial hierarchies extending two levels*, Dissertation, Technische Universität München, 1981.

[9] T. LONG, *On some polynomial time reducibilities*, Dissertation, Purdue University, W. Lafayette, IN, 1978.

[10] A. R. MEYER AND L. J. STOCKMEYER, *The equivalence problem for regular expressions with squaring requires exponential space*, Proc. 13th IEEE Symposium on Switching and Automata Theory, 1972, pp. 125–129.

[11] H. ROGERS, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.

[12] J. SEIFERAS, M. FISCHER AND A. R. MEYER, *Separating nondeterministic complexity classes*, J. Assoc. Comput. Mach., 25 (1978), pp. 146–167.

[13] L. J. STOCKMEYER AND A. R. MEYER, *Word problems requiring exponential time*, 5th Annual ACM Symposium on the Theory of Computing, Austin, Texas, 1973, pp. 1–9.

[14] C. WRATHALL, *Complete sets and the polynomial hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 23–33.

SIAM J. COMPUT.
Vol. 13, No. 4, November 1984
© 1984 Society for Industrial and Applied Mathematics
006

THE POWER OF SYNCHRONIZATION MECHANISMS*

H. W. BARZ†

Abstract. A common question in parallel programming is: How to implement a program efficiently for a particular synchronization problem? Here we pose a more general question: Is a certain synchronization mechanism capable of implementing efficiently a particular synchronization problem represented by a program using another synchronization mechanism? We give a definition of "implementing efficiently" which measures efficiency by the amount of parallelism of the parallel programs. If this measure is reduced by the implementation the efficiency of the implementation is inferior. It is shown that some synchronization mechanisms cannot implement programs based on another mechanism without some loss in parallelism, i.e., a reduced measure. These results imply a power hierarchy (or efficiency hierarchy) of synchronization mechanisms. A further outcome of this paper is that simple synchronization mechanisms, with only a few variables, are capable of implementing any synchronization problem implementable by more powerful mechanisms, albeit at some loss in parallelism.

Key words. synchronization problem, synchronization mechanisms, parallelism of programs, power of mechanisms

1. Introduction. There exist a great variety of synchronization mechanisms. The purpose of this paper is not to introduce a new one, but to discuss the differences between those mechanisms. Such a comparison must be based on specified goals. These goals may be modularity [22], ease of verification [2], expressiveness [5], [22], run-time behaviour [12] and the capability to implement problems [17], [18], [25]. Only the last two goals can be specified exactly.

Here we will treat the capability to implement problems. Since we want to do our comparison accurately, the problem has to be specified in an exact manner. Therefore we assume that the problem is already expressed in a parallel program (or a system of processes). Now we want to know whether we can find any other system of processes based on another synchronization mechanism that can implement the same formalized problem. The most serious step in this approach is the definition of "implement". Lipton's [17], [18] "simulate" represents his understanding of "implement". On the basis of his definition he gives a hierarchy of the power of mechanisms—see § 6. He states, for example, that there are problems expressed by PV-multiple [24], a concept which allows the use of several semaphore variables in an operation, which cannot be "simulated" by PV [9], a concept which allows exactly one semaphore variable in an operation.

In our opinion these results need some refinement because intuitively every problem can be implemented by any mechanism [20]—even if this solution is in no way elegant. Our approach accepts this intuitive view but points out in what manner solutions differ. We will show that the solutions differ in the amount of parallelism and the number of variables used. Our definition of a c-weak simulation (§ 3) replaces one feature of Lipton's definition of simulate by a measure for the loss of parallelism. We believe in the intuitive evidence of our results but confess that our proofs are not as elegant as Lipton's.

This paper incorporates only semaphore mechanisms but, using an appropriate model, extensions to other synchronization mechanisms are straightforward.

* Received by the editors September 24, 1982, and in revised form June 30, 1983.

† Institut für Informatik, Abt. 3, Universtät Bonn, Bertha-von-Suttner Platz 6, 5300 Bonn 1, West Germany.

Throughout the paper we use the following notational conventions:

Suppose $\eta = a_1 a_2 \cdots a_k a_{k+1} \cdots a_l \cdots$ is a sequence of elements of a set A. If η is finite, $\lg(\eta)$ denotes the length of η. ε is the empty sequence ($\lg(\varepsilon) = 0$). $\eta[k:l]$ is the substring $a_k a_{k+1} \cdots a_l$ of the sequence η. The set PERMUTATIONS (η) is the set of all permutations of η. The set $M(\eta)$ is the set of all elements in the sequence η. The sequence $a_1 a_2 \cdots a_k$ is a restriction of $\gamma = \gamma_1 a_1 \gamma_2 a_2 \cdots \gamma_k a_k \gamma_{k+1}$ with respect to N, if and only if $\bigcup_{i=1}^{k+1} M(\gamma_i) \cap N = \varnothing$ and $M(a_1 \cdots a_k) \subseteq N$; this will be noted as $\gamma \| N$. The set $\{m, m+1, \cdots, n\} \subset \mathbb{Z}$, $m < n$ is denoted by $[m:n]$. The cardinality of a set N is $\# N$. If $\mathfrak{x}$ and $\mathfrak{y}$ are n- and m-tuples $\mathfrak{x};\mathfrak{y}$ denotes the concatenated $n+m$-tuple. Let $f: K \to N$ be a function: $f(K')$ with $K' \subseteq K$ is the set $\{f(k') | k' \in K'\}$; $f(\eta)$ is the sequence $f(a_1)f(a_2)f(a_3) \cdots$ with $\eta = a_1 a_2 a_3 \cdots$, $M(\eta) \subseteq K$. For the convenience of the reader there is a list of symbols in the Appendix.

2. The model. We have incorporated Lipton's specification of a system of processes [17] in a model based on the parallel program schemata [14], [27].

A state z of an ordered set Var $= (v_1, \cdots, v_i)$ relative to an ordered set Domain $= (D_1, \cdots, D_i)$ is a function from Var to Domain with $z(v_j) \in D_j$.

DEFINITION 2.1. A *system of processes* SP $= (V, W, z_0, A)$ consists of

(1) a nonempty set of variables $V = (\beta^1, \cdots, \beta^k, x_1, \cdots, x_n, y_1, \cdots, y_m)$;

(2) a domain of the variables

$$W = (B^1 \cup \{\omega\}, \cdots, B^k \cup \{\omega\}, X_1 \cdots X_n, Y_1 \cdots Y_m);$$

(3) an initial state z_0 of V relative to W;

(4) a set of actions A, where every $a \in A$ has the form

$$\begin{aligned} a\text{: } \textbf{when } \beta^j = l \wedge e_a(y_1, \cdots, y_m) \textbf{ do } \beta^j &:= f(\beta^j, x_1, \cdots, x_n) \\ x_1 &:= g_1(x_1, \cdots, x_n) \\ &\vdots \\ x_n &:= g_n(x_1, \cdots, x_n) \\ y_1 &:= h_1(y_1, \cdots, y_m) \\ &\vdots \\ y_m &:= h_m(y_1, \cdots, y_m) \end{aligned}$$

and defines a partial function $\bar{a}$ from the set of states to the set of states (Z) with $f, g_1, \cdots, g_n, h_1, \cdots, h_m$ computable, e_a a predicate, $j \in [1:k]$, label $l \in B^j$ and $f(\beta^j, \cdots) \neq l$.

There are functions pr: $A \to [1:k]$, adr: $A \to \bigcup_{i=1}^{k} B^i$ (bijective) so that $\forall a \in A$, $i \in [1:k]$ (adr (a) is label of $a \wedge$ adr $(a) \in B^i \Leftrightarrow$ pr $(a) = i$).

Several remarks may be helpful for the understanding of this definition:

$(\beta^1, \cdots, \beta^k) = \mathfrak{b}$ are the program counters of the k processes.

$(x_1, \cdots, x_n) = \mathfrak{x}$ are the nonsynchronizer variables.

$(y_1, \cdots, y_m) = \mathfrak{y}$ are the synchronizer variables and only these variables are admissible in the predicate e_a by which synchronization is possible.

The function $\bar{a}$ represents the collateral assignment of the values computed by the functions f, g, h. The function pr gives the unique association of actions to processes. The function adr defines the bijective association of actions to their labels. The "stop-symbol" ω is not associated with any action.

The set of actions A^i consists of all actions a with pr $(a) = i$ and this set forms a process. By a^i we denote that pr $(a) = i$.

The function $\bar{a}^i$ of an action a^i is *defined* in a state z iff $z(\beta^i) = \text{adr}(a^i)$ and $e_a(z(\mathfrak{y}))$ is true. The *available-set* $\Gamma(z)$ is the set of actions a^i in the state z with $z(\beta^i) = \text{adr}(a^i)$ and the *ready-set* $\Psi(z)$ is the subset of actions $A' \subseteq \Gamma(z)$, whose corresponding functions are defined in z. An action a is available (ready) in z, iff $a \in \Gamma(z)(\Psi(z))$.

DEFINITION 2.2. Let $\text{SP} = (V, W, z_0, A)$ be a system of processes and $z \in Z$ a state. Then:

(1) A sequence $\eta = a_1 a_2 a_3 \cdots$ of actions of A is called a *transition sequence in z* if and only if $a_1 \in \Psi(z)$, $a_2 \in \Psi(\bar{a}_1(z))$, $a_3 \in \Psi(\bar{a}_2(\bar{a}_1(z)))$, $\cdots$.

(2) A state $z * \eta$ is reached by the transition sequence η in z; i.e. $z * \eta = z$ if $\eta = \varepsilon$, $z * \eta = \bar{a}_l(z * \eta[1:l-1])$ if $\eta \neq \varepsilon$, $\lg(\eta) = l$.

We will call a transition sequence σ in z_0 an *execution* if and only if σ is finite and $\Psi(z_0 * \sigma) = \varnothing$ or σ is an infinite transition sequence. Σ_{SP} is the set of executions in z_0. A state $z = z_0 * \eta$ reached by η in z_0 is *reachable* and Ω_{SP} is the set of all reachable states. Σ_{SP} (resp. Ω_{SP}) will be abbreviated Σ (resp. Ω) when the system of processes is unambiguous.

Example 2.1.

$$\text{SP}_1 = (V_1, W_1, z_{0_1}, A_1),$$
$$V_1 = (\beta^1, \beta^2, y_1, y_2),$$
$$W_1 = (\{l_1, l_2, \omega\}, \{l_3, l_4, l_5, \omega\}, \mathbb{N}_0, \mathbb{N}_0),$$
$$z_{0_1} = (l_1, l_3, 1, 3)$$
$$\text{(a shorthand for } z_{0_1}(\beta^1) = l_1, z_{0_1}(\beta^2) = l_3, \cdots),$$
$$\begin{aligned} A_1 = \{ & a_1\text{: } \textbf{when } \beta^1 = l_1 \wedge y_1 = 1 \textbf{ do } \beta^1 := l_2\ y_1 := 2, \\ & a_2\text{: } \textbf{when } \beta^1 = l_2 \wedge y_2 < 7 \textbf{ do } \beta^1 := \omega, \\ & a_3\text{: } \textbf{when } \beta^2 = l_3 \wedge \textbf{true do } \beta^2 := l_4\ y_1 := y_1 - 1, \\ & a_4\text{: } \textbf{when } \beta^2 = l_4 \wedge y_2 < 7 \textbf{ do } \beta^2 := \omega, \\ & a_5\text{: } \textbf{when } \beta^2 = l_5 \wedge y_2 \leqq 6 \textbf{ do } \beta^2 := l_4\ y_1 := y_1 - 3\}. \end{aligned}$$

We do not note identity functions in actions.

Definition 2.1 implies the existence of pr and adr:

$$\text{pr} = (1, 1, 2, 2, 2), \quad \text{adr} = (l_1, l_2, l_3, l_4, l_5), \quad A^1 = \{a_1, a_2\}, \quad A^2 = A_1 - A^1,$$
$$\Gamma(z_{0_1}) = \{a_1, a_3\} = \Psi(z_{0_1}),$$
$$\Gamma(z_{0_1} * a_3) = \{a_1, a_4\} \supset \Psi(z_{0_1} * a_3) = \{a_4\},$$
$$\Sigma_{\text{SP}_1} = \{a_1 a_2 a_3 a_4,\ a_1 a_3 a_2 a_4,\ a_1 a_3 a_4 a_2,\ a_3 a_4\},$$
$$\begin{aligned} \Omega_{\text{SP}_1} = \{ & z_{0_1}, (l_2, l_3, 2, 3), (\omega, l_3, 2, 3), (\omega, l_4, 1, 3), (\omega, \omega, 1, 3), (l_2, l_4, 1, 3), \\ & (l_2, \omega, 1, 3), (l_1, l_4, 0, 3), (l_1, \omega, 0, 3)\}. \end{aligned}$$

To simplify the following discussion we need knowledge of the input/output variables of actions—see Miller and Yap [23] for an analogous definition. We say a pair of states z, z' are identical except on a variable $v \in V - \mathfrak{b}$ iff $\forall v' \in V - \{v\}\ (z(v') = z'(v'))$ and $z(v) \neq z'(v)$.

DEFINITION 2.3. Let $SP=(V, W, z_0, A)$, $a \in A$, $A' \subseteq A$, $\bar{V}=V-\mathfrak{b}$.

$$\begin{aligned} \mathrm{IN}\,(a)=\{v \in \bar{V} \mid \exists z, z' \in Z \text{ identical except on } v, a \in \Gamma(z) \cap \Gamma(z') \\ ((a \in \Psi(z) \wedge a \notin \Psi(z')) \\ \vee (a \in \Psi(z) \cap \Psi(z') \\ \wedge \exists v' \in V(z * a(v') \neq z' * a(v') \wedge z' * a(v') \neq z'(v'))))\}, \end{aligned}$$

$$\mathrm{OUT}\,(a)=\{v \in \bar{V} \mid \exists z \in Z, a \in \Psi(z)(z(v) \neq z * a(v))\},$$

$$\mathrm{IO}\,(a)=\mathrm{IN}\,(a) \cup \mathrm{OUT}\,(a),$$

$$\mathrm{IN}\,(A')=\bigcup_{a \in A'} \mathrm{IN}\,(a), \quad \mathrm{OUT}\,(A')=\bigcup_{a \in A'} \mathrm{OUT}\,(a),$$

$$\mathrm{IO}\,(A')=\mathrm{IN}\,(A') \cup \mathrm{OUT}\,(A').$$

A variable v is element of IN (a) iff there exists a pair of identical states except on v with a available in both states and

- either a is solely ready in one of the two states
- or a is ready in both states and a changes at least one variable.

It is not sufficient to state only $\exists v' \in V\ (z * a(v') \neq z' * a(v'))$ in the above definition since v' may be v.

A set of actions A' is *interference-free*, iff for every $a, b \in A'$, $a \neq b$

$$\mathrm{OUT}\,(a) \cap \mathrm{OUT}\,(b)=\varnothing \wedge \mathrm{OUT}\,(a) \cap \mathrm{IN}\,(b)=\varnothing \wedge \mathrm{IN}\,(a) \cap \mathrm{OUT}\,(b)=\varnothing$$

is valid.

DEFINITION 2.4. $SP=(V, W, z_0, A)$ is *legal* if and only if there exist pairwise disjoint sets A_R, A_N, $A_W \subseteq A$ and the following conditions hold for any $z, z' \in Z$:

(1) $A_R \cup A_N \cup A_W = A$;

(2) $\forall a \in A_N (a \in \Gamma(z) \Rightarrow a \in \Psi(z) \wedge \mathrm{IO}\,(a) \subseteq \{x_1, \cdots, x_n\})$;

(3) $\forall a^i \in A_R (a^i \in \Gamma(z) \Rightarrow a_i \in \Psi(z) \wedge \mathrm{IO}\,(a) \subseteq \{y_1, \cdots, y_m\}$
$\wedge\, a^i \in \Gamma(z) \cap \Gamma(z') \Rightarrow z * a^i(\beta^i) = z' * a^i(\beta^i))$;

(4) $\forall a^i \in A_W (\exists z'' \in Z(a^i \in \Gamma(z'') \wedge a^i \notin \Psi(z'')) \wedge \mathrm{IO}\,(a) \subseteq \{y_1, \cdots, y_m\}$
$\wedge\, a^i \in \Psi(z) \cap \Psi(z') \Rightarrow z * a^i(\beta^i) = z' * a^i(\beta^i))$;

(5) $\Gamma(z) \cap A_N$ is interference-free.

By this definition only the following actions are admissible in a legal system:

- The nonsynchronizers (A_N) are not synchronized and do not handle synchronization variables.
- The releasers (A_R) are never waiting but may release other actions by changing synchronization variables.
- The waiters (A_W) may wait on some predicate and handle only synchronization variables.

Executing a synchronizer—i.e., a releaser or waiter—always leads to the same state of the program counter, i.e., no branching by synchronizers is allowed. Since these restrictions as well as the interference-freeness reflect real parallel programming constructs such as semaphores [9], monitors [6] and message-passing [13], we call such systems legal systems.

Example 2.1a.

$$\mathrm{IN}(a_1)=\{y_1\}=\mathrm{OUT}(a_1),\quad \mathrm{IN}(a_2)=\{y_2\},\quad \mathrm{OUT}(a_2)=\varnothing,$$

$$\mathrm{IN}(a_3)=\{y_1\}=\mathrm{OUT}(a_3),\quad \mathrm{IN}(a_4)=\{y_2\},\quad \mathrm{OUT}(a_4)=\varnothing,$$

$$\mathrm{IN}(a_5)=\{y_1, y_2\},\quad \mathrm{OUT}(a_5)=\{y_1\},$$

$$A_R=\{a_3\},\quad A_W=A-A_R,\quad A_N=\varnothing,$$

SP_1 is legal.

We denote by $\Theta(z)$ the set of all possible transition sequences in the state z.

There are five basic properties valid in this model which we will not prove here. The proofs are given in [3].

THEOREM 2.1. *Let* $SP=(V, W, z_0, A)$, *for any* $i, j\in[1:k]$ *with* $i\neq j$ *and* $z\in Z$.

(A) $B^i\cap B^j=\varnothing$, *i.e., the sets of labels for process i and process j have no element in common.*

(B) $\#(\Gamma(z)\cap A^i)=1\vee z(\beta^i)=\omega$, *i.e., every process has exactly one available action or has already terminated.*

(C) $\forall\eta\in\Theta(z)(M(\eta)\cap A^i=\phi\Rightarrow\Gamma(z)\cap A^i=\Gamma(z*\eta)\cap A^i)$, *i.e., any transition sequence with no actions of a process i leaves the available set with respect to process i unchanged.*

(D) $\forall a\in\Psi(z),\ b\in\Gamma(z)-\Psi(z)(b\in\Psi(z*a)\Rightarrow\mathrm{OUT}(a)\cap\mathrm{IN}(b)\neq\varnothing)$, *i.e., any action b, which changes from available and not ready in z to ready in* $z*a$, *has at least one common variable in* OUT (a) *and* IN (a).

(E) $\forall\gamma a^ia^j\eta\in\Theta(z)(\{a^i, a^j\}$ *interference-free* $\Rightarrow\gamma a^ja^i\eta\in\Theta(z))$, *i.e., a transition sequence in z remains a transition sequence in z if successive interference-free actions* a^i, a^j $(i\neq j)$ *are interchanged.*

We will refer to these properties by their corresponding letters. Systems of processes often have additional features, which are defined as follows.

DEFINITION 2.5. Let $SP=(V, W, z_0, A)$ be legal. Then:

(1) SP is *deadlock-free* iff for any $\sigma\in\Sigma$ either σ is finite and the available-set $\Gamma(z_0*\sigma)$ is empty or σ is infinite.

(2) SP is *standard* iff for any $z\in Z$ and for any ready waiter a^i in z $\Psi(z)-A^i\supseteq\Psi(z*a^i)-A^i$ holds.

(3) SP is *uniform* iff for any $z\in Z$ and, for any available waiter a in z with $\mathrm{IN}(a)\subseteq\mathrm{IN}(\Psi')$, $\Psi'\subseteq\Psi(z)\cap A_W$, the action a is also ready in z.

(4) SP is *commutative* iff for any $z\in Z$ and any pair of synchronizers a, b with $ab, b\in\Theta(z)$ the implication $ba\in\Theta(z)\wedge z*ab=z*ba$ holds.

(5) SP is IO-*limited* iff for any synchronizer a $\mathrm{IN}(a)=\mathrm{OUT}(a)$ holds.

A system of processes is standard, iff for any state z the execution of a waiter moves no other waiter in the ready set; only very general synchronization mechanisms are nonstandard (see Table 1, § 4). A system of processes is called uniform, if for any state z the input variables are used in an uniform manner in the predicates of the waiters, i.e., if the input variables of an available waiter a form a subset of the input variables of other ready waiters we can conclude that this action is also ready.

Example 2.1b.

SP_1 is not deadlock-free since $\Gamma(z_0*a_3a_4)=\{a_1\}$.

SP_1 is not standard since $z=(l_1, l_5, 3, 4)$ and $\Psi(z)-A^2\not\supseteq\Psi(z*a_5)-A^2$.

SP_1 is uniform.

SP_1 is not commutative since $a_1a_3,\ a_3\in\Theta(z_0)\not\Rightarrow a_3a_1\in\Theta(z_0)$.

SP_1 is not IO-limited since $\mathrm{IN}(a_2)=\{y_2\}\neq\mathrm{OUT}(a_2)=\varnothing$.

We assume any system of processes legal for our further discussion. The subscripts N, R, W for any set S of actions restrict S to A_N, A_R or A_W ($S_N = S \cap A_N, \cdots$) and will be used mainly for the available- and ready-sets. Whenever possible we will abbreviate the notation for an action using the following conventions:

a^i: $P(y_j|c_j, \cdots, y_r|c_r)$, **goto** $\mathrm{adr}^{-1}(t)$;
instead of
a: **when** $\beta^i = l \wedge y_j \geqq c_j \cdots y_r \geqq c_r$ **do** $\beta^i := t \quad y_j := y_j - c_j \cdots y_r := y_r - c_r$;

a^i: $V(y_j|c_j, \cdots, y_r|c_r)$, **goto** $\mathrm{adr}^{-1}(t)$;
instead of
a: **when** $\beta^i = l \wedge$ **true do** $\beta^i := t \quad y_j := y_j + c_j \cdots y_r := y_y + c_r$;

a^i: $x_1 := \cdots, \cdots, x_n := \cdots$, **goto** $\mathrm{adr}^{-1}(t)$;
instead of
a: **when** $\beta^i = l \wedge$ **true do** $\beta^i := t \quad x_1 := \cdots, \cdots x_n := \cdots$.

In addition $y_j|1$ may be written as y_j and the **goto** $\cdots$ may be omitted if the specified label is always the label of the next action in the listing. A system of processes consisting only of actions which may be abbreviated according to the conventions with $c_j, \cdots, c_r \in \mathbb{N}$, is standard, commutative and IO-limited. In general such a system is neither uniform nor deadlock-free. A system SP is called a finite system if all executions have finite length.

3. Simulation of systems of processes. A system of processes simulates another system, if certain conditions hold in both systems. The choice of such conditions depends on the aspects of interest. Since we deal with synchronization mechanisms, we are interested in synchronization conditions. Synchronization problems are specifications of admitted or nonadmitted sequences of actions—see for example path expressions in [7]. This is the basis for the following definitions which are related to Lipton's [18].

DEFINITION 3.1. Let $\mathrm{SP} = (V, W, z_0, A)$ and $\mathrm{SP}' = (V', W', z_0', A')$ be systems of processes and rep: $A \cup \{\varepsilon\} \to A' \cup \{\varepsilon\}$ with $\mathrm{rep}(\varepsilon) = \varepsilon$.

SP *simulates* SP′ by rep iff $\{\mathrm{rep}(\sigma) | \sigma \in \Sigma_{\mathrm{SP}}\} = \Sigma_{\mathrm{SP}'}$.

Note, that the representation rep may represent some actions of A by ε—i.e., the empty word. Consequently, the system SP may have a lower level of abstraction and may have several actions to "simulate" one action of SP′. Since rep is not unique several actions of SP may represent the same action a of SP′. This makes sense when these actions are alternatives of the action on the higher level of abstraction. An action $a \in A$ is *visible* by rep, if a is not represented by ε—i.e., $\mathrm{rep}(a) \neq \varepsilon$.

Example 3.1. This example is an implementation of a general, not necessarily binary, semaphore by binary semaphores. Using the implementation in more than two processes could result in an error since the binary domains of the semaphores would no longer be guaranteed.

$$H = (V_H, W_H, z_{0_H}, A_H),$$

$$V_H = (\beta^1, \beta^2, y_1),$$

$$W_H = (\{l_1, l_2, \omega\}, \{l_3, l_4, \omega\}, \mathbb{N}_0),$$

$$z_{0_H} = (l_1, l_3, 0),$$

$A_H = \{b_1^1$: $P(y_1)$;

b_2^1: **goto** b_1^1;

b_3^2: $V(y_1)$;

b_4^2: **goto** b_3^2;$\}$,

$G = (V_G, W_G, z_{0_G}, A_G),$

$V_G = (\beta^1, \beta^2, x_1, y_1, y_2),$

$W_G = (\{l_1, l_2, l_3, l_4, l_5, l_6, \omega\}, \{l_7, l_8, l_9, l_{10}, l_{11}, \omega\}, \mathbb{Z}, \{0, 1\}, \{0, 1\}),$

$z_{0_G} = (l_1, l_7, 0, 1, 0),$

$A_G = \{a_1^1\colon P(y_1);$

$a_2^1\colon x_1 := x_1 - 1,$ **goto if** $x_1 < 1$ **then** a_3^1 **else** a_5^1; [1]

$a_3^1\colon V(y_1);$

$a_4^1\colon P(y_2),$ **goto** a_6^1;

$a_5^1\colon V(y_1);$

$a_6^1\colon$ **goto** a_1^1;

$a_7^2\colon P(y_1);$

$a_8^2\colon x_1 := x_1 + 1,$ **goto if** $x_1 \leqq -1$ **then** a_9^2 **else** a_{10}^2; [1]

$a_9^2\colon V(y_1, y_2),$ **goto** a_{11}^2;

$a_{10}^2\colon V(y_1);$

$a_{11}^2\colon$ **goto** $a_7^2;\}.$

Let $\text{rep}_1\colon A_G \cup \{\varepsilon\} \to A_H \cup \{\varepsilon\}$ with

$$\text{rep}_1(a) = \begin{cases} b_1^1 & \text{if } a = a_4^1, a_5^1, \\ b_2^1 & \text{if } a = a_6^1, \\ b_3^2 & \text{if } a = {}_9^2, a_{10}^2, \\ b_4^2 & \text{if } a = a_{11}^2, \\ \varepsilon & \text{otherwise.} \end{cases}$$

The proof that G simulates H by rep_1 is omitted but is similar to the proof of Theorem 5.5.

The simulation of Example 3.1 satisfies some other conditions which are in general necessary to keep an intuitive understanding of a simulation.

First of all we will define a measure of the parallelism of a system since this is the basis for our comparison.

DEFINITION 3.2. Let $\text{SP} = (V, W, z_0, A)$, $a^i, a^j \in A$, $\text{pnp} \subseteq A \times A$

(1) $(a^i, a^j) \in \text{pnp}$ if and only if $i = j$ or $\{a^i, a^j\}$ is not interference-free (pnp stands for potentially not parallel).

(2) pnp^t is the transitive closure of pnp.

(3) Since pnp^t is an equivalence relation, let MP_{SP} denote the number of equivalence classes implied by pnp^t.

To understand the motivation for this definition let us consider the parallelism of actions in a state z. Sometimes we find the statement that every set of ready actions

[1] Note, that all functions are executed collaterally, so that x_1 is tested for the "old" value.

is also parallel in z if every permutation of these actions leads to the same state. Such a statement does not reflect the atomicity of synchronization actions which work on the same variables. Consequently, we are forced to state that every set of interference-free, ready actions is a set of parallel actions as well. This corresponds directly to the above definition, since every two actions a_1, a_2 of different equivalence classes (implied by pnp^t) are interference-free and may be parallel in a state z, if a_1, a_2 are ready in z.

The equivalence classes have an additional characteristic if we assume SP deadlock-free. As long as no process has terminated, at least one available action a exists in every equivalence class—since $(a^i, a^j) \in \text{pnp}$ if $i = j$ and Property (B) holds. Now suppose a to be not ready. Because SP is deadlock-free a has to be ready at some time. Due to Property (D), another action working on at least one common variable must have changed a to be ready. Following this argument further we are able to conclude that in any state at least one action of every equivalence class is ready if no process has terminated yet. By this argument we are able to call MP the minimal parallelism of SP since MP_{SP} processors can work in parallel as long as no process has reached its end.

In some systems of processes this measure is too pessimistic, but we will use this measure solely for simulations of systems. If even this pessimistic measure changes in a simulation, it shows a drastic change in efficiency.

DEFINITION 3.3. Let SP simulate SP′ by rep. SP *simulates* SP′ *c-weakly* with $c \in \mathbb{N}_0$ iff

(a) for every pair of visible actions $a, b \in A$

$$\text{pr}(a) = \text{pr}(b) \Leftrightarrow \text{pr}(\text{rep}(a)) = \text{pr}(\text{rep}(b));$$

(b) SP′ deadlock-free $\Leftrightarrow$ SP deadlock-free;

(c) there exists an $e \in \mathbb{N}$ so that for any transition sequence η in z_0

$$\lg(\eta) \leqq (\lg(\text{rep}(\eta)) + 1) \cdot e;$$

(d) $\text{MP}_{\text{SP}} - g = \text{MP}_{\text{SP}'} - c$ and g denotes the number of equivalence classes with no visible action;

(e) for every visible action $a \in A$ $\text{rep}(a) \in A'_R \Rightarrow a \in A_R$;

Definition 3.3 needs further discussion.

Condition (a) is necessary to reflect the unity of a process in the simulation. If condition (a) does not hold, we can transform explicit synchronization, i.e., by synchronization operations, to sequencing in a process and vice versa. Inhibiting such transformations avoids garbling simulations.

Condition (c) states the simulation to be busy-wait-free. The addition of 1 is necessary if $\eta \neq \varepsilon$ and $\text{rep}(\eta) = \varepsilon$.

Because SP simulates c-weakly SP′ the system SP has a minimal parallelism reduced by c compared to SP′. The number of equivalence classes with no visible actions is not part of the comparison since these actions will not influence any important, i.e., visible action. If we omit the subtraction of g, every system SP simulating SP′ c-weakly may be extended to a system SP″, which simulates 0-weakly SP′. SP″ is easily constructed by adding c processes with no global variables and no visible actions to SP. Consequently, SP″ is still a simulation of SP′ and also a 0-weak one, by Definition 3.3.

Condition (e) is not necessary for our results but shortens the proofs.

Example 3.1a. $\text{MP}_H = 1$, $\text{MP}_G = 1$. G simulates H 0-weakly.

Example 3.2. At first glance this example looks rather artificial. This is due to the brevity of the example. A realistic, but more complex example is any simulation which introduces centralized control (e.g., by a kind of monitor).

$$H=(V_H, W_H, z_{0_H}, A_H),$$

$$V_H=(\beta^1, \beta^2, \beta^3, \beta^4, y_1, y_2),$$

$$W_H=(\{l_1, \omega\}, \{l_2, \omega\}, \{l_3, \omega\}, \{l_4, \omega\}, \{0,1\}, \{0,1\}),$$

$$z_{0_H}=(l_1, l_2, l_3, l_4, 0, 0),$$

$$\begin{aligned} A_H=\{&b_1^1\colon V(y_1), \mathbf{goto}\ \omega;\\ &b_2^2\colon P(y_1), \mathbf{goto}\ \omega;\\ &b_3^3\colon V(y_2), \mathbf{goto}\ \omega;\\ &b_4^4\colon P(y_2), \mathbf{goto}\ \omega;\ \},\end{aligned}$$

$$G=(V_G, W_G, z_{0_G}, A_G),$$

$$V_G=(\beta^1, \beta^2, \beta^3, \beta^4, x_1, x_2, y_1),$$

$$\begin{aligned} W_G=(&\{l_1, l_2, l_3, \omega\}, \{l_4, l_5, l_6, l_7, \omega\}, \{l_8, l_9, l_{10}, \omega\}, \{l_{11}, l_{12}, l_{13}, l_{14}, \omega\},\\ &\{\mathbf{true}, \mathbf{false}\}, \{\mathbf{true}, \mathbf{false}\}, \mathbb{N}_0),\end{aligned}$$

$$z_{0_G}=(l_1, l_4, l_8, l_{11}, \mathbf{false}, \mathbf{false}, 2),$$

$$\begin{aligned} A_G=\{&a_1^1\colon P(y_1|2);\\ &a_2^1\colon x_1 := \mathbf{true};\\ &a_3^1\colon V(y_1|3), \mathbf{goto}\ \omega;\\ &a_4^2\colon P(y_1|3);\\ &a_5^2\colon \boldsymbol{if}\ x_1\ \mathbf{then}\ a_7^2\ \mathbf{else}\ a_6^2;\\ &a_6^2\colon V(y_1|2), \mathbf{goto}\ a_4^2;\\ &a_7^2\colon V(y_1|3), \mathbf{goto}\ \omega;\\ &a_8^3\colon P(y_1|2);\\ &a_9^3\colon x_2 := \mathbf{true};\\ &a_{10}^3\colon V(y_1|3), \mathbf{goto}\ \omega;\\ &a_{11}^4\colon P(y_1|3);\\ &a_{12}^4\colon \boldsymbol{if}\ x_2\ \mathbf{then}\ a_{14}^4\ \mathbf{else}\ a_{13}^4;\\ &a_{13}^4\colon V(y_1|2), \mathbf{goto}\ a_{11}^4;\\ &a_{14}^4\colon V(y_1|3), \mathbf{goto}\ \omega;\ \},\end{aligned}$$

$$\mathrm{rep}_2(a)=\begin{cases} b_1^1 & \text{if } a=a_3^1,\\ b_2^2 & \text{if } a=a_7^2,\\ b_3^3 & \text{if } a=a_{10}^3,\\ b_4^4 & \text{if } a=a_{14}^3,\\ \varepsilon & \text{otherwise},\end{cases}$$

$$\mathrm{pnp}_H^t=\{(b_1^1, b_1^1), (b_1^1, b_2^2), (b_2^2, b_1^1), (b_2^2, b_2^2), (b_3^3, b_3^3), (b_3^3, b_4^4), (b_4^4, b_3^3), (b_4^4, b_4^4)\},$$

$$\mathrm{PNP}_G^t = \{(a, a') | a, a' \in A_G\},$$

$$\mathrm{MP}_H = 2, \qquad \mathrm{MP}_G = 1.$$

G simulates 1-weakly H by rep_2.

Now we are able to express our understanding of the power of synchronization mechanisms.

DEFINITION 3.4. Let $\mathfrak{P}$ and $\mathfrak{Q}$ denote sets of systems of processes and $c \in \mathbb{N}$ then:

(a) $\mathfrak{P} \leqq \mathfrak{Q}$ iff $\forall P \in \mathfrak{P}(\exists Q \in \mathfrak{Q}(Q$ simulates 0-weakly $P))$;

(b) $\mathfrak{P} \underset{c}{<} \mathfrak{Q}$ iff $\mathfrak{P} \leqq \mathfrak{Q} \wedge \exists Q \in \mathfrak{Q}(\nexists P \in \mathfrak{P}, g \in [0\colon c-1]$ $(P$ simulates g-weakly $Q))$;

(c) $\mathfrak{P} \lhd \mathfrak{Q}$ iff $\forall c' \in \mathbb{N}(\mathfrak{P} \underset{c'}{<} \mathfrak{Q})$.

If $\mathfrak{P} \underset{c}{<} \mathfrak{Q}$ holds, the loss of parallelism simulating a system of $\mathfrak{Q}$ by a system of $\mathfrak{P}$ can in general be c.

Without proof we state a partial transitivity of $\underset{c}{<}$.

THEOREM 3.1. *Let $\mathfrak{P}$, $\mathfrak{Q}$ and $\mathfrak{R}$ be sets of systems of processes, $s, t \in \mathbb{N}$. Then*:

(a) $\mathfrak{P} \leqq \mathfrak{Q} \underset{s}{<} \mathfrak{R} \Rightarrow \mathfrak{P} \underset{s}{<} \mathfrak{R}$;

(b) $\mathfrak{P} \underset{t}{<} \mathfrak{Q} \leqq \mathfrak{R} \Rightarrow \mathfrak{P} \underset{t}{<} \mathfrak{R}$;

(c) $\mathfrak{P} \underset{t}{<} \mathfrak{Q} \underset{s}{<} \mathfrak{R} \Rightarrow \mathfrak{P} \underset{\max(s,t)}{<} \mathfrak{R}$.

The reader may wonder why the implication in (c) is not $\mathfrak{P} \underset{s+t}{<} \mathfrak{R}$. Assume $R' \in \mathfrak{R}$ to be a system which can be s-weakly simulated by $Q' \in \mathfrak{D}$ and $\mathfrak{D} \underset{s}{<} \mathfrak{R}$. In general we do not know if Q' is just the system by which $\mathfrak{P} \underset{t}{<} \mathfrak{D}$ is established.

4. General properties of semaphore-mechanisms. For the proofs in § 5 we need some general properties of semaphore-mechanisms.

4.1. Relations of actions. We are interested in relations of actions which allow an interchange of actions. Lipton [21] has introduced the notion of the left- and right-mover property:

An action $a\colon P(y_j)$ is a right-mover, since for every execution $\alpha a b \beta$ with $\mathrm{pr}(a) \neq \mathrm{pr}(b)$ $\alpha b a \beta$ is an execution as well.

An action $a\colon V(y_j)$ is a left-mover, since for every execution $\alpha b a \beta$ with $\mathrm{pr}(a) \neq \mathrm{pr}(b)$ $\alpha a b \beta$ is an execution as well.

Kowalk and Valk [15] have somewhat extended his notion and put it in a more formal framework. We will treat mover properties in a more general way and the right-(resp. left-) mover properties will only be special cases.

DEFINITION 4.1. Let $\mathrm{SP} = (V, W, z_0, A)$, and ρ be a predicate on A^*. Then:

(1) $\rho(a)$ is true;

(2) $\rho(a^i \eta a^j) \Leftrightarrow (i = j$

$$\vee \exists b^i \in A_R^i, b^k \in A_W^k, \eta_1, \eta_2, \eta_3 \in A^*$$
$$(a^i \eta a^j = \eta_1 b^i \eta_2 b^k \eta_3 \wedge \mathrm{OUT}(b^i) \cap \mathrm{IN}(b^k) \neq \varnothing \wedge \rho(b^k \eta_3))).$$

If $\rho(a \eta b)(\rho(a))$ is true, we say a and $b(a)$ are *related* by $\eta(\varepsilon)$.

If two actions a and b are related by η, these actions may have been serialized by synchronization actions or by the implicit serialization in processes. For example a and b are related by ε, if and only if $\mathrm{pr}(a) = \mathrm{pr}(b)$ or $a \in A_R \wedge b \in A_W \wedge \mathrm{OUT}(a) \cap \mathrm{IN}(b) \neq \varnothing$. In this example b could have waited for the action a to occur. If a and b are not related by η, we know that there is no chance for a to serialize b. This consideration is the basis for the above definition and the following theorem.

THEOREM 4.1 (movability of actions). *Let* $SP=(V, W, z_0, A)$ *be standard and commutative. Then*

$$\forall \alpha a^i\beta a^j\gamma \in \Sigma(\neg\rho(a^i\beta a^j) \Rightarrow \exists \alpha\beta' a^j a^i \beta''\gamma \in \Sigma(\beta'\beta'' \in \text{PERMUTATIONS}(\beta))).$$

For the proof we need the following lemma:

LEMMA 4.2. *Let* $SP=(V, W, z_0, A)$ *be standard and commutative. If* $\alpha a^i\beta a^j\gamma$ *is an execution of* SP *with* $i\neq j$, *the following holds*

$$\begin{aligned}&\exists b^i \in A_R^i, b^j \in A_W^j, \eta_1, \eta_2, \eta_3 \in A^*\\ &(a^i\beta a^j = \eta_1 b^i \eta_2 b^j \eta_3 \wedge \text{OUT}(b^i) \cap \text{IN}(b^j) \neq \emptyset)\\ &\Rightarrow \alpha(\beta \| \{a^l \in A | l \neq i\}) a^j a^i (\beta \| A^i)\gamma \in \Sigma.\end{aligned}$$

Proof. Let d^i denote the last action of A^i in the sequence $a^i\beta$, i.e., $a^i\beta a^j = \mu_1 d^i e \mu_2$. The precondition implies $e\mu_2 \neq \varepsilon$ and $i \neq \text{pr}(e)$.

By (C) (1) $d^i, e \in \Gamma(z)$ with $z = z_0 * \alpha\mu_1$ holds. We denote the property $\alpha\mu_1 e d^i \mu_2 \gamma \in \Sigma$ by (2).

Case 1. $d^i, e \in A_N$. Since SP is legal (1) implies d^i, e is interference-free. By Property (E) (2) is true.

Case 2. $d^i \in A_N$, $e \in A_W \cup A_R$ (similarly $d^i \in A_W \cup A_R$, $e \in A_N$). Since SP is legal $\text{IO}(d^i) \cap \text{IO}(e) = \emptyset$ and (E) implies (2).

Case 3. $d^i, e \in A_R$. By (F) $d^i, e \in \Psi(z)$. Since SP is commutative (2) holds.

Case 4. $d^i, e \in A_W$. As $d^i \in \Psi(z)$, $d_2 \in \Gamma(z)$ and $d_2 \in \Psi(z * d^i)$ the feature standard implies $d_2 \in \Psi(z)$. Since SP is commutative (2) holds.

Case 5. $d^i \in A_R$, $e \in A_W$. The precondition states $\text{OUT}(d^i) \cap \text{IN}(e) = \emptyset$. By (D) and the feature commutative (2) is implied.

Case 6. $d^i \in A_W$, $e \in A_R$. Consequently, $d^i, e \in \Psi(z)$ and the feature commutative implies (2).

Thus $\alpha\mu_1 e d^i \mu_2 \gamma \in \Sigma$ for any e, d^i. This series of actions may be repeated until $\mu_2 = \varepsilon$. As a result the execution $\alpha\mu_1 e \mu_2 d^i \gamma$ exists. As long as other actions of process i exist in μ_1 the above transformation may be repeated. The last moved action is a^i. Since the last action of μ_2 is still a^j the lemma holds. □

Now the proof of Theorem 4.1 is obvious, since the negation of $\rho(a\beta b)$ is only a repetition of the argument in Lemma 4.2 on several processes. Consequently, we will omit this induction proof here.

The benefit of ρ for our proceeding further is shown by the following theorem:

THEOREM 4.3. *Let* $SP=(V, W, z_0, A)$ *be standard and commutative and* $\alpha a\beta b\gamma \in \Sigma$. *Then*

$$\rho(a\beta b) \Rightarrow (a, b) \in \text{pnp}^t.$$

The proof of this theorem is easily established by looking at the pairs of noninterference-free actions relating a and b by Definition 4.1. On the other hand let a and b be actions of different equivalence classes implied by pnp^t. From this condition we may derive by Theorem 4.3 that for every execution a and b are not related at all.

The next two theorems will show that our proceedings includes the right- and left-mover properties:

THEOREM 4.4. *Let* SP *be standard and commutative and* $\alpha a^i\beta\gamma \in \Sigma$. *Then*

$$M(a^i\beta) \cap A_R^i = \emptyset \Rightarrow \alpha(\beta \| \{a \in A | \text{pr}(a) \neq i\}) a^i (\beta \| A^i)\gamma \in \Sigma.$$

THEOREM 4.5. *Let* SP *be standard and commutative and* $\alpha\beta a^i\gamma \in \Sigma$. *Then*

$$M(\beta a^i) \cap A_W^i = \emptyset \Rightarrow \alpha(\beta \| A^i) a^i (\beta \| \{a \in A | \text{pr}(a) \neq i\})\gamma \in \Sigma.$$

The proofs of both Theorem 4.4 and Theorem 4.5 require that any pair of actions b^i, b^j in the sequence βa^i not be related.

The substitution of $\beta = a^j_w, a^j_w \in A^j_W (\beta = a^j_r, a^j_r \in A_R)$ in Theorem 4.4 (4.3) gives us the right- and left-mover properties:

$$\alpha a^i a^j_w \gamma \in \Sigma \wedge i \neq j \Rightarrow \alpha a^j_w a^i \gamma \in \Sigma$$

$$(\alpha a^j_r a^i \gamma \in \Sigma \wedge i \neq j \Rightarrow \alpha a^i a^j_r \gamma \in \Sigma).$$

Throughout this chapter we have assumed every system of processes to be standard and commutative. For the main semaphore-mechanisms we list their features (standard, commutative, IO-limited, uniform) in Table 1 without proof ($c, c_1, \cdots, c_r \in \mathbb{N}$).

TABLE 1

Mechanisms	Predicates	Operations	Features
PV [9]	$\beta^i = l \wedge y_j \geqq 1$ $\beta^i = l$	$y_j := y_j - 1$ $y_j := y_j + 1$	standard, comm. IO-lim., uniform
PV-chunk [26]	$\beta^i = l \wedge y_j \geqq c$ $\beta^i = l$	$y_j := y_j - c$ $y_j := y_j + c$	standard, comm. IO-lim.
PV-multiple [24]	$\beta^i = l \wedge y_{j1} \geqq 1, \cdots, y_{jr} \geqq 1$ $\beta^i = l$	$y_{j1} := y_{j1} - 1, \cdots, y_{jr} := y_{jr} - 1$ $y_{j1} := y_{j1} + 1, \cdots, y_{jr} := y_{jr} + 1$	standard, comm. IO-lim., uniform
PV-general [8]	$\beta^i = l \wedge y_{j1} \geqq c_1, \cdots, y_{jr} \geqq c_r$ $\beta^i = l$	$y_{j1} := y_{j1} - c_1, \cdots, y_{jr} := y_{jr} - c_r$ $y_{j1} := y_{j1} + c_1, \cdots, y_{jr} := y_{jr} + c_r$	standard, comm. IO-lim.
UP-DOWN [4]	$\beta^i = l \wedge \sum_{j \in \{j1, \cdots, jm\}} y_j \geqq 0$ $\beta^i = l \wedge \sum_{j \in \{j1, \cdots, jm\}} y_j \geqq 0$	$y_k := y_k + 1$ $y_k := y_k - 1$	
PV-extended [1]	$\beta^i = l \wedge y_{j1} \geqq 1, \cdots, y_{jr} \geqq 1 \wedge y_{jr'} = 0, \cdots, y_{jr''} = 0$ $\beta^i = l$	$y_{j1} := y_{j1} - 1, \cdots, y_{jr} := y_{jr} - 1$ $y_{j1} := y_{jr} + 1, \cdots, y_{jr} := y_{jr} + 1$	IO-lim.
LOCK/UNLOCK [9]	$\beta^i = l \wedge y_j = 1$ $\beta^i = l$	$y_j := 0$ $y_j := 1$	standard, uniform IO-lim.

Note, that the above definitions of PV-operations are so-called weak PV-operations [11], [25]. Strong (resp. blocked-set) PV-operations [11], [25] include additionally a fairness characteristic (resp. a preference to already waiting actions).

4.2. Uniform systems of processes. In this subsection we will prove a characteristic of uniform systems which are necessary for the proofs in the next chapter. The following theorem is only valid for deadlock-free systems. Since the definition of deadlock-free contains dynamic properties of a system, we have to restrict the choice of the states to the reachable states.

THEOREM 4.6. *Let SP be uniform, deadlock-free and finite,* PNP *an equivalence class induced by* pnp^t *where all actions of* PNP *work at most on one synchronizer variable* y_j $(\text{IO}(\text{PNP} - A_N) = \{y_j\})$, *z a reachable state* $(z \in \Omega)$. *Then for every action* $a^i \in \text{PNP} \cap \Gamma_W(z)$ *there exists a transition sequence* η *in z so that every other process*

$A^j \subseteq$ PNP $(j \neq i)$ has already terminated in $z * \eta$ $(\text{pr}(\Gamma(z*\eta)) \cap (\text{pr}(\text{PNP}) - \{i\}) = \emptyset)$ *but there is no action of process i that has occurred in η* $(i \notin \text{pr}(M(\eta)))$.

Proof. We will construct the transition sequence η. Let $\Gamma' = (\Gamma(z) \cap \text{PNP}) - \{a^i\}$ denote all available actions of PNP in z without a^i. If $\Gamma' = \emptyset$, $\text{pr}(\Gamma(z*\eta)) \cap (\text{pr}(\text{PNP}) - \{i\}) = \emptyset \wedge i \notin \text{pr}(M(\eta))$ is trivially satisfied by $\eta = \varepsilon$.

Assume $\Gamma' \cap \Psi(z) = \emptyset$ with $\Gamma' \neq \emptyset$. Consequently, an action $b \in \Gamma' \cap A_W$ working on the variable y_j exists $(\text{IN}(b) = \{y_j\})$. Since SP is uniform the action a^i is not ready in z. (Assume a^i ready. This implies $b \in \Psi(z)$ since SP is uniform.) Therefore, $\Psi(z) \cap$ PNP is empty. Since SP is deadlock-free there has to be a transition sequence γ in z with $b \in \Psi(z*\gamma) \cap \text{PNP}$ and $M(\gamma) \cap \text{PNP} = \emptyset$. Because of Property (D) there exists an action $c \in M(\gamma)$ with $\text{OUT}(c) \cap \text{IN}(b) \supseteq \{y_j\}$. The action c has to be an element of PNP because $\{c, b\}$ is not interference-free. This contradicts $M(\gamma) \cap \text{PNP} = \emptyset$. Moreover, it contradicts $\Gamma' \cap \Psi(z) = \emptyset$. This way we have found an action $b \in \Gamma' \cap \Psi(z)$. This is the first action of η. With a finite repetition of the above (SP is finite) we can construct the transition sequence η only by actions of processes $\text{pr}(\text{PNP}) - \{i\}$. □

Note that Theorem 4.6 is only valid because we did not assume any fair scheduling discipline. The theorem could be refined somewhat if we want to add fairness requirements [16].

5. Power of semaphore-mechanisms. Here we will give the results according to our definition of power of mechanisms (Definition 3.4). Other definitions of power may result in other hierarchies, see § 6.

By $\mathfrak{PV}$ we denote the set of all PV-systems according to Table 1. Comparing PV-systems with other semaphore-mechanisms, we observe that they differ in power only if we limit the number of semaphores and/or their domain. Consequently, we denote by $\mathfrak{PV}(d, m)$ the set of all PV-systems with at most m synchronizer variables $y_1, \cdots, y_m$ out of a domain $[0{:}\ d]$. Therefore, $\mathfrak{PV}(1, m)$ is the set of systems using only binary semaphores [9].

Since PV-systems are IO-limited, the sets $\text{AV}_j = \{a \in A \mid y_j \in \text{IO}(a)\}$ are disjoint for different j. We augment AV_j if necessary by the use of subscripts R, W according to our previously introduced conventions. The next theorem characterizes $\mathfrak{PV}(d, m)$-systems.

THEOREM 5.1. *Suppose $m, d \in \mathbb{N}$ and $P \in \mathfrak{PV}(d, m)$. Then for any transition sequence η in a reachable state z of Ω and any $j \in [1{:}\ m]$ the following implication holds*

$$\lg(\eta \mid \text{AV}_{j,R}) > d \Rightarrow \exists a \in \text{AV}_{j,W},\ b, c \in \text{AV}_{j,R},\ \eta_1, \eta_2, \eta_3, \eta_4 \in A^* \quad (\eta_1 b \eta_2 a \eta_3 c \eta_4 = \eta \wedge \rho(a \eta_3 c)).$$

Proof. Assume first that there is no $a \in \text{AV}_{j,W}$ in η $(\text{AV}_{j,W} \cap M(\eta) = \emptyset)$. Since $z(y_j) \in [0{:}\ d]$ and $\lg(\eta \| \text{AV}_{j,R}) > d$ leads to $z*\eta(y_j) > d$, P is not an element of $\mathfrak{PV}(d, m)$. Consequently, our assumption is false and we can write η as

$$\eta = \alpha a \beta.$$

Suppose that α does not contain an element of $\text{AV}_{j,R}(M(\alpha) \cap \text{AV}_{j,R} = \emptyset)$. This leads to $\lg(\beta \| \text{AV}_{j,R}) > d$. We may conclude that there has to be an $a \in \text{AV}_{j,W}$ in η with an action of $\text{AV}_{j,R}$ before it. In the same way the assumption $M(\beta) \cap \text{AV}_{j,R} = \emptyset$ is a contradiction.

Therefore, we can rewrite η, without loss of generality, as

$$\eta = \eta_1 b \eta_2 a \eta_3 c \eta_4, \qquad b, c \in \text{AV}_{j,R}.$$

Let us assume that for any such $c \neg \rho(a \eta_3 c)$ holds. Applying Theorem 4.1 to all actions

$c \in \mathrm{AV}_{j,R}$ gives us

$$\eta' = \eta_1 b \eta_5 a \eta_6 \in \Sigma \wedge \lg(\eta_1 b \eta_6 \| \mathrm{AV}_{j,R}) > d.$$

Therefore, we can conclude that there exists an a and c with $\rho(a\eta_3 c)$. □

On the basis of this characteristic of $\mathfrak{PV}(d, m)$ we are able to prove the main theorems of PV-systems.

THEOREM 5.2 (power concerning variables). *Let $d, m \in \mathbb{N}$. Then*

$$\mathfrak{PV}(d, m) \underset{2}{<} \mathfrak{PV}(d, m+1).$$

THEOREM 5.3 (power concerning domain). *Let $d, m \in \mathbb{N}$. Then*

$$\mathfrak{PV}(d, m) \underset{1}{<} \mathfrak{PV}(d+1, m).$$

Here we will only deal with Theorem 5.2. The theorem states:

(1) Any PV-system with m semaphores in a domain $[0: d]$ may be simulated by a PV-system with $m+1$ semaphores in a domain $[0: d]$ without a change in minimal parallelism

(2) There exists a PV-system with $m+1$ semaphores in a domain $[0: d]$ which may only be simulated by a PV-system with m semaphores in a domain $[0: d]$ and a minimal parallelism reduced by 2.

Stating that every system may in principle be simulated, is a typical result. However, our results emphasize a different quality—parallelism.

Proof of Theorem 5.2. *Case* 1. $\mathfrak{PV}(d, m) \leqq \mathfrak{PV}(d, m+1)$. Since $\mathfrak{PV}(d, m) \subseteq \mathfrak{PV}(d, m+1)$ and every system $P \in \mathfrak{PV}(d, m)$ simulates itself 0-weakly, the statement is immediately valid.

Case 2. $\exists Q \in \mathfrak{PV}(d, m+1)$. $(\nexists P \in \mathfrak{PV}(d, m)(P$ simulates 0-weakly $Q \vee P$ simulates 1-weakly $Q))$.

We will consider the following system:

$$\begin{aligned}
&Q = (V, W, z_0, A),\\
&V = (\beta^1, \cdots, \beta^{2\cdot(m+1)}, y_1, \cdots, y_{m+1}),\\
&W = (\{l_1, \cdots, l_d, \omega\}, \cdots, [0: d], \cdots, [0: d]),\\
&z_0 = (l_1, l_{d+1}, \cdots, 0, \cdots, 0).
\end{aligned}$$

A is given informally for ease of understanding:

$$\begin{array}{ll}
A^1\colon a_1^1\colon V(y_1); & A^2\colon a_1^2\colon P(y_1);\\
\quad\vdots & \quad\vdots\\
\quad a_d^1\colon V(y_1), \textbf{goto}\ \omega; & \quad a_d^2\colon P(y_1), \textbf{goto}\ \omega;\\
\cdots A^{2i-1}\colon a_1^{2i-1}\colon V(y_i); & A^{2i}\colon a_1^{2i}\colon P(y_i);\\
\quad\vdots & \quad\vdots\\
\quad a_d^{2i-1}\colon V(y_i), \textbf{goto}\ \omega; & \quad a_d^{2i}\colon P(y_i), \textbf{goto}\ \omega;
\end{array}$$

with $i \in [1: m+1]$.

The subscripts of the actions do not correspond to our conventions, since a_1^2 should be noted as a_{d+1}^2 or in general a_r^j should be noted as $a_{(j-1)\cdot d+r}^j$. However, this notation

is unique and more convenient. Several features of Q are obvious:

$Q \in \mathfrak{PB}(d, m+1)$;

Q is deadlock-free and finite;

$\text{MP}_Q = m+1$, where the equivalence classes implied by pnp_Q^t are

$$\text{PNP}_1 = A^1 \cup A^2 \cdots \text{PNP}_i = A^{2i-1} \cup A^{2i} \cdots.$$

First we assume a system $P = (V', Q', z_0', A')$ to simulate Q 0-weakly by rep with $P \in \mathfrak{PB}(d, m)$. Any execution of P incorporates two actions $\underline{a}_1^{2i-1}$, $\underline{a}_1^{2i}$ with $\text{rep}(\underline{a}_1^{2i-1}\underline{a}_1^{2i}) = a_1^{2i-1}a_1^{2i}$, since P simulates Q by rep[2]. Furthermore, the simulation specification implies $\rho(\underline{a}_1^{2i-1}\eta\underline{a}_1^{2i})$ for any transition sequence $\underline{a}_1^{2i-1}\eta\underline{a}_1^{2i}$ by Theorem 4.1. Theorem 4.3 and Definition 3.2 imply the existence of the following equivalence classes (by pnp_P^t):

$$\text{PNP}_1' \subseteq A'^1 \cup A'^2 \cdots \text{PNP}_i' \subseteq A'^{2i-1} \cup A'^{2i} \cdots, \qquad i \in [1: m+1].$$

Since other equivalence classes cannot contain visible actions due to Definitions 3.3(a), (d) and 3.2 we are only concerned with the above ones. If all these equivalence classes are different, they have no common variables. Since $\rho(\underline{a}_1^{2i-1}\eta\underline{a}_1^{2i})$ implies the existence of at least one synchronizer variable in processes $2i-1$ and $2i$ and since there are only m synchronizer variables available in P, the above equivalence classes are not all different.

This contradicts $\text{MP}_P - g = m+1$, where g denotes the number of equivalence classes with no visible action.

Now we assume P to simulate Q 1-weakly by rep. The notation PNP will stand for the equivalence class, which has to contain at least four processes as stated above. Without loss of generality, let A'^1, A'^2, A'^3, A'^4 be these four processes. Suppose that the actions of PNP use more than one synchronizer variable ($\#\,\text{IO}\,(\text{PNP} - A_N) > 1$). This contradicts immediately $MP_P - g = m$ because of what we stated above, since the other $m-1$ equivalence classes have only $m-2$ synchronizer variables at their disposal. Consequently, $\text{IO}\,(\text{PNP} - A_N) = \{y_j\}$, $j \in [1: m]$ holds.

Let us consider the following execution:

$$\sigma' = \eta_1 \underbrace{\underline{a}_1^1 \cdots \underline{a}_d^1 \cdots \underline{a}_1^3 \cdots \underline{a}_d^3}_{\gamma} \underbrace{\cdots \underline{a}_1^2 \cdots \underline{a}_d^2 \cdots \underline{a}_1^4 \cdots \underline{a}_d^4}_{\delta} \eta_2 \in \Sigma_P$$

with $\text{rep}\,(\underline{a}_j^i) = a_j^i$.

Since $a_j^i \in A_R'(\text{e})$ of Definition 3.3 implies $\underline{a}_j^i \in A_R$ [3], the statement $\text{lg}\,(\gamma \| \text{AV}_{j,R}) > d$ holds. By Theorem 5.1 we conclude

$$\exists a \in \text{AV}_{j,W},\, b, c \in \{\underline{a}_1^1, \cdots, \underline{a}_d^3\},\, \gamma_1, \gamma_2, \gamma_3, \gamma_4 \in A^*(\gamma_1 b \gamma_2 a \gamma_3 c \gamma_4 =_{\cdot} \gamma \wedge \rho(a\gamma_3 c)).$$

Let us assume that every such a is not an element of processes 1 and 3. Since a and c are related by γ_3, there has to be an action $a' \in A_W^{\text{pr}(c)} \cap M(\gamma_3)$. However, a' is related to c and therefore the assumption is false. Without loss of generality, let $\text{pr}\,(a) = 3$. We rewrite σ' so that

$$\sigma' = \eta_1 \gamma_{12} a^3 \gamma_{34} \eta_2 \wedge M(\gamma_{34}) \cap \{\underline{a}_1^3, \cdots, \underline{a}_d^3\} \neq \varnothing \wedge a^3 \in \text{AV}_{j,W}.$$

[2] Without loss of generality, we assume in this proof $\text{pr}\,(a) = \text{pr}\,(\text{rep}\,(a))$ for every visible action.

[3] In connection with Definition 3.3 we have noted that condition (e) is not necessary, but that we have added it to simplify our proofs. This is the case here, since by additional considerations, we could also conclude $\underline{a}_j^i \in A_R$.

Taking $z = z_0' * \eta_1 \gamma_{12}$, we see that all preconditions of Theorem 4.6 are satisfied. Consequently, there exists a transition sequence β_1 with

$$\bar{\sigma} = \eta_1 \gamma_{12} \beta_1 \beta_2 \wedge M(\beta_2) \cap (A_2' \cup A_4') = \emptyset \wedge M(\beta_1) \cap A^3 = \emptyset.$$

Since rep $(\bar{\sigma})$ has to be an execution in Q, every action of A has to be an element of rep $(\bar{\sigma})$. In rep $(\eta_1 \gamma_{12})$ at least one a_i^3 is missing by the construction of σ'. However, all actions $\{a_1^2, \cdots, a_d^2, a_1^4, \cdots, a_d^4\}$ are part of rep (β_1). This contradicts the simulation specification. This again contradicts the assumption, that P simulates Q 1-weakly. □

The reader might be interested now to see a 2-weak simulation of Q. We will show the principle of such a simulation for $m = 3$ and $d = 1$.

Example 5.1.

$Q = (V, W, z_0, A)$,

$V = (\beta^1, \beta^2, \beta^3, \beta^4, \beta^5, \beta^6, y_1, y_2, y_3)$,

$W = (\{l_1, \omega\}, \{l_2, \omega\}, \{l_3, \omega\}, \{l_4, \omega\}, \{l_5, \omega\}, \{l_6, \omega\}, [0{:}\ 1], [0{:}\ 1], [0{:}\ 1])$,

$z_0 = (l_1, l_2, l_3, l_4, l_5, l_6, 0, 0, 0)$,

$A = \{a_1^1$: $V(y_1)$, **goto** ω;
a_2^2: $P(y_1)$, **goto** ω;
a_3^3: $V(y_2)$, **goto** ω;
a_4^4: $P(y_2)$, **goto** ω;
a_5^5: $V(y_3)$, **goto** ω;
a_6^6: $P(y_3)$, **goto** $\omega\}$.

To facilitate the understanding we denote P rather informally. Every $V(y_i)$-operated is substituted in P by

$P(y_1)$;
$x_i :=$ **true**;
if $x_4 = 0$ **then** $\boxed{V(y_1)}$ **else** $\boxed{V(y_2)}$;

and every $P(y_1)$-operation is substituted by

label: $P(y_1)$;
  if not x_i **then**
    begin
    $x_4 := x_4 + 1$; $V(y_1)$;
    $P(y_2)$; $x_4 := x_4 - 1$;
    if $x_4 = 0$ **then** $V(y_1)$ **else** $V(y_2)$;
    goto *label*;
    end;
  $\boxed{V(y_1);}$

The above framed actions are the visible actions. At a first glance the reader may doubt that the system P is busy-wait-free (Definition 3.3(c)). However, the processes can repeat a test of x_i if and only if at least one other process has executed the substitute for $V(y_i)$. Since only a limited number of $V(y_i)$'s exist, the test on x_i is also limited. Note, that the simulation uses split-binary-semaphores [10].

Without proofs we list in Fig. 1 the hierarchy of semaphore-systems, where

- $\mathfrak{PVC}(d, m)$ denotes PV-chunk systems with m variables and domain $[0{:}\ d]$;
- $\mathfrak{PVM}(d, m)$ denotes PV-multiple systems with m variables and domain $[0{:}\ d]$;
- $\mathfrak{PVG}(d, m)$ denotes PV-general systems with m variables and domain $[0{:}\ d]$;
- $\mathfrak{PVE}(d, m)$ denotes PV-extended systems with m variables and domain $[0{:}\ d]$;
- a directed arc from $\mathfrak{P}$ to $\mathfrak{Q}$ labeled by c denotes $\mathfrak{Q} \underset{c}{<} \mathfrak{P}$;
- a directed arc from $\mathfrak{P}$ to $\mathfrak{Q}$ without a label denotes $\mathfrak{Q} \lhd \mathfrak{P}$;
- an undirected, unlabeled arc between $\mathfrak{P}$ and $\mathfrak{Q}$ denotes $\mathfrak{Q} \leqq \mathfrak{P} \wedge \mathfrak{P} \leqq \mathfrak{Q}$;
- $m, d \in \mathbb{N}$;
- $\lfloor x \rfloor$ is the smallest number ($\in \mathbb{N}$) with $\lfloor x \rfloor \geqq x$.

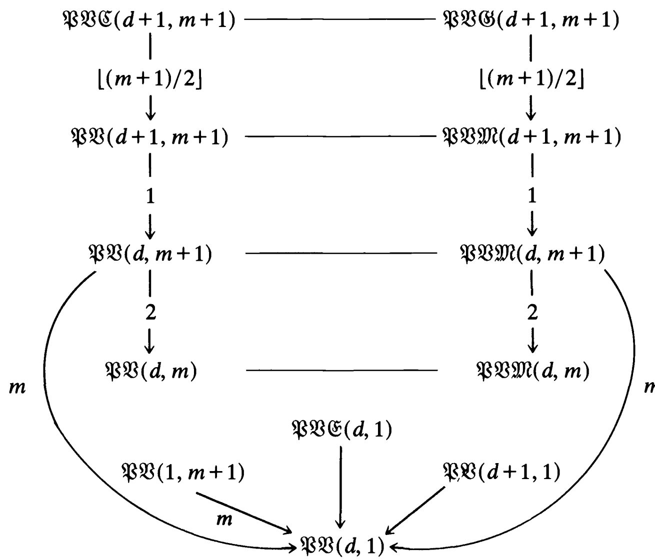


FIG. 1.

Here, we will treat only those proofs of the stated results, which are not simple analogies of the proof of Theorem 5.2.

THEOREM 5.4. *Let $d, m \in \mathbb{N}$ and $d > 1$. Then*

$$\mathfrak{PVC}(d, m) \underset{\lfloor m/2 \rfloor}{>} \mathfrak{PV}(d, m).$$

Sketch of the proof of Theorem 5.4. We use a system $P \in \mathfrak{PVC}(2, m)$ with the following actions:

$$A_P{:}\ a_1^1{:}\ P(y_1); \qquad a_1^2{:}\ V(y_1);$$

$$a_2^1{:}\ V(y_1|2), \textbf{goto}\ \omega; \qquad a_2^2{:}\ P(y_1|2), \textbf{goto}\ \omega;$$

$$\dots \qquad a_1^{2m-1}{:}\ P(y_m); \qquad a_1^{2m}{:}\ V(y_m);$$

$$a_2^{2m-1}{:}\ V(y_m|2), \textbf{goto}\ \omega; \qquad a_2^{2m}{:}\ P(y_m|2), \textbf{goto}\ \omega;$$

$z_0(y_i) = 0$, $i \in [1{:}\ m]$.

The system P is not uniform because $\mathrm{IN}\,(a_2^2) \subseteq \mathrm{IN}\,(\Psi(z_0 * a_1^2) \cap A_W)$ does not imply a_2^2 ready in $z_0 * a_1^2$. This verifies one feature of PV-chunk systems in Table 1. Now suppose $Q \in \mathfrak{PV}(d, m)$ is a system which simulates P c-weakly. First we tackle the

assumption that any pair of processes A_Q^{2i-1} and A_Q^{2i} ($i \in [1: m]$) works only on one synchronization variable. Since the simulation specification and the busy-wait-free condition have to hold, every A_Q^{2i-1} has to have a waiter action. Consequently, Theorem 4.6 contradicts the simulation specification and the assumption cannot hold.

Therefore any pair of processes A_Q^{2i-1} and A_Q^{2i} works on at least two synchronization variables. □

The next three theorems will show that more restrictive mechanisms can simulate less restrictive mechanisms 0-weakly.

THEOREM 5.5. *Let $d, m \in \mathbb{N}$. Then*

$$\mathfrak{PVM}(d, m) \leqq \mathfrak{PV}(d, m).$$

Proof. Let P be an arbitrary element of $\mathfrak{PVM}(d, m)$. Consequently, any synchronization operation $a \in A_{W,R}$ has one of the following forms

$$a\colon V(y_t, \cdots, y_r), \mathbf{goto}\ \tilde{a}; \quad a\colon P(y_t, \cdots, y_r), \mathbf{goto}\ \tilde{a}; \quad \tilde{a} \in A \cup \{\omega\}.$$

We will denote the equivalence classes implied by pnp_P^l by $\mathrm{PNP}_1, \cdots, \mathrm{PNP}_l$. Constructing the system $Q \in \mathfrak{PV}(d, m)$ can now be accomplished by the use of the sets $\mathrm{PNP}_1, \cdots, \mathrm{PNP}_l$. First, we consider the case $\#\,\mathrm{IO}\,(\mathrm{PNP}_i - A_N) = 1$. In this case every action of $\mathrm{PNP}_i - A_N$ is already a "normal" PV-action. All actions of PNP_i will not be changed by the transformation to Q.

Next, we treat $\#\,\mathrm{IO}\,(\mathrm{PNP}_i - A_N) > 1$. Therefore we are able to choose two variables v_{i1}, v_{i2} from $\mathrm{IO}\,(\mathrm{PNP}_i - A_N)$. These two variables will do all synchronization in Q, which corresponds to the synchronization expressed by PNP_i in P. The variables v_{i1}, v_{i2} will only be binary semaphores in Q, i.e., $d = 1$ is included. For every variable y_j of $\mathrm{IO}\,(\mathrm{PNP}_i - A_N)$ we use a nonsynchronization variable $\bar{x}_j$ in Q, $\bar{x}_j$ memorizes the value of the original y_j, and an additional waiter count $wait_i$ for all actions $\mathrm{PNP}_i - A_N$. Now, Q is constructed by substituting every action $a\colon V(y_t, \cdots, y_r)$, **goto** $\tilde{a} \in \mathrm{PNP}_i$ by

(*) $b_1\colon P(v_{i1})$;
$b_2\colon \bar{x}_t := \bar{x}_t + 1 \cdots \bar{x}_r := \bar{x}_r + 1$, **goto** *if* $wait_i = 0$ **then** b_3 **else** b_4;
$b_3\colon V(v_{i1})$, **goto** $\tilde{a}$;
$b_4\colon V(v_{i2})$, **goto** $\tilde{a}$;

and every action $a'\colon P(y_t, \cdots, y_r)$, **goto** $\tilde{a} \in \mathrm{PNP}_i$ by

(**) $b'_1\colon P(v_{i1})$;
b'_2: **goto** *if* $\bar{x}_t > 0 \wedge \cdots \wedge \bar{x}_r > 0$ **then** b'_{10} **else** b'_3;
$b'_3\colon wait_i := wait_i + 1$;
$b'_4\colon V(v_{i1})$;
$b'_5\colon P(v_{i2})$;
$b'_6\colon wait_i := wait_i - 1$;
b'_7: **goto if** $wait_i = 0$ **then** b'_8 **else** b'_9;
$b'_8\colon V(v_{i1})$, **goto** b'_1;
$b'_9\colon V(v_{i2})$, **goto** b'_1;
$b'_{10}\colon \bar{x}_t := \bar{x}_t - 1 \cdots \bar{x}_r := \bar{x}_r - 1$;
$b'_{11}\colon V(v_{i1})$, **goto** $\tilde{a}$;

The initialization is $z_{0_Q}(\bar{x}_j) = z_{0_P}(y_j)$, $z_0(v_{i1}) = 1$, $z_0(v_{i2}) = 0$ and $wait_i = 0$ for any

PNP_i. The representation for an action $b \in A_Q$ is

$$\mathrm{rep}(b) = \begin{cases} b & \text{if } b \in A_P, \\ a & \text{if } b = b_3, b_4, \\ a' & \text{if } b = b'_{11}, \\ \varepsilon & \text{otherwise.} \end{cases}$$

This is a somewhat informal definition since the unique association for any a: $V(y_t, \cdots, y_r)$, **goto** $\tilde{a}$; resp. a': $P(y_t, \cdots, y_r)$, **goto** $\tilde{a}$; to their substitutes (*) resp. (**) has been omitted.

We will first prove that Q simulates P. Relating the variables of the two systems we need a function f from a set of actions to $\mathbb{Z}$:

$$f(A) = \begin{cases} 1 & \text{if } b'_{11} \in A, \\ -1 & \text{if } b_4, b_3 \in A, \\ 0 & \text{otherwise.} \end{cases}$$

(1) $\forall \gamma \in \theta(z_{0_Q}), j \in [1{:}\ m], i \in [1{:}\ n]$

$$(z_{0_P} * \mathrm{rep}(\gamma)(y_j) = z_{0_Q} * \gamma(\bar{x}_j) + f(\Gamma(z_{0_Q} * \gamma)) \wedge z_{0_P} * \mathrm{rep}(\gamma)(x_i) = z_{0_Q} * \gamma(x_i);$$

The proof of (1) is established by induction on the length of γ. The basis holds by the initialization. For the induction step we consider all possible actions $a \in A_Q$. Since the transformation of P does not affect actions $a \in A_{N_P}$ we only have to look at $A_{R_P} \cup A_{W_P}$.

(2) $$\forall \sigma \in \Sigma_Q(\mathrm{rep}(\sigma) \in \Sigma_P).$$

Because the preconditions for the execution of a resp. a' in P correspond to the tests in b_2 resp. b'_2 by (1) the statement (2) is easily proven by induction on the length of the suffix for any computation σ.

(3) $$\forall \sigma \in \Sigma_P(\exists \sigma' \in \Sigma_Q(\mathrm{rep}(\sigma') = \sigma)).$$

We construct σ' replacing every a (resp. a') in σ by the corresponding $b_1 b_2 b_3$ (resp. $b'_1 b'_2 b'_{10} b'_{11}$]. By induction, σ' is proven a transition sequence in z_{0_Q}. It is left to prove that Q simulates P 0-*weakly*:

- rep respects the unity of the processes. Consequently, Definition 3.3(a) holds.
- Discussing the deadlock-freeness of Q and P we do not consider actions a out of $A_{W_P} \cup A_{R_P}$ which have not been transformed, i.e., $\# \mathrm{IO}(\mathrm{PNP}_i - A_N) = 1$, $a \in \mathrm{PNP}_i$.

Assume P is not deadlock-free, i.e., an execution σ with $a \in \Gamma(z_{0_P} * \sigma) - \Psi(z_{0_P} * \sigma)$. For any σ' with $\mathrm{rep}(\sigma') = \sigma$ the corresponding test in b'_2 must have failed by (1) and Q has $b'_5 \in \Gamma(z_{0_Q} * \sigma') - \Psi(z_{0_Q} * \sigma')$.

Assume Q contains an execution σ ending in a deadlock. If a $b'_5 \in \Gamma(z_{0_Q} * \sigma) - \Psi(z_{0_Q} * \sigma) = D$ exists there is an analogous deadlock in Q by (1). Now, assume $b'_1, b_1 \in D$. Because no other action is ready in this state the last execution of an action b_1 must have been followed by an action b_4. b_4 could only happen, if $wait_i > 0$ and another process has b'_5 available. b'_5 leads always to b'_8. This is a contradiction to $b'_1, b_1 \in D$ and Definition 3.3(b) holds.

- A busy-wait can only occur in an infinite loop containing only nonvisible actions in Q. The additionally introduced loop starts with b'_1 and ends with b'_8 or b'_9. Since the repetition of the loop has to be unbounded, an unbounded number of actions b'_5: $P(v_{i2})$ are executed. This is only possible, if an unbounded number

of actions b'_9: $V(v_{i2})$ occurs (the action b_4 cannot occur because this action is visible). This implies that $wait_i$ had an unbounded value. Since $wait_i$ is not greater than the number of the processes (k) such an infinite loop is not possible. The Definition 3.3(c) holds with $e = 11 \cdot k$.

- The equivalence classes implied by pnp^t are the same for both systems apart from the replacement of $b_1 \cdots b_4$ resp. $b'_1 \cdots b'_{11}$ instead of a resp. a'.
- By $\text{rep}(a) \in A_{R_P} \Rightarrow a \in A_{R_Q}$ the last part of Definition 3.3 holds immediately.

Consequently Q simulates P 0-weakly. □

THEOREM 5.6. *Let $d, m \in \mathbb{N}$. Then*

$$\mathfrak{PVG}(d, m) \leqq \mathfrak{PVC}(d, m).$$

After replacing the tests and the expressions in the proof of Theorem 5.5 by the more general forms of the PV-general systems the proof of Theorem 5.6 is complete. By the above two theorems and the trivial statements $\mathfrak{PV}(d, m) \leqq \mathfrak{PVM}(d, m)$ and $\mathfrak{PVC}(d, m) \leqq \mathfrak{PVG}(d, m)$ the undirected arcs of Fig. 1 are valid. Theorem 3.1 allows us to extend the hierarchy as shown in Fig. 1.

THEOREM 5.7. *Let $d, m \in \mathbb{N}$. Then*

$$\forall P \in \mathfrak{PVG}(d, m)$$
$$(\exists Q \in \mathfrak{PV}(1, 2), m' \in [0: m-1](Q \textit{ simulates } P\ m'\textit{-weakly})).$$

Proof. Since Q is an element out of $\mathfrak{PV}(1, 2)$ we may use two binary semaphore variables—say v_1, v_2. For every synchronization variable $y_j (j \in [1: m_P])$, we use a nonsynchronization variable $\bar{x}_j$. *wait* is the additional waiter count. Q is constructed by substituting every action $a \in A_{R_P}$ resp. $a \in A_{W_P}$ by (*) resp. (**) as in the proof of Theorem 5.5. While the original transformation (see the proof of Theorem 5.5) used separate synchronization variables v_{i1}, v_{i2} for every equivalence class, this transformation uses only two variables v_1, v_2 for the union of all equivalence classes. All the results of the proof of Theorem 5.5 are consequently valid with the exception of equal number of equivalence classes implied by pnp_P^t (resp. pnp_Q^t). Since m different equivalence classes could have collapsed to one the simulation is at least a $(m-1)$-weak simulation. □

This last theorem proves that two binary semaphores are capable of doing all necessary synchronization but the loss of parallelism may be substantial.

6. Comparison with related work. Most related work is based on Lipton's thesis and notion of "simulation" [17]. Here we call his notion "strong simulation" as the comparison between the results implies it. The following definition is due to Lipton with a slight change in (a).

DEFINITION 6.1. Let $\text{SP} = (V, W, z_0, A)$ simulate $\text{SP}' = (V', W', z'_0, A')$ by rep. Then SP *simulates* SP' by rep *strongly* iff

(a) for every pair of visible actions $a, b \in A$ $\text{pr}(a) = \text{pr}(b) \Leftrightarrow \text{pr}(\text{rep}(a)) = \text{pr}(\text{rep}(b))$;

(b) there exists an $e \in N$ so that for any transition sequence η in z_0 $\lg(\eta) \leqq (\lg(\text{rep}(\eta)) + 1) \cdot e$;

(c) for any transition sequence η in z_0 $\text{pr}(\Psi(z'_0 * \text{rep}(\eta))) \subseteq \text{pr}(\Psi(z_0 * \eta))$.

Condition (c) prohibits the replacement of an action $a' \in A'$ by a sequence of actions which may introduce additional waiting for a process of SP.

Example 3.1b. G simulates H by rep_1, but not strongly, because with $\eta = a_1^1$

$$\text{pr}(\Psi(z_{0_H} * \varepsilon)) = \{2\} \not\subseteq \text{pr}(\Psi(z_{0_G} * a_1^1)) = \{1\}.$$

In Fig. 2 we give the main results of Lipton [17], [18], Lipton, Snyder and Zalcstein [19], where a directed arc from $\mathfrak{P}$ to $\mathfrak{Q}$ denotes the existence of a system $P \in \mathfrak{P}$ such that there is no system in $\mathfrak{Q}$ which simulates P strongly.

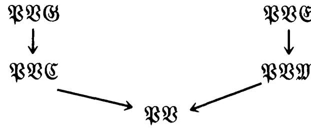


FIG. 2.[4]

At a first glance one might suspect that for deadlock-free systems strong simulation implies 0-weak simulation. In general, this is not true because Condition (c) of Definition 6.1 is a "dynamic" property and Condition (d) of Definition 3.3 is a more "static" property. To understand this difference let us suppose SP simulates SP′ with two equivalence classes implied by pnp^t_{SP} (resp. $\text{pnp}^t_{\text{SP}'}$). Now we add an action $V(y_j)$ (resp. $P(y_j)$) to any process of SP (resp. SP^1) with y_j a new synchronizer variable and $z(y_j) > 1$ for any reachable state z. This synchronization is useless but the simulation is strong and 1-weak. For "useful-synchronized" systems strong simulation implies 0-weak simulation.

Recently, Stark [25] worked on the relative power of semaphore primitives. The notion of relative power is given by posing and answering questions of the form: Under certain natural constraints, is it possible to implement starvation-free mutual exclusion with a given kind of semaphore? Stark proved that weak general semaphores are more powerful than weak binary semaphores for a certain subclass of solutions to the starvation-free mutual exclusion problem. Since our definition of semaphores corresponds to weak semaphores (see note after Table 1), this result seems to contradict Theorem 5.7. The reason for this diverging result is Stark's finite-delay property: If an action remains continuously ready from a certain state on, the action has to be executed eventually. This property corresponds to the definition of just [16]. With this property in mind let us look at our simulation of the action a: $P(y_t, \cdots, y_r)$ in the proof of Theorem 5.7. If we assume a to be continuously ready from a certain state on, this action has to be executed once. Replacing a by the sequence of actions as given in the proof of Theorem 5.7, we cannot ensure that any action of the sequence—especially b'_1—is always ready in the corresponding state sequence. Consequently, b'_1 and b'_{11} are not forced to happen after some finite delay. Interestingly, our simulation fulfills the stronger fair property [16]: If an action is infinitely often ready, the action is infinitely often executed.

7. Conclusion. This paper introduced a refined method to analyze the power of sets of systems of processes. A comparison with already known methods showed that our approach produces more realistic results.

We have dealt only with semaphore-mechanisms in this paper. However, it is possible to compare other synchronization mechanisms using the same ideas in appropriate models.

Looking at the results of this paper it is evident that even using very few synchronizer variables, every synchronization problem can be implemented. However, our results show the difference in quality—i.e. parallelism. As a consequence any

[4] The rightmost arc in Fig. 2 is not part of the literature, but is easily added using [18, Thm. 7, 8] and P with $A_P = \{a^1_1$: **when** $\beta^1 = l_1 \wedge y_1 = 0$ **do** $\beta^1 := \omega$, a^2_2: $V(y_1)$, **goto** $\omega\}$, $z_0(y_1) = 0$.

solution for a synchronization problem should be checked for any restriction of parallelism. Any restriction of parallelism introduced by a compiler for a language incorporating parallelism is even more serious. Such a restriction is difficult to realize for a user since a single run of a nondeterministic program is not a measure for the quality. Consequently, any compiler should give information about the real parallelism of the compiled code.

Usually the number of variables and the values they may assume determines to some extent the complexity of programs. Our results illustrate that simple, elegant solutions may imply a reduced parallelism.

Acknowledgment. Thanks are due to the referees for their suggestions which improved the readability of this paper.

Appendix (list of symbols).

a — action
A — set of actions
a^i — action of process i
A^i — set of actions of process i
$\bar{a}$ — function of a
A_N — set of nonsynchronizer actions
A_R — set of releaser actions
A_W — set of waiter actions
$\mathrm{adr}\,(a)$ — label of action a
B — domain of β (labels)
β^i — program-counter of process i
$\mathfrak{b}$ — ordered set of program-counters
e_a — predicate for an action a
ε — empty sequence
$\Gamma(z)$ — available set in state z
$\Gamma_{N,R,W}$ — restriction of Γ to A_N resp. A_R resp. A_W
$\mathrm{IN}\,(a)$ — input variable of a
$\lg\,(\eta)$ — length of η
m_{SP} — number of synchronizers in SP
$M(\eta)$ — elements of the sequence η
$\mathrm{MP}_{\mathrm{SP}}$ — number of equivalence classes implied by $\mathrm{pnp}^t_{\mathrm{SP}}$
n_{SP} — number of nonsynchronizers in SP
$\mathrm{OUT}\,(a)$ — output variable of a
Ω_{SP} — set of reachable states of SP
ω — stop-symbol for a program-counter
pnp — potentially not parallel relation
$\mathrm{pr}\,(a)$ — process of a
$\Psi(z)$ — set of ready actions in state z
$\Psi_{N,R,W}$ — restriction of Ψ to A_N resp. A_R resp. A_W
$\mathfrak{PV}(d, m)$ — set of PV-systems with m variables out of a domain $[0\colon d]$
rep — representation of a simulation
SP — system of processes
Σ_{SP} — set of executions of SP
$\theta(z)$ — set of transition sequences in z
V — variables of a system
W — domain of V

x	nonsynchronizer variable
$\mathfrak{x}$	ordered set of nonsynchronizer variables
y	synchronizer variable
$\mathfrak{y}$	ordered set of synchronizer variables
z	a state
z_0	an initial state
Z	set of states
$\# N$	cardinality of the set N
$\eta \Vert N$	restriction of η to N
$[m: n]$	the set of $m, m+1, \cdots, n$
$\eta[m: n]$	subsequence of η from the mth element to the nth element

REFERENCES

[1] T. AGERWALA, *Some extended semaphore primitives*, Acta Informatica, 8 (1977), pp. 201–220.

[2] S. ANDLER, *Synchronization primitives and the verification of concurrent programs*, Ph.D. Thesis, Dept. Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, 1977.

[3] H. W. BARZ, *Vergleich der Mächtigkeit von Synchronisationsmechanismen*, Institut für Informatik, Univ. Bonn, Bonn, West Germany, 1981.

[4] G. BELPAIRE AND J. P. WILMOTTE, *A semantic approach to the theory of parallel processes*, International Computing Symposium 1973, A. Guenther, ed., North-Holland, Amsterdam, 1974, pp. 159–164.

[5] T. BLOOM, *Synchronization mechanisms for modular programming languages*, Massachusetts Institute of Technology, Cambridge, MA, MIT/LCS/TR-211, 1979.

[6] P. BRINCH HANSEN, *Concurrent programming concepts*, Computing Surveys, 4 (1973), pp. 223–245.

[7] R. H. CAMPBELL AND A. N. HABERMANN, *The specification of process synchronization by path expressions*, Operating Systems, Lecture Notes in Computer Science 16, E. Gelenbe and C. Kaiser, eds., Springer, Berlin, 1974, pp. 89–102.

[8] V. G. CERF, *Multiprocessors, semaphores and a graph model of computation*, Ph.D. Thesis, Computer Science Dept., Univ. California, Los Angeles, CA, 1972.

[9] E. W. DIJKSTRA, *Co-operating sequential processes*, Programming Languages, F. Genuys, ed., Academic Press, New York, pp. 43–112.

[10] ———, *A tutorial on the split binary semaphore*, Burroughs Corporation, Nuenen, Netherlands, Rept. EWD 703, 1979.

[11] ———, *A strong* P/V-*implementation of conditional critical regions*, Burroughs Corporation, Nuenen, Netherlands, Rept. EWD 651, 1978.

[12] W. EVENTOFF, D. HARVEY AND R. J. PRICE, *The rendezvous and monitor concepts: is there an efficiency difference?*, Proc. ACM SIGPLAN Symposium on the ADA Programming Language, ACM SIGPLAN Notices, 15 (1980), pp. 156–165.

[13] J. A. FELDMAN, *High level programming for distributed programming*, Comm. ACM, 6 (1979), pp. 353–368.

[14] R. M. KARP AND R. E. MILLER, *Parallel program schemata*, J. Comput. System Sci., 3 (1969), pp. 147–195.

[15] W. KOWALK AND R. VALK, *On reductions of parallel programs*, Automata, Languages and Programming, Lecture Notes in Computer Science 71, H. A. Maurer, ed., Springer, Berlin, 1979, pp. 356–369.

[16] D. LEHMANN, A. PNUELI AND J. STAVI, *Impartiality, justice and fairness: the ethics of concurrent termination*, Automata, Languages and Programming, Lecture Notes in Computer Science 115, S. Even and O. Kariv, eds., Springer, Berlin, 1981, pp. 264–277.

[17] R. J. LIPTON, *On synchronization primitive systems*, Computer Science Dept., Yale Univ., New Haven, CN, Tech. Rept. 22, 1973.

[18] ———, *Limitations of synchronization primitives with conditional branching and global variables*, Proc. Sixth ACM Symposium on Theory of Computing Machinery, New York, 1974, pp. 230–241.

[19] R. J. LIPTON, L. SNYDER AND Y. ZALCSTEIN, *A comparative study of models of parallel computations*, Conference Record of the Fifteenth Annual IEEE Symposium on Switching and Automata Theory, IEEE, New York, 1974, pp. 145–155.

[20] R. J. LIPTON, L. SNYDER AND Y. ZALCSTEIN, *Evaluation criteria for process synchronization*, Proc. 1975 Sagamore Conference on Parallel Computing, IEEE, New York, 1975, pp. 245–250.

[21] R. J. LIPTON, *Reduction: a method of proving properties of parallel programs*, Comm. ACM, 12 (1975), pp. 717–721.

[22] J. R. MCGRAW, *Language features for process interaction and process control*, Dept. Computer Science, Cornell University, Ithaca, NY, TR-77-319, 1977.

[23] R. E. MILLER AND C. K. YAP, *Formal specification and analysis of loosely connected processes*, IBM T. J. Watson Research Center, Yorktown Heights, NY, RC6716, 1979.

[24] D. L. PARNAS, *On a solution to the cigarette smokers problem (without conditional statements)*, Dept. Computer Science, Carnegie Mellon Univ., Pittsburgh, PA, 1972.

[25] E. W. STARK, *Semaphore primitives and starvation-free mutual exclusion*, J. ACM, 4 (1982), pp. 1049–1072.

[26] H. VANTILBORGH AND A. VAN LAMSWEERDE, *On an extension of Dijkstra's semaphore primitives*, Inform. Process. Lett., 1 (1972), pp. 181–186.

[27] H. ZIMA, *Betriebssysteme (Parallele Prozesse)*, Bibliographisches Institut, Zürich, Switzerland, 1976.

SIAM J. COMPUT.
Vol. 13, No. 4, November 1984
© 1984 Society for Industrial and Applied Mathematics
007

n-RATIONAL ALGEBRAS I. BASIC PROPERTIES AND FREE ALGEBRAS*

JEAN H. GALLIER†

Abstract. A (strict) hierarchy of algebras in which only certain "constructive" chains have a least upper bound is studied. Such algebras, called *n-rational*, are "ideal" interpretations for finite recursion schemes of higher types. Indeed, the constructive chains that have least upper bounds in these algebras are obtained by unfolding rational recursion schemes of higher types. Part I of this paper deals with basic properties of these algebras, the existence of free algebras in particular. In Part II [SIAM J. Comput., 13 (1984), pp. 776–794], varieties and a logic of inequalities are studied. A proof system with one infinitary inference rule is shown to be complete.

Key words. algebraic semantics, recursion schemes of higher types, infinite trees, fixed-points, varieties of algebras, logic of inequalities

1. Introduction. The main goal of this paper is to study further the semantics and (inequational) logic of recursion schemes of finite types as defined by Damm [11], [12], [13]. More specifically, this paper is concerned with defining the (strict) hierarchy of what are called n-rational algebras, and with investigating certain of their mathematical properties.

The motivation for studying n-rational algebras comes from computer science. Essentially, the n-rational algebras are those in which one can solve nth level recursion schemes by means of least fixed-points.

In this paper, we follow the approach in which a program is defined as a pair (scheme, interpretation), where an interpretation is an algebra with a certain order structure (so that fixed-point equations induced by programs can be solved). The meaning of a program is taken to be the function (derived operator) induced by an infinite tree associated with the scheme (and obtained by "unfolding" the scheme). Following Nivat [33], we will refer to such a semantics as an *algebraic semantics.* There are two other equivalent definitions of the meaning of a program: one using a *least fixed-point approach,* the other using an *operational approach* in terms of computation sequences. The reader is referred to Guessarian [27] or Nivat [33] for further details. Adopting the "algebraic semantics" has the advantage that semantic questions are reduced to questions about certain infinite trees. Hence, techniques used in studying such trees, such as formal language theory and algebra, can be used to answer semantic questions.

Using program schemes, one can study the properties of a *class* of programs, such as the family of all programs defined by a *class* of interpretations. Interesting properties about programs include extension, equivalence and various forms of correctness. One may also be interested in semantic-preserving program transformations, as in Courcelle [7].

In most applications, a class of interpretations arises as the class of models of a set of first-order axioms. In particular, equational classes of interpretations have been studied by Guessarian [26], [27], Courcelle and Nivat [10], Courcelle and Guessarian [8], Courcelle [7] and Nivat [33].

* Received by the editors June 24, 1981, and in final revised form June 21, 1983. This research was partially supported by the National Science Foundation under grant MCS-8111726.

† Department of Computer and Information Sciences, Moore School of Electrical Engineering D2, University of Pennsylvania, Philadelphia, Pennsylvania 19104.

The main technical tool for studying classes of algebras is the notion of a *Herbrand* (or *universal*) algebra. For example, two schemes are equivalent for all algebras in a class, if and only if they are equivalent in the Herbrand algebra for that class. Unfortunately, for arbitrary classes of algebras (even classes of models of a set of axioms), the structure of the Herbrand algebra is intractable (see Courcelle and Guessarian [8] and Guessarian [27]). It is only reasonably tractable for equational classes (Courcelle and Nivat [10]), and, even in this case, requires a "completion operation" (to imbed an ordered algebra into a continuous algebra).

The problem arises because in this approach, an interpretation is defined as an ω-continuous algebra. This means that carriers are partially ordered sets in which every countable ascending chain has a least upper bound. This approach in which all (countable) chains are considered is directly inspired by the seminal work of Scott [37], [38], [39], [40]. The problem with ω-completeness is that it is not preserved by certain operations, images under homomorphisms and quotients, in particular. It is therefore necessary to use completion constructions in order to regain ω-continuity. Thus, the theory of varieties of ω-continuous algebras is rather complex. A detailed study of such varieties can be found in Meseguer [31].

However, as first noted by Elgot [14] and then by Goguen, Thatcher, Wagner and Wright [2], Tiuryn [44], [45] and Gallier [20], [21], ω-continuity is a "luxury", and weaker notions of completeness requiring that only certain "constructive" chains have a least upper bound work just as well, if not better.

In this paper, this idea is exploited further, by defining a (strict) hierarchy of classes n-R-ALG_Σ of algebras such that for each $n \geqq 0$, algebras in the nth class contain least upper bounds of certain constructive chains. These chains are obtained by "unfolding" certain "rational" recursion schemes of level n (see Damm [11], [12], [13], Engelfriedt and Schmidt [16]). Algebras in the nth class are called *n-rational.* The n-rational algebras are well-behaved with respect to images under homomorphisms and quotients and, in particular, to form quotient algebras one just takes the algebra of equivalence classes. These algebras are ordered (Bloom [4]), ending for $n = 0$ with the ordered *regular algebras* of Tiuryn [44], [45], [46], [47], [48], and forming a (strict) chain of inclusions

$$C\text{-ALG}_\Sigma \cdots \subseteq (n+1)\text{-}R\text{-ALG}_\Sigma \subseteq n\text{-}R\text{-ALG}_\Sigma \subseteq \cdots \subseteq 1\text{-}R\text{-ALG}_\Sigma \subseteq 0\text{-}R\text{-ALG}_\Sigma,$$

where C-ALG_Σ denotes the class of ω-continuous algebras. Using results from formal language theory (Courcelle [7], Damm [11], [12], [13], Gallier [19]), it can be shown that the hierarchy is strict for $n = 0, 1, 2$. Using recent results of Damm [13], based on complexity arguments (the notion of rational index), it is shown here that the *entire hierarchy* is strict (see Theorem 3.11).

The technical device used for defining recursion schemes of higher types is the notion of a *derived alphabet* which was first used by Maibaum [30]. The operators used in derived alphabets are similar to typed combinators (Hindley, Lercher and Seldin [28]). Indeed, there are composition, abstraction and projection operators.

In the first part of this paper, basic properties of n-rational algebras are established, such as the existence of free algebras and the characterization of subalgebras generated by a subset. Free n-rational algebras have a particularly nice structure: they consist of infinite trees obtained by first "unfolding" a rational recursion scheme of "level n", and then applying a mapping (usually called yield^n or beta^n) in order to perform certain substitutions (Engelfriedt and Schmidt [16], Damm [11], [12], [13]).

In Part II (this issue, pp. 776–794), varieties of n-rational algebras satisfying a set of inequalities and the corresponding logic are studied. The varieties under consideration are classes of n-rational algebras satisfying sets of inequalities of the form $t_1 \leqq t_2$, where t_1 and t_2 are (possibly infinite) n-rational trees in the free n-rational algebra $n\text{-}RT_\Sigma(X)$ over the countable set of variables X. We first show that there is a bijective Galois connection between classes of inequational varieties and certain n-rational (pre-)congruences, generalizing "fully invariant" congruences (Cohn [5]). Then, we characterize the least n-rational precongruence induced by a set of (infinite) inequalities, using a proof system with one infinitary inference rule (the "lub" rule). We then prove the completeness of this (infinitary) proof system. If we consider sets of inequalities between finite trees, we obtain a simpler infinitary inference rule, which looks very much like an induction rule. In that case, we can also prove that the preorder induced by the inequalities is "algebraic" (or inductive) in the sense of Courcelle and Nivat [10]. The Herbrand algebra of an inequational variety is obtained as a quotient of a free n-rational algebra, and does not require a completion operation.

The logic that we are investigating is a logic in which certain constructive infinite terms (really trees) are allowed. We have only considered a simple fragment, namely the logic of inequalities. Unfortunately, even for $n=1$, the set of valid inequalities is not even recursively enumerable. This has been shown by Courcelle [7] and follows from the undecidability of the inclusion problem for simple languages proved by Friedman [18]. Hence there is no hope for a recursive axiomatization. However, we have found a complete axiomatization involving one infinitary inference rule. We leave to interested logicians the study of the ordinals measuring the size of the (countable) well founded proof trees arising in such proofs. There might well be connections with studies of recursive functionals of finite types arising in proof theory (Schutte [35], Feferman [17], Schwichtenberg [36]).

The full paper being rather long, it has been divided into two parts.

We now outline the contents of Part I. In § 2, we review a number of concepts, derived alphabets and rational schemes of level n in particular. The classes of n-rational algebras are defined in § 3. The existence of free algebras is shown, and it is proved that the hierarchy $(n\text{-}R\text{-ALG}_\Sigma)_{n\geqq 0}$ is strict. In § 4, we give a characterization of the least n-rational subalgebra generated by a subset. We also characterize morphisms of n-rational algebras, in the manner of Tiuryn [44], [45]. Section 5 is devoted to a study of n-rational derived operators. We show that they form a $(n-1)$-rational derived algebra. This has the interesting consequence that the nth derived algebra is 0-rational. In other words, rational schemes over (derived) level n functionals have least fixed-points. In § 6, we mention problems which to the author's knowledge are still open.

Many ideas in algebraic semantics, parameterized schemes in particular, are due to Wagner [49], [50], [51]. The papers by Engelfriedt and Schmidt [16], Damm [12], [13] and especially Tiuryn [44], [45], [46], [47] were also sources of inspiration for this work. We have chosen to use standard universal algebra as presented in Cohn [5] or Gratzer [25] instead of a more categorical approach as in Meseguer [31] or Lehmann [29]. A categorical treatment using algebraic theories is probably possible. We leave this as a topic for further research, hoping that our research will contribute to such a treatment.

2. Preliminaries: infinite trees, algebras, derived alphabets. First, we review some definitions, especially those of derived alphabets and derived algebras, which are basic tools in our study of hierarchies of rational algebras.

DEFINITION 2.1. *Sorts* (*or types*). By a set of *sorts* we understand any nonempty set S. Sorts are usually intended to be primitive data types.

DEFINITION 2.2. *S-ranked alphabet.* An *S-ranked alphabet or signature* Σ is a family $(\Sigma_{(u,s)})_{(u,s)\in S^*\times S}$ of disjoint sets $\Sigma_{u,s}$ indexed by the pairs (u, s) in $S^*\times S$ (S^* denotes the set of all strings over S, the empty string being denoted by e). Intuitively, if $u = u_1 \cdots u_n$, each symbol f in $\Sigma_{u,s}$ represents an operation taking n arguments, the ith argument of sort u_i, and yielding an element of sort s. We say that f has *type* (u, s), *arity* u and *sort* s. Symbols of arity e are called *constants.* In the rest of this paper, we will assume that a special symbol (the bottom symbol) denoted $\perp_s$ is adjoined to every alphabet $\Sigma_{e,s}$. For readability, we drop subscripts whenever possible.

DEFINITION 2.3. *Tree domain.* A *tree domain* D is a nonempty subset of strings over N_+ (the set of positive integers) satisfying the following conditions:

(1) For all u, v in N_+^*, if uv is in D, then u is also in D.

(2) For all u in N_+^*, for all i in N_+, if ui is in D, then, for every j, $1 \leqq j \leqq i$, uj is also in D.

DEFINITION 2.4. *Σ-tree.* Given an S-ranked alphabet Σ, a *Σ-tree* (for short, a tree) is a function $t : D \to \Sigma$ such that the following conditions hold:

(1) D is a tree domain.

(2) For all w in D, let $n = \text{card}(\{wi \mid wi \in D\})$:

  (i) if $n = 0$ then $t(w)$ belongs to some $\Sigma_{e,s}$;

  (ii) if $n > 0$ and if each $t(wi)$ is of sort u_i, then $t(w)$ is in $\Sigma_{u,s}$ for some sort s, where $u = u_1 \cdots u_n$.

D is called the *domain* of the tree t and is denoted by dom (t). The elements of the domain are called *nodes* or *tree addresses.* A node satisfying condition 2(i) is called a *leaf.* The node corresponding to the empty string is the *root* of the tree. The sort of the symbol labeling the root of the tree is also called the sort of the tree. The definition of a tree using the notion of a tree domain goes back to Gorn [24] (see also Rosen [34]).

The set of all trees of sort s is denoted by CT_Σ^s and the set of all trees by CT_Σ. A tree is *total* if no tree address is labeled with one of the symbols $\perp_s$, and otherwise we say that the tree is a *partial* tree. A tree is finite if its domain is finite. The set of total finite trees is denoted by T_Σ and the set of partial and total finite trees is denoted by FT_Σ.

Remark. In order to avoid pathological cases, it will be assumed from now on that for every ranked alphabet Σ, for every sort s, FT_Σ^s contains some tree distinct from $\perp_s$. We say that such ranked alphabets are *nontrivial.*

Each set CT_Σ^s (and FT_Σ^s) is partially ordered. Intuitively speaking, $t_1 \leqq t_2$ if t_1 is obtained from t_2 by replacing any number of subtrees of t_2 with the symbol $\perp$. Formally, for every pair of trees t_1, t_2, the relation $t_1 \leqq t_2$ holds if and only if

(1) $\text{dom}(t_1) \subseteq \text{dom}(t_2)$; and

(2) for all w in $\text{dom}(t_1)$, $t_1(w) \neq \perp$ implies that $t_2(w) = t_1(w)$.

The tree $\perp_s$ is the least element of CT_Σ^s.

DEFINITION 2.5. *Tree composition.* The *composition* (or *substitution*) of trees is defined as follows. For each u in S^*, let $X_u = \{x_1^u, \cdots, x_n^u\}$ be a set of variables, with each x_i^u of sort u_i ($n = |u|$ is the length of the string u, and $X_e = \emptyset$). Let $X^s = \{x_i^u \mid u_i = s, u \in S^*\}$ and let $X = \bigcup X^s$. Adjoining each set of variables X^s to the constants in $\Sigma_{e,s}$, we obtain the set $CT_\Sigma(X)$ of trees with variables in X (with $CT_\Sigma^s(X) = CT_{\Sigma\cup X}^s$). Similarly, viewing X_u as an S-indexed family, we obtain the set of trees with variables in X_u, $CT_\Sigma(X_u)$. The variables are used as markers to indicate where the substitution takes place. Given a tree t' in $CT_\Sigma(X_v)$ with $|v| = n$, given an n-tuple $(t_1, \cdots, t_n)$ of trees in $CT_\Sigma(X_u)$, with each t_i of sort v_i for $i = 1, \cdots, n$, the *composition* of $(t_1, \cdots, t_n)$

and t' is denoted by $(t_1, \cdots, t_n) \circ t'$ and is defined by the function whose graph is the set of pairs

$$\{(wz, t_i(z)) \mid w \in \text{dom}(t'), t'(w) = x_i^v, z \in \text{dom}(t_i) \text{ and } t_i \text{ is a tree of sort } v_i, 1 \leqq i \leqq n\}$$
$$\cup \{(w, t'(w)) \mid w \in \text{dom}(t') \text{ and } t'(w) \notin X_v\}.$$

In the special case where the tree t' is given by the graph $\{(e, f), (1, x_1^v), \cdots, (n, x_n^v)\}$ (where f has arity v and $n = |v|$), the composition $(t_1, \cdots, t_n) \circ t'$ is also denoted by $f(t_1, \cdots, t_n)$ or even by $ft_1 \cdots t_n$. The tree t' is also denoted by $f(x_1^v, \cdots, x_n^v)$ or even by f. Similarly, the tree whose graph is $\{(e, a)\}$ is denoted by a. Composition is extended to tuples as follows: if $t = (t_1, \cdots, t_m)$ and $t' = (t'_1, \cdots, t'_n)$, where each t_i is a tree of sort v_i in $CT_\Sigma(X_u)$, $1 \leqq i \leqq m$, and each t'_j is a tree in $CT_\Sigma(X_v)$, $1 \leqq j \leqq n$, $|v| = m$, the composition of t and t' is defined as $t \circ t' = (t \circ t'_1, \cdots, t \circ t'_n)$.

Given a finite tree t, the *depth* of the tree t is defined as depth $(t) = \max \{|u| \mid u \in \text{dom}(t)\}$.

DEFINITION 2.6. *Partially ordered sets.* A *partially ordered set*, for short, a *poset* is a pair $(D, \leqq)$ where $\leqq$ is a reflexive, transitive and antisymmetric relation on D. If $\leqq$ is only reflexive and transitive, it is called a *preorder*.

A poset is *ω-complete* if every countable ascending chain has a least upper bound. The least upper bound of a chain C is denoted by $\bigsqcup C$. Note that an ω-complete poset always has a least element denoted by $\perp$ (the bottom symbol).

Let $(D, \leqq)$ and $(D', \leqq')$ be two posets. A function $h: D \to D'$ is *monotonic* if for all a, b in D, $a \leqq b$ implies $h(a) \leqq' h(b)$. A poset having a least element $\perp$ is called a *strict* poset. If D and D' are strict posets, a function $h: D \to D'$ is *strict* if $h(\perp) = \perp'$.

A function h is *ω-continuous* if it preserves least upper bounds of nonempty (countable) chains, that is, for every nonempty (countable) chain C in D, if $\bigsqcup C$ exists then $\bigsqcup h(C)$ exists and $h(\bigsqcup C) = \bigsqcup h(C)$. (Note that this definition does not require D or D' to be chain complete.)

It is well known that each $CT_\Sigma^s(X)$ is an ω-complete poset [3]. The following result is also fundamental.

PROPOSITION 2.7. *For every countable chain $\{t_i \mid i \in N\}$ of finite trees, for any finite tree t, if $t \leqq \bigsqcup\{t_i \mid i \in N\}$ then there is some i such that $t \leqq t_i$. The proposition also holds for a countable chain of not necessarily finite trees.*

DEFINITION 2.8. *Ordered algebras.* In this paper, we will be dealing with algebras whose carriers are posets with a least element. We refer the reader to Goguen, Thatcher, Wagner and Wright [1], [2], [3], Thatcher, Wagner and Wright [41], [42], [43] and Engelfriedt and Schmidt [16] for more details. Let $A = (A_s)_{s \in S}$ be an S-indexed family of sets. For any u in S^*, let

$$A^u = A^{u_1} \times \cdots \times A^{u_n} \quad \text{where } u = u_1 \cdots u_n \text{ with } A^e = \{e\}.$$

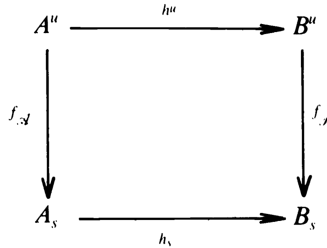
An *ordered Σ-algebra* is a pair $\mathscr{A} = ((A_s)_{s \in S}, \{f_{\mathscr{A}} \mid f \in \Sigma_{u,s}, (u, s) \in S^* \times S\})$ where each *carrier* A_s is a poset with least element $\perp_s$ and each $f_{\mathscr{A}}: A^u \to A_s$ is a monotonic function (called an *operation*). If $u = e$, $f_{\mathscr{A}}$ is a constant, that is, an element of A_s.

An *ω-continuous Σ-algebra* is an ordered Σ-algebra such that each carrier is ω-complete and the operations are ω-continuous.

Given two S-indexed families A and B, an S-indexed family $h = (h_s)_{s \in S}$ of functions $h_s: A_s \to B_s$, and a string $u = u_1 \cdots u_n$ in S^*, we also denote by h^u the function $h^u: A^u \to B^u$ such that

$$h^u(a_1, \cdots, a_n) = (h_{u_1}(a_1), \cdots, h_{u_n}(a_n)).$$

DEFINITION 2.9. A Σ-*homomorphism* (for short a *morphism*) of ordered Σ-algebras $h:\mathcal{A}\to\mathcal{B}$ is an S-indexed family of strict monotonic functions $h:A\to B$ such that for every function symbol f of type (u, s), $f_{\mathcal{A}}\cdot h_s = h^u\cdot f_{\mathcal{B}}$, (we are denoting the composition of two functions $g:A\to B$ and $h:B\to C$ by $g\cdot h$ or gh). Note that for



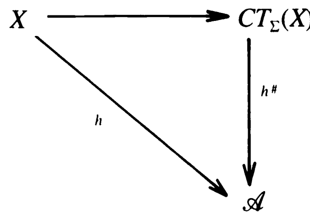
$u = e$, we have $h_s(f_{\mathcal{A}}) = f_{\mathcal{B}}$. A function h which is strict and satisfies the above commutative diagram but is not necessarily monotonic, is called a *weak morphism* of ordered algebras.

DEFINITION 2.10. A Σ-morphism of ω-continuous Σ-algebras is a morphism of ordered Σ-algebras which is also ω-continuous, i.e., each h_s is ω-continuous.

Ordered Σ-algebras and their (monotonic) morphisms form a category P-ALG$_\Sigma$; ordered Σ-algebras and their weak morphisms form a category P^*-ALG$_\Sigma$; ω-continuous Σ-algebras and their morphisms form a category C-ALG$_\Sigma$.

$FT_\Sigma(X)$ can be made into a Σ-algebra by defining the operations as follows: for every f in $\Sigma_{u,s}$, for every $(t_1,\cdots,t_n)$ in $(FT_\Sigma(X))^u$, $f_{FT_\Sigma(X)}(t_1,\cdots,t_n)=f(t_1,\cdots,t_n)$. Similarly, $CT_\Sigma(X)$ is made into a Σ-algebra by replacing $FT_\Sigma(X)$ by $CT_\Sigma(X)$. The following facts are well known. Details can be found in Goguen, Thatcher, Wagner and Wright [3].

PROPOSITION 2.11. *For any S-indexed family X, $FT_\Sigma(X)$ is the free ordered Σ-algebra over X. Similarly, $CT_\Sigma(X)$ is the free ω-continuous Σ-algebra over X. Recall that this means that for every function $h:X\to A$ from X to the carrier of any ω-continuous Σ-algebra $\mathcal{A}$, there is a unique ω-continuous Σ-morphism $h^*:CT_\Sigma(X)\to\mathcal{A}$ extending h.*



DEFINITION 2.12. Given an ordered Σ-algebra $\mathcal{A}$, for each $u\in S^*$, to each tree t in $FT_\Sigma(X_u)$ there corresponds a monotonic function $t_{\mathcal{A}}:A^u\to A_s$ called a *derived operator* defined as follows: $t_{\mathcal{A}}(a_1,\cdots,a_n)=a^*(t)$ where, for any $(a_1,\cdots,a_n)$ in A^u (with $n=|u|$), $a^*:FT_\Sigma(X_u)\to\mathcal{A}$ is the unique morphism extending the function $a:X_u\to A$, such that $a(x_i^u)=a_i$ for $1\leqq i\leqq n$.

A continuous derived operator is defined in the same way for any tree in $CT_\Sigma(X_u)$ and any ω-continuous Σ-algebra $\mathcal{A}$.

Remark. Since a^* is a homomorphism, the following holds: For every tree t of the form $ft_1\cdots t_m$,

$$t_{\mathcal{A}}(a_1,\cdots,a_n)=f_{\mathcal{A}}((t_1)_{\mathcal{A}}(a_1,\cdots,a_n),\cdots,(t_m)_{\mathcal{A}}(a_1,\cdots,a_n)).$$

If $t=(t_1, \cdots, t_n)$ is a v-tuple of trees in $CT_\Sigma(X_u)$ with each t_i of sort v_i ($|v|=n$), t defines a function $t_{\mathscr{A}}: A^u \to A^v$. Proposition 2.13 stated below shows that the mapping which associates a derived operator with a tree (or a tuple of trees) is a homomorphism. We refer the reader to Goguen, Thatcher, Wagner and Wright [3] for details.

In order to reduce the number of subscripts and superscripts, we will denote $(CT_\Sigma(X_u))^v$ by $CT_\Sigma(u, v)$ and similarly for FT_Σ.

PROPOSITION 2.13. *Given any two tuples of trees t in $CT_\Sigma(u, v)$ and t' in $CT_\Sigma(v, w)$, for any ω-continuous Σ-algebra $\mathscr{A}$, $(t \circ t')_{\mathscr{A}} = t_{\mathscr{A}} \cdot t'_{\mathscr{A}}$.*

The above proposition suggests that derived operators form an algebra. This is indeed the case, but it is an algebra not sorted by S but instead by $S^* \times S$. It is also an algebra on a ranked alphabet containing other function symbols besides those in Σ, in particular, function symbols for composition. These ideas can be formulated rigorously as follows. We refer the reader to Damm [11], [12], [13], Engelfriedt and Schmidt [16] and Courcelle and Franchi-Zannettacci [9] for further details.

DEFINITION 2.14. Given a set of sorts S, the *derived set of sorts $D(S)$* is defined as $D(S) = S^* \times S$. Given an S-ranked alphabet Σ, the *derived alphabet $D(\Sigma)$* is defined as follows:

$$D(\Sigma) = \Sigma \cup \begin{cases} C_{u,v,s}, & \text{where } u \in S^*,\ v \in S^+,\ s \in S, \\ \text{abst}_{u,s}, & \text{where } u \in S^*,\ s \in S, \\ \pi_i^u, & \text{where } u \in S^+. \end{cases}$$

$D(\Sigma)$ is sorted by $D(S)$. Each $C_{u,v,s}$ is a composition operator of type $((u, v_1) \cdots (u, v_n)(v, s), (u, s))$, ($|v|=n$), each $\text{abst}_{u,s}$ is an abstraction operator of type $((e, s), (u, s))$, each π_i^u is a projection operator of type $(e, (u, u_i))$ and each f in $\Sigma_{u,s}$ is an operator of type $(e, (u, s))$.

Note that symbols in Σ are now treated as constants, as are the projections. We mention the similarity between the above operators and the combinators of typed combinatory logic, as presented in Hindley, Lercher and Seldin [28].

Given a set of sorts S and an S-ranked alphabet Σ, we define inductively $D^n(S)$ and $D^n(\Sigma)$ as follows:

$$D^0(S) = S, \quad D^{n+1}(S) = D(D^n(S)), \quad D^0(\Sigma) = \Sigma \quad \text{and} \quad D^{n+1}(\Sigma) = D(D^n(\Sigma)).$$

For simplicity, we often omit parentheses and denote $D^n(S)$ by D^nS and $D^n(\Sigma)$ by $D^n\Sigma$.

DEFINITION 2.15. Given an ordered Σ-algebra $\mathscr{A}$, the *$D\Sigma$-algebra of monotonic functions* over $\mathscr{A}$ denoted as $\mathscr{F}\mathscr{A}$, is defined as follows: The carrier $\mathscr{F}\mathscr{A}_{(u,s)}$ of sort (u, s) is the set of all monotonic functions $h: A^u \to A_s$; each f in $\Sigma_{u,s}$ is interpreted as the function $f_{\mathscr{A}}$, each π_i^u is interpreted as the projection from A^u to A_{u_i}, each $C_{u,v,s}$ is interpreted as the composition operation taking n functions $f_i: A^u \to A_{v_i}$ and a function $g: A^v \to A_s$ (with $n = |v|$) and yielding the function $(f_1, \cdots, f_n) \cdot g: A^u \to A_s$; each $\text{abst}_{u,s}$ is interpreted as the "abstraction operation" taking a constant in A_s into the corresponding constant function of u arguments.

DEFINITION 2.16. Similarly, given an ω-continuous Σ-algebra $\mathscr{A}$, we can define the $D\Sigma$-algebra of ω-continuous functions over $\mathscr{A}$. This algebra is denoted by $\mathscr{C}\mathscr{A}$, and its carrier of sort (u, s) is the set $[A^u \to A_s]$ of all ω-continuous functions from A^u to A_s with the pointwise ordering.

It is well known that each poset $[A^u \to A_s]$ is ω-complete, and it is easily verified that $\mathscr{C}\mathscr{A}$ is an ω-continuous algebra (see Goguen, Thatcher, Wagner and Wright [3] or Engelfriedt and Schmidt [16]).

DEFINITION 2.17. Trees with variables can also be made into a $D\Sigma$-algebra as follows. The *tree-substitution algebra* denoted by DFT_Σ is the ordered $D\Sigma$-algebra whose carrier of sort (u, s) is the set of trees $FT_\Sigma(u, s)$; Each f in $\Sigma_{u,s}$ is interpreted as the tree $f(x_1^u, \cdots, x_n^u)$ (where $n = |u|$); each π_i^u is interpreted as x_i^u; each $\mathrm{abst}_{u,s}$ is interpreted as the inclusion function from $FT_\Sigma(e, s)$ to $FT_\Sigma(u, s)$; Each $C_{u,v,s}$ is interpreted as a composition of trees such that, $C_{u,v,s}(t_1, \cdots, t_n, t) = (t_1, \cdots, t_n) \circ t$, for $(t_1, \cdots, t_n)$ in $FT_\Sigma(u, v)$ and t in $FT_\Sigma(v, s)$.

A similar definition can be given by replacing FT_Σ by CT_Σ, thus obtaining the continuous tree-substitution algebra DCT_Σ.

The mapping which assigns a derived operator to a tree is in fact a morphism of ordered algebras (the unique morphism mapping x_i^u to π_i^u)

$$\mathrm{derop}_{\mathcal{A}} : DFT_\Sigma \to \mathcal{F}\mathcal{A}$$

such that for any tree t in $FT_\Sigma(u, s)$, $\mathrm{derop}_{\mathcal{A}}(t) = t_{\mathcal{A}}$. We frequently omit subscripts when the algebra $\mathcal{A}$ is understood.

DEFINITION 2.18. The subalgebra derop (DFT_Σ) of $\mathcal{F}\mathcal{A}$ is an ordered algebra, and it is precisely the algebra of all monotonic derived operators. This algebra will be denoted as der-$\mathcal{A}$.

DEFINITION 2.19. Similarly, if $\mathcal{A}$ is an ω-continuous Σ-algebra, there is a continuous morphism

$$\mathrm{derop}_{\mathcal{A}} : DCT_\Sigma \to \mathcal{C}\mathcal{A}.$$

However, in this case, the ordered algebra derop (DCT_Σ) of derived operators is not necessarily chain-complete.

DEFINITION 2.20. Since $FT_{D\Sigma}$ is the initial ordered $D\Sigma$-algebra, there is a unique monotonic morphism

$$\mathrm{beta} : FT_{D\Sigma} \to DFT_\Sigma.$$

This morphism is of fundamental importance for our development. It behaves very much like beta-conversion in the lambda-calculus, whence the name (borrowed from Courcelle and Franchi-Zannettacci [9]). This morphism can be described explicitly by (primitive) recursion as follows:

$$\mathrm{beta}_{u,s}(f) = f(x_1^u, \cdots, x_n^u) \quad \text{if } f \text{ is in } \Sigma_{u,s}$$

$$\mathrm{beta}_{u,s}(\pi_i^u) = x_i^u \quad \text{if } u_i = s,$$

$$\mathrm{beta}_{u,s}(\mathrm{abst}_{u,s}(t)) = \mathrm{beta}_{e,s}(t) \quad \text{if } t \text{ is in } FT_{D\Sigma}(e, (e, s))$$

$$\mathrm{beta}_{u,s}(C_{u,v,s}(t_1, \cdots, t_n, t)) = (\mathrm{beta}_{u,v_1}(t_1), \cdots, \mathrm{beta}_{u,v_n}(t_n)) \circ \mathrm{beta}_{v,s}(t).$$

The morphism beta has a left inverse which "lifts" a tree into its "functional form," in which composition operators appear explicitly. The family of functions $\mathrm{lift} = (\mathrm{lift}_{u,s})_{(u,s) \in DS}$ is defined recursively as follows:

$$\mathrm{lift}_{u,s}(x_i^u) = \pi_i^u \quad \text{if } u_i = s,$$

$$\mathrm{lift}_{u,s}(f) = \mathrm{abst}_{u,s}(f) \quad \text{if } f \text{ is in } \Sigma_{e,s},$$

$$\mathrm{lift}_{u,s}(f(t_1, \cdots, t_n)) = C_{u,v,s}(\mathrm{lift}_{u,v_1}(t_1), \cdots, \mathrm{lift}_{u,v_n}(t_n), f)$$
$$\text{if } f \text{ is in } \Sigma_{v,s} \text{ and } t_i \text{ in } FT_\Sigma(u, v_i), (n = |v|).$$

Note that lift $\cdot$ beta = id, but beta is not injective in general. A similar treatment applies to DCT_Σ.

Example 2.1. A tree in "functional form" is shown in Fig. 1.

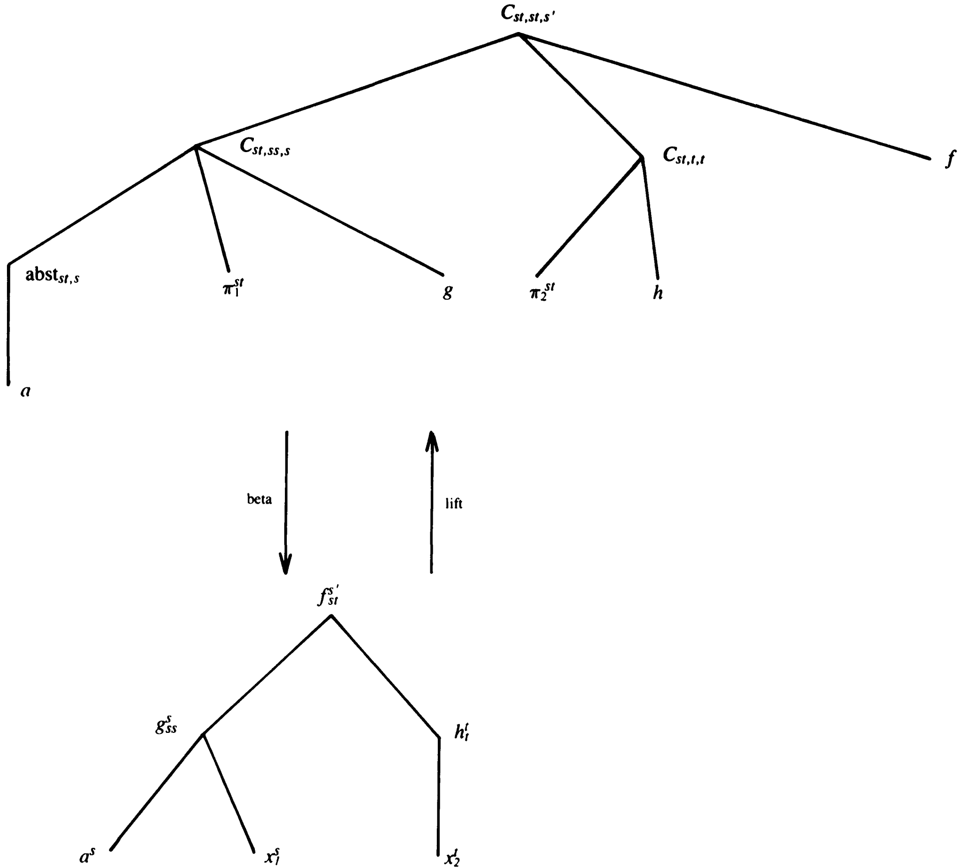
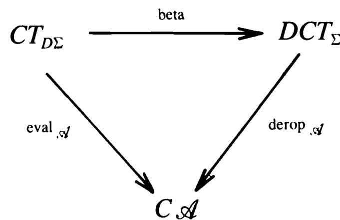


FIG. 1. *Tree with variables.*

We have the following useful lemma (see Engelfriedt and Schmidt [16]).

LEMMA 2.4. *For any ω-continuous Σ-algebra $\mathscr{A}$, we have* $\text{beta} \cdot \text{derop}_{\mathscr{A}} = \text{eval}_{\mathscr{A}}$, *where* $\text{eval}_{\mathscr{A}}$ *is the unique ω-continuous morphism from $CT_{D\Sigma}$.*



The lemma holds, because since $CT_{D\Sigma}$ is initial there is a unique morphism $\text{eval}_{\mathscr{A}}$ from $CT_{D\Sigma}$ to $\mathscr{C}\mathscr{A}$, and all functions involved are continuous morphisms. A similar lemma also holds for ordered Σ-algebras.

A rational scheme of higher type will be defined as a scheme over some derived alphabet $D^n\Sigma$. To provide its semantics, such a scheme will be unfolded into an infinite

tree in $CT_{D^n\Sigma}$, which will then be mapped into DCT_Σ by a function beta^n (obtained by iterating beta). However, not all trees in $CT_{D^n\Sigma}$ can be mapped into DCT_Σ, as we now explain. For every $n \geqq 0$, since $CT_{D^{n+1}\Sigma}$ is the initial D^{n+1} Σ-algebra, there is a unique morphism

$$\text{beta}_{n+1}: CT_{D^{n+1}\Sigma} \to DCT_{D^n\Sigma}.$$

It is tempting to define beta^{n+1} as the function $\text{beta}_{n+1} \cdot \text{beta}_n \cdot \cdots \cdot \text{beta}$ obtained by composing the morphisms $\text{beta}_{n+1}, \text{beta}_n, \cdots, \text{beta}$, but this composite function is not well defined. This is because the sources and targets of these functions do not match up. In particular, $DCT_{D^n\Sigma} \neq CT_{D^n\Sigma}$. Indeed, for each $(\bar{u}, \bar{s})$ in $D^{n+1}\Sigma$, by Definition 2.17, the carrier $(DCT_{D^n\Sigma})_{(\bar{u},\bar{s})}$ of sort $(\bar{u}, \bar{s}) \in D^{n+1}\Sigma$ of the algebra $DCT_{D^n\Sigma}$ is $CT_{D^n\Sigma}(\bar{u}, \bar{s})$, and for $\bar{u} \neq e$, this set contains trees with variables. However, for all $\bar{s}$ in $D^n\Sigma$, $(DCT_{D^n\Sigma})_{(e,\bar{s})} = (CT_{D^n\Sigma})_{\bar{s}}$ (the carrier of sort $\bar{s}$ of the algebra $CT_{D^n\Sigma}$). As a consequence, the $\bar{s}$-component of beta$_n$ is well defined on the carrier of sort $(e, \bar{s})$ of the algebra $DCT_{D^n\Sigma}$, and can be composed with the $(e, \bar{s})$-component of beta$_{n+1}$. Hence, we will define a restricted class of sorts $(b_{S,n}(u, s))$ such that trees in $CT_{D^n\Sigma}$ of this sort can be mapped into DCT_Σ.

DEFINITION 2.22. Given (u, s) in DS, define $b_{S,n}(u, s)$ as follows: $b_{S,0}(u, s) = s$, $b_{S,1}(u, s) = (u, s)$ and $b_{S,n+1}(u, s) = (e, b_{S,n}(u, s))$. We will often omit the subscript S for simplicity.

We obtain, by induction, the family of functions $\text{beta}^n = (\text{beta}^n_{u,s}: CT_{D^n\Sigma}(e, b_n(u, s)) \to (DCT_\Sigma)_{(u,s)} | (u, s) \in DS)$ defined as follows:

$$\text{beta}^0 = \text{id}, \qquad \text{beta}^1 = \text{beta}: CT_{D\Sigma} \to DCT_\Sigma,$$

(since $(DCT_\Sigma)_{(u,s)} = CT_\Sigma(u, s)$) and for $n \geqq 1$,

$$\text{beta}^{n+1}_{u,s} = \text{beta}_{b_{n+1}(u,s)} \cdot \text{beta}^n_{u,s},$$

where $\text{beta}_{b_{n+1}(u,s)}$ is the component (on the carrier of sort $b_{n+1}(u, s)$) of the unique morphism

$$\text{beta}_{n+1}: CT_{D^{n+1}\Sigma} \to DCT_{D^n\Sigma}.$$

Note that beta is *not* a morphism from $CT_{D^n\Sigma}$ to DCT_Σ since it is only partially defined.

DEFINITION 2.23. Given an ordered Σ-algebra $\mathcal{A}$, we define the algebras $\mathcal{F}^n\mathcal{A}$ as follows: $\mathcal{F}^1\mathcal{A} = \mathcal{F}\mathcal{A}$, and $\mathcal{F}^{n+1}\mathcal{A} = \mathcal{F}(\mathcal{F}^n\mathcal{A})$. A similar definition applies to continuous Σ-algebras with $\mathcal{C}$ instead of $\mathcal{F}$. The following fact can also be shown by induction using Lemma 2.4.

LEMMA 2.24. *For every t in $CT_{D^n\Sigma}(e, b_n(u, s))$,*

$$\text{eval}_{\mathcal{C}^n\mathcal{A}}(t) = \text{derop}_{\mathcal{A}}(\text{beta}^n(t)).$$

We now define another fundamental tool, the concept of a rational scheme.

DEFINITION 2.25. We will be considering schemes with parameters. Note that we are assuming that Σ and the S-indexed family of variables X are mutually disjoint. A *rational scheme with parameters of type* (v, s), for short, a *rational scheme*, is a pair (α, ℓ), where α is a function $\alpha: X_u \to FT_\Sigma$ with each $\alpha_i = \alpha(x^u_i)$ a tree in $FT_\Sigma(vu, u_i)$ for some u in S^+, v in S^*, and ℓ is a tree in $FT_\Sigma(vu, s)$ called the *main definition.*

If $u = u_1 \cdots u_m$ and $v = v_1 \cdots v_n$, α can be viewed as a set of recursive definitions for the functions $x_{n+1}, \cdots, x_{n+m}$, the variables $x_1, \cdots, x_n$ being treated as parameters and ℓ can be viewed as a main program with calls to the functions $x_{n+1}, \cdots, x_{n+m}$, and

with $x_1, \cdots, x_n$ as parameters:

$$x_{n+1} = a_1(x_1, \cdots, x_n, x_{n+1}, \cdots, x_{n+m}),$$
$$\cdots$$
$$x_{n+m} = a_m(x_1, \cdots, x_n, x_{n+1}, \cdots, x_{n+m}).$$

DEFINITION 2.26. Given an ω-continuous Σ-algebra $\mathcal{A}$ and a rational scheme with parameters (a, b) of type (v, s), the sequence of functions $a_{\mathcal{A}}^{(i)}: A^v \to A^{vu}$ is defined inductively as follows:

$$a_{\mathcal{A}}^{(0)} = (I_v, \perp_{v,u}),$$

where $\perp_{v,u}: A^v \to A^u$ is the "bottom function" and $I_v: A^v \to A^v$ is the identity, and

$$a_{\mathcal{A}}^{(i+1)} = a_{\mathcal{A}}^{(i)} \cdot (\pi_v^{vu}, a_{\mathcal{A}}),$$

where $\pi_v^{vu}: A^{vu} \to A^v$ is the projection on the v components. (Given two functions $f: A \to B$ and $g: A \to C$, $(f, g): A \to B \times C$ denotes the function obtained by "target tupling," that is, the function such that $(f, g)(x) = (f(x), g(x))$ for all x in A.) The function $a_{\mathcal{A}}^*$ is the least upper bound of the chain $\{a_{\mathcal{A}}^{(i)} | i \in N\}$. It is easily seen that $a_{\mathcal{A}}^* = (I_v, a_{\mathcal{A}}^+)$, where the function $a_{\mathcal{A}}^+: A^v \to A^u$ is the least upper bound of the chain $\{a_{\mathcal{A}}^{(i)} \cdot a_{\mathcal{A}} | i \in N\}$. Furthermore, $a_{\mathcal{A}}^+$ is the least fixed-point of the functional equation

$$h = (I_v, h) \cdot a_{\mathcal{A}},$$

where h ranges over functions in $[A^v \to A^u]$ (see Goguen, Thatcher, Wagner and Wright [2], Thatcher, Wagner and Wright [42], [43] and Gallier [20], [21] for further details).

DEFINITION 2.27. The *meaning* of (a, b) in the interpretation $\mathcal{A}$ is then the function $a_{\mathcal{A}}^* \cdot b_{\mathcal{A}}: A^v \to A_s$, which is also denoted by $(a, b)_{\mathcal{A}}$ (where $b_{\mathcal{A}}$ is the function $\text{derop}_{\mathcal{A}}(b)$).

From the Mezei–Wright theorem [32], the meaning of the scheme (a, b) can also be obtained by first "unfolding" the scheme a in the free interpretation $CT_\Sigma(v, vu)$ thus obtaining the tuple of trees a^* (in $CT_\Sigma(v, vu)$), then composing it with b obtaining $a^* \circ b$, and finally interpreting the resulting tree as the derived operator $(a^* \circ b)_{\mathcal{A}}$.

DEFINITION 2.28. The *unfoldment* a^* of a is the least upper bound of the sequence $a^{(i)}$ defined inductively as follows:

$$a^{(0)} = (I_v, \perp_{v,u}) \quad \text{where } I_v = (x_1^v, \cdots, x_n^v) \quad (n = |v|),$$
$$a^{(i+1)} = a^{(i)} \circ (\pi_v^{vu}, a),$$

where $\circ$ is tree composition and $\pi_v^{vu} = (x_1^{vu}, \cdots, x_n^{vu})$. Again, we define a^+ as the last $m = |u|$ components of a^*, and we have $a^* = (I_v, a^+)$. Note that $a^+ = a^*$ when $v = e$, that is when there are no parameters. For simplicity of notation, we often identify π_v^{vu} with I_v.

The fact that the meaning of the scheme (a, b) in the interpretation $\mathcal{A}$ is equal to the derived operator $(a^* \circ b)_{\mathcal{A}}$ is crucial to our further investigations. Indeed, calling trees of the form $a^* \circ b$ *rational trees*, the above condition can be restated by saying that rational trees induce well defined derived operators. The key idea behind the definition of a n-rational algebra is to generalize the concept of a rational tree, by creating a hierarchy of families of trees, the n-rational trees. Such trees are obtained by unfolding rational schemes (of a certain sort) over derived alphabets of the form $D^n\Sigma$ and applying the mapping beta^n. Roughly speaking, an n-rational algebra is then an algebra such that n-rational trees induce well defined derived operators.

It should be emphasized that we are going to consider a restricted family of schemes. We could define a more general class of schemes with higher type parameters, but the unwinding process via the beta's would not work. Only schemes for which the unwinding process works all the way down are considered.

DEFINITION 2.29. An *n-rational scheme of type $b_{n+1}(v, s)$*, for short an n-rational scheme, is a rational scheme (with parameters) over the derived alphabet $D^n\Sigma$. Note that an n-rational scheme actually has parameters only for $n=0$. Given an ω-continuous Σ-algebra $\mathcal{A}$, the meaning of an n-rational scheme of type $b_{n+1}(v, s)$, when interpreted in $\mathcal{C}^n\mathcal{A}$, is a function in $[A^v \to A_s]$.

Furthermore, for $n \geqq 1$, from the Mezei–Wright theorem [32], the meaning of the scheme $(\mathscr{a}, \mathscr{b})$ is the derived operator $(\mathscr{a}^* \circ \mathscr{b})_{\mathcal{C}^n\mathcal{A}}$, which is also equal to $\text{eval}_{\mathcal{C}^n\mathcal{A}}(\mathscr{a}^* \circ \mathscr{b})$, since $\mathscr{a}^* \circ \mathscr{b}$ is a "constant" tree in $CT_{D^n\Sigma}(e, b_n(v, s))$. Using Lemma 2.5, we then have

$$\text{eval}_{\mathcal{C}^n\mathcal{A}}(\mathscr{a}^* \circ \mathscr{b}) = (\text{beta}^n(\mathscr{a}^* \circ \mathscr{b}))_{\mathcal{A}}.$$

Hence the meaning of a level n-rational scheme of type $b_{n+1}(v, s)$ is also a derived operator induced by a (generally infinite) tree. Furthermore, this tree $\text{beta}^n(\mathscr{a}^* \circ \mathscr{b})$ is obtained by unfolding the n-rational scheme (at level n) into the tree $\mathscr{a}^* \circ \mathscr{b}$ over $D^n\Sigma$, and then "bringing down" this tree to $CT_\Sigma(v, s)$ by applying the mapping beta^n which performs recursively all the substitutions specified by the substitution operators in that tree.

Example 2.2. To avoid excessive subscripting, let us define the following symbols:

$$K_1 = C_{(1,1),(11,1)(1,1)(1,1),(1,1)}, \qquad L_1 = \text{abst}_{(1,1),(11,1)},$$
$$K_2 = C_{(1,1),(1,1)(1,1),(1,1)}, \qquad L_2 = \text{abst}_{(1,1),(1,1)},$$
$$K_3 = C_{(1,1),(1,1),(1,1)}, \qquad L_3 = \text{abst}_{e,(1,1)},$$
$$K_4 = C_{e,(1,1),(1,1)}, \qquad L_4 = \text{abst}_{e,(e,1)},$$
$$K_5 = C_{e,(e,1),(e,1)}, \qquad P = \pi_1^{(1,1)}.$$

Also, F is a variable of sort $((1,1),(1,1))$ in $D^2\Sigma$, G is a variable of sort $(e,(1,1))$ in $D^2\Sigma$ and H is a variable of sort $(e,(e,1))$. Σ contains a binary symbol $+$, a unary symbol f and a constant a (we assume that the set of sorts is $S=\{1\}$). We then have the rational scheme of level 2 (shown in Fig. 2).

It can be verified that beta of the definition for F is the tree shown in Fig. 3, and that beta^2 of the component of the unfoldment for H is the tree of Fig. 4.

3. n-rational algebras, basic properties. In this section we define the class of n-rational Σ-algebras for every $n \geqq 0$. Such algebras are defined so that all n-rational schemes of type $b_{n+1}(v, s)$ have a well-defined meaning when interpreted in them. Technically, this means that every derived operator induced by a (possibly infinite) tree of the form $\text{beta}^n(\mathscr{a}^* \circ \mathscr{b})$ for some n-rational scheme $(\mathscr{a}, \mathscr{b})$ is well defined. This requires the existence of least upper bounds of certain "constructive" chains arising from unfolding n-rational schemes. Such a condition is referred to as *n-rational completeness.* Another condition is also necessary to guarantee that such least upper bounds are indeed fixed-points: certain derived operators must preserve these least upper bounds. This second condition is referred to as *n-rational continuity.* An ordered algebra is n-rational if it is n-rationally complete and operators are n-rationally continuous.

We now proceed with the formal definitions. We first define the set of n-rational trees, and n-iterations which are the "constructive" chains of interest. Then, we show that free n-rational algebras consist of n-rational trees.

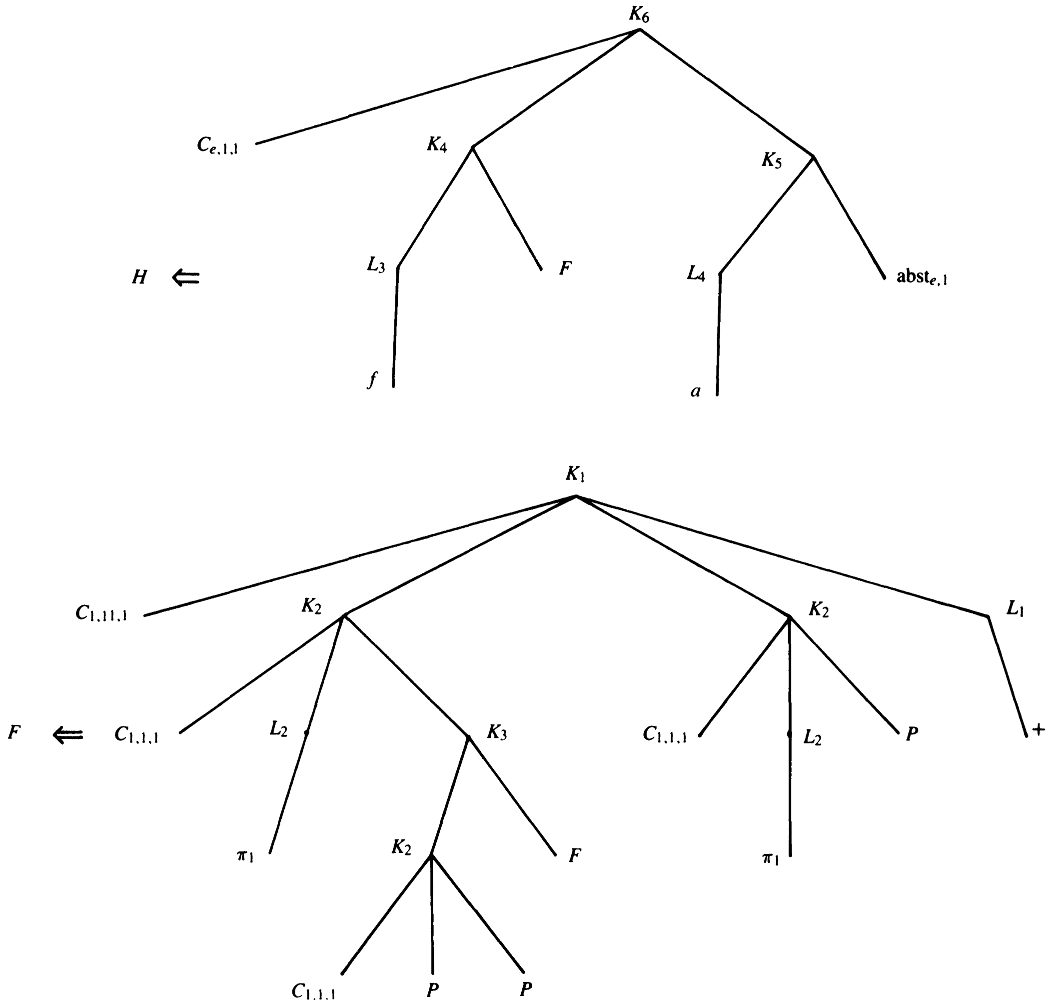


FIG. 2

Recall that X denotes the S-indexed family of countable sets of variables.

DEFINITION 3.1. The set of *n-rational trees of type* (v, s), denoted by $n\text{-}RT_\Sigma(v, s)$, is the set of all trees of the form $\text{beta}^n(\mathscr{a}^* \circ \mathscr{b})$, for some n-rational scheme $(\mathscr{a}, \mathscr{b})$ of type $b_{n+1}(v, s)$. The set of all n-rational trees is denoted by $n\text{-}RT_\Sigma(X)$. Given an arbitrary S-indexed family Y, we define $n\text{-}RT^s_\Sigma(Y)$ as $n\text{-}RT_{\Sigma\cup Y}(e, s)$ and $n\text{-}RT_\Sigma(Y)$ as the S-indexed family $(n\text{-}RT^s_\Sigma(Y) | s \in S)$.

We will prove shortly that $n\text{-}RT_\Sigma(Y)$ is the free n-rational algebra over Y.

DEFINITION 3.2. Given an ordered Σ-algebra $\mathscr{A}$, an *n-iteration* is a sequence $E = \{(\text{beta}^n(\mathscr{a}^{(i)} \circ \mathscr{b}))_{\mathscr{A}}(c) | i \geq 0\}$, for some n-rational scheme $(\mathscr{a}, \mathscr{b})$ of type $b_{n+1}(v, s)$ and some c in A^v.

Since $\mathscr{A}$ is an ordered Σ-algebra, beta^n is monotonic and since the trees $\text{beta}^n(\mathscr{a}^{(i)} \circ \mathscr{b})$ are finite, it is easily shown that an n-iteration is an ascending chain.

DEFINITION 3.3. An ordered Σ-algebra $\mathscr{A}$ is *n-rationally complete* if every n-iteration E has a least upper bound in $\mathscr{A}$.

DEFINITION 3.4. Given an n-rationally complete Σ-algebra $\mathscr{A}$, a function $h: A^u \to A_s$ is *n-rationally continuous* if, for every m-tuple $(m = |u|)$ of n-iterations

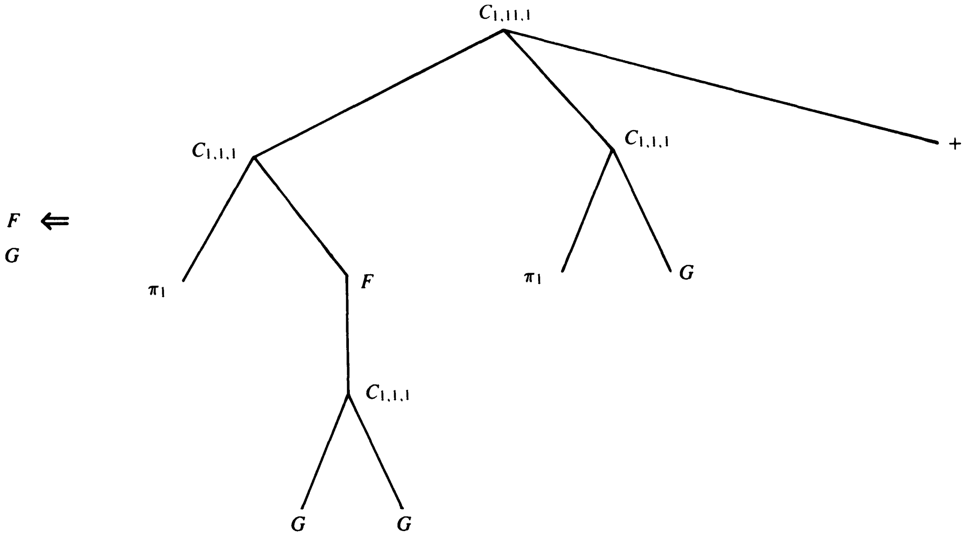


FIG. 3

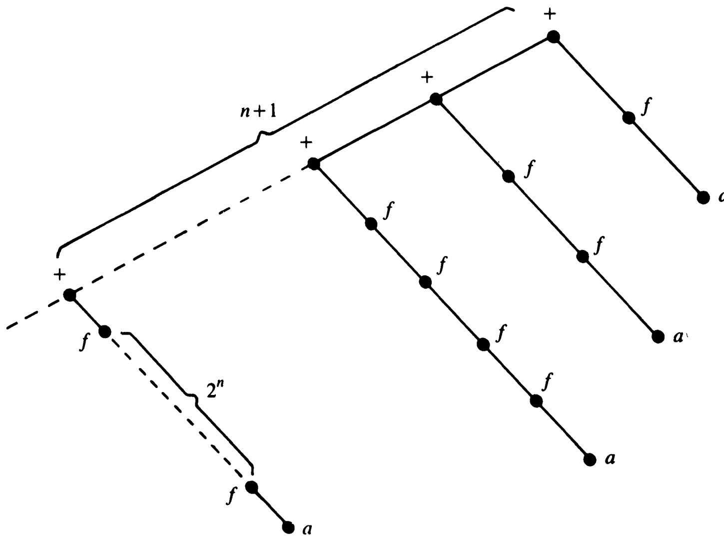


FIG. 4. *Note that the paths in this tree are encoded by a language which is not context-free.*

$(E_1, \cdots, E_m)$ with $E_j = \{a_i^j | i \in N\}$ a subset of A_{u_j}, we have:

(1) $\{h(a_i^1, \cdots, a_i^m) | i \in N\}$ is a subset of some n-iteration E' in A_s;

(2) $h(\sqcup a_i^1, \cdots, \sqcup a_i^m) = \sqcup h(a_i^1, \cdots, a_i^m)$.

Note that for $n = 0$, 0-rational completeness is equivalent to Tiuryn's "algebraic completeness" and 0-rational continuity is equivalent to Tiuryn's "algebraic continuity" [44], [45], [46], [47].

DEFINITION 3.5. An ordered Σ-algebra $\mathcal{A}$ is *n-rational* if:

(1) it is n-rationally complete;

(2) for every function symbol f in $\Sigma_{u,s}$, $f_{\mathcal{A}} : A^u \to A_s$ is n-rationally continuous.

For every $n \geqq 0$, the class of n-rational Σ-algebras is denoted as n-R-ALG_Σ. It is immediately verified that every ω-continuous algebra is n-rational (for every n). It

is also easy to show that an n-rational algebra is i-rational for every $i<n$ $(n>0)$. Hence we have a hierarchy ending with 0-rational algebras and starting with ω-continuous algebras. It is shown in Theorem 3.11 that this hierarchy is strict.

LEMMA 3.6. *Given an n-rational Σ-algebra $\mathcal{A}$, for every finite tree t in $FT_\Sigma(uv, s)$, for every c in A^v, the function $g: A^u \to A_s$ such that $g(x) = t_\alpha(x, c)$ for all x in A^u is n-rationally continuous.*

Proof. First, observe that for every $n \geqq 0$, n-rational completeness implies that every c in A_s is the least upper bound of an n-iteration $\{c_i | i \in N\}$ such that $c_0 = \perp_s$ and $c_i = c$ for all $i \geqq 1$. Hence, it is obvious that it suffices to prove that finite derived operators are n-rationally continuous. We proceed by induction on the depth of trees. The property clearly holds for trees of depth zero since a tree of depth zero is either a variable or a constant. If $t = ft_1 \cdots t_k$, then for any m-tuple of n-iterations $(E_1, \cdots, E_m)$ with $E_j = \{a_i^j | i \in N\}$,

$$t_{\mathcal{A}}(\sqcup a_i^1, \cdots, \sqcup a_i^m)$$

$$= f_{\mathcal{A}}((t_1)_{\mathcal{A}}(\sqcup a_i^1, \cdots, \sqcup a_i^m), \cdots, (t_k)_{\mathcal{A}}(\sqcup a_i^1, \cdots, \sqcup a_i^m))$$

(by the remark following Definition 2.10)

$$= f_{\mathcal{A}}(\sqcup (t_1)_{\mathcal{A}}(a_i^1, \cdots, a_i^m), \cdots, \sqcup (t_k)_{\mathcal{A}}(a_i^1, \cdots, a_i^m))$$

(by the inductive hypothesis)

$$= \sqcup f_{\mathcal{A}}((t_1)_{\mathcal{A}}(a_i^1, \cdots, a_i^m), \cdots, (t_k)_{\mathcal{A}}(a_i^1, \cdots, a_i^m))$$

(since $f_{\mathcal{A}}$ is n-rationally continuous, Definition 3.4)

$$= \sqcup (ft_1 \cdots t_k)_{\mathcal{A}}(a_i^1, \cdots, a_i^m) \quad \text{(by the remark following Definition 2.10)}$$

$$= \sqcup t_{\mathcal{A}}(a_i^1, \cdots, a_i^m) \quad \text{(by the definition of } t\text{).}$$

DEFINITION 3.7. An *n-rational morphism* $h: \mathcal{A} \to \mathcal{B}$ is an S-indexed family of functions $h_s: A_s \to B_s$ such that:

(1) h is a morphism of ordered Σ-algebras;

(2) for every n-iteration E in A_s, $h(\sqcup E) = \sqcup h(E)$.

We have a similar definition if h is only a weak morphism of ordered Σ-algebras. In both cases, since h is strict and a homomorphism, it is easy to verify that $h(E)$ is an n-iteration in $\mathcal{B}$.

We shall now prove that n-$RT_\Sigma(Y)$ is the free n-rational algebra on Y. First, we show that it is an n-rational algebra. This is proved using two lemmas. Note that Lemma 3.8 states a result which goes beyond what is actually needed to prove that n-$RT_\Sigma(Y)$ is an n-rational algebra.

LEMMA 3.8. *For every m-tuple $(t_1, \cdots, t_m)$ of trees with each t_j a tree of sort v_j in n-$RT_\Sigma(Y)$ and every tree t in n-$RT_\Sigma(v, s)$ (with $m = |v|$), $(t_1, \cdots, t_m) \circ t$ is in n-$RT_\Sigma(Y)$.*

Proof. First, the following claim is shown by induction on $n \geqq 1$.

CLAIM. *For all $n \geqq 1$, for every m-tuple $(T_1, \cdots, T_m)$ of trees with each T_i in 0-$RT_{D^n\Sigma}(e, b_n(u, v_i))$ for $i = 1, \cdots, m$, for every tree T in 0-$RT_{D^n\Sigma}(e, b_n(v, s))$ $(m = |v|)$, a tree COMP_n in $FT_{D^n\Sigma}(U, b_n(u, s))$ with $U = b_n(u, v_1) \cdots b_n(u, v_m) b_n(v, s)$ can be defined inductively, such that*:

(i) COMP_n *is of the form* $C_n(X_1, \cdots, X_{m+1}, R_1, \cdots, R_{n-1})$;

(ii) $(\mathrm{beta}^n(T_1), \cdots, \mathrm{beta}^n(T_m)) \circ \mathrm{beta}^n(T) = \mathrm{beta}^n((T_1, \cdots, T_m, T) \circ \mathrm{COMP}_n)$

(*where C_n is a composition combinator in $D^n\Sigma$, and $X_1, \cdots, X_m, X_{m+1}$ are variables of sorts $b_n(u, v_1), \cdots, b_n(u, v_m), b_n(v, s)$, respectively*). *The trees $R_1, \cdots, R_{n-1}$ are also defined during the course of the induction.*

Proof of claim. For $n=1$, we have $(\text{beta}(T_1), \cdots, \text{beta}(T_m)) \circ \text{beta}(T) = \text{beta}(C_{u,v,s}(T_1, \cdots, T_m, T))$, by Definition 2.20. Defining $\text{COMP}_1 = C_{u,v,s}(X_1, \cdots, X_m, X_{m+1})$, the claim holds.

For $n>1$, since $\text{beta}^n = \text{beta}_n \cdot \text{beta}^{n-1}$ (by Definition 2.22), we have

$$(\text{beta}^n(T_1), \cdots, \text{beta}^n(T_m)) \circ \text{beta}^n(T)$$
$$= (\text{beta}^{n-1}(\text{beta}_n(T_1)), \cdots, \text{beta}^{n-1}(\text{beta}_n(T_m))) \circ \text{beta}^{n-1}(\text{beta}_n(T)) \quad \text{(by Definition 2.22)}$$
$$= \text{beta}^{n-1}((\text{beta}_n(T_1), \cdots, \text{beta}_n(T_m), \text{beta}_n(T)) \circ \text{COMP}_{n-1}) \quad \text{(by the inductive hypothesis)},$$

where COMP_{n-1} is of the form $C_{n-1}(Y_1, \cdots, Y_m, Y_{m+1}, R_1, \cdots, R_{n-2})$ for some composition combinator C_{n-1} in $D^{n-1}\Sigma$. Then,

$$(\text{beta}_n(T_1), \cdots, \text{beta}_n(T_m), \text{beta}_n(T)) \circ \text{COMP}_{n-1}$$
$$= C_{n-1}(\text{beta}_n(T_1), \cdots, \text{beta}_n(T_m), \text{beta}_n(T), R_1, \cdots, R_{n-2}).$$

Using the fact that $\text{lift}_n \cdot \text{beta}_n = \text{id}$, and that $\text{beta}_n(C_{n-1}) = C_{n-1}(Y_1, \cdots, Y_m, Y_{m+1})$, we have

$$C_{n-1}(\text{beta}_n(T_1), \cdots, \text{beta}_n(T_m), \text{beta}_n(T), R_1, \cdots, R_{n-2})$$
$$= (\text{beta}_n(T_1), \cdots, \text{beta}_n(T_m), \text{beta}_n(T),$$
$$\text{beta}_n(\text{lift}_n(R_1)), \cdots, \text{beta}_n(\text{lift}_n(R_{n-2}))) \circ \text{beta}_n(C_{n-1})$$
$$= \text{beta}_n(C_n(T_1, \cdots, T_m, T, \text{lift}_n(R_1), \cdots, \text{lift}_n(R_{n-2}), C_{n-1})) \quad \text{(by definition of beta}_n\text{, Definition 2.22)}$$

for some appropriate composition combinator C_n in $D^n\Sigma$. Defining $\text{COMB}_n = C_n(X_1, \cdots, X_m, X_{m+1}, \text{lift}_n(R_1), \cdots, \text{lift}_n(R_{n-2}), C_{n-1})$, the inductive step is established, and the claim is proved.

Assume that t is defined by the scheme (a, b) of type $b_{n+1}(v, s)$ and that each t_j is defined by the scheme (a_j, b_j) of type $b_{n+1}(u, v_j)$. Then

$$(t_1, \cdots, t_m) \circ t = (\bigsqcup \text{beta}^n(a_1^{(i)} \circ b_1), \cdots, \bigsqcup \text{beta}^n(a_m^{(i)} \circ b_m)) \circ (\bigsqcup \text{beta}^n(a^{(i)} \circ b)).$$

Let $T_j^i = (a_j^{(i)} \circ b_j)$ for $j=1, \cdots, m$ and $T^i = (a^{(i)} \circ b)$. By continuity of composition of trees,

$$(\bigsqcup \text{beta}^n(T_1^i), \cdots, \bigsqcup \text{beta}^n(T_m^i)) \circ (\bigsqcup \text{beta}^n(T^i))$$
$$= \bigsqcup (\text{beta}^n(T_1^i), \cdots, \text{beta}^n(T_m^i)) \circ \text{beta}^n(T^i).$$

From the above proved claim, there exists a tree COMP_n in $FT_{D^n\Sigma}$ of the form $C_n(X_1, \cdots, X_{m+1}, R_1, \cdots, R_{n-1})$, and such that

$$(\text{beta}^n(T_1^i), \cdots, \text{beta}^n(T_m^i)) \circ \text{beta}^n(T^i)$$
$$= \text{beta}^n((T_1^i, \cdots, T_m^i, T^i) \circ \text{COMP}_n).$$

It only remains to show that we can construct a scheme (g, d) such that, for all $i \geqq 0$,

$$g^{(i)} \circ d = (a_1^{(i)} \circ b_1, \cdots, a_m^{(i)} \circ b_m, a^{(i)} \circ b) \circ \text{COMP}_n.$$

However, this is known from Goguen, Thatcher, Wagner and Wright [2] and Tiuryn [44], [45]. To avoid an abundance of variables, only the schemes a and b will

be written as sets of equations. To avoid ambiguity, the ith component of the scheme $\mathscr{a}$ will be denoted by $\mathscr{a}^i$ and not by $\mathscr{a}_i$, which denotes another scheme. The scheme $\mathscr{g}$ consists of the following definitions:

$$X_0 \Leftarrow C_n(X_1, \cdots, X_m, X_{m+1}, R_1, \cdots, R_{n-1})$$

$$\mathscr{a}_1$$
$$\vdots$$
$$\mathscr{a}_m$$

$$Y_{q+1} \Leftarrow \mathscr{a}^1(Y_1, \cdots, Y_q, Y_{q+1}, \cdots, Y_{q+p})$$
$$\vdots$$
$$Y_{q+p} \Leftarrow \mathscr{a}^p(Y_1, \cdots, Y_q, Y_{q+1}, \cdots, Y_{q+p})$$
$$X_1 \Leftarrow \mathscr{b}_1$$
$$\vdots$$
$$X_m \Leftarrow \mathscr{b}_m$$
$$X_{m+1} \Leftarrow \mathscr{b}(Y_{q+1}, \cdots, Y_{q+p}).$$

Let $\mathscr{d}$ be the projection picking out the first component of $\mathscr{g}$. By continuity of (tree) composition, we have

$$\bigsqcup \text{beta}^n(\mathscr{g}^{(i)} \circ \mathscr{d}) = (t_1, \cdots, t_m) \circ t.$$

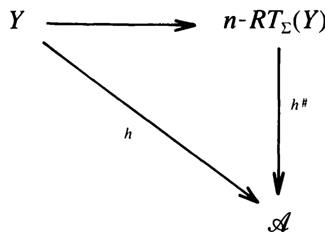
LEMMA 3.9. *n-$RT_\Sigma(Y)$ is an n-rational Σ-algebra.*

Proof. Letting $t = f(x_1, \cdots, x_n)$ in Lemma 3.8 shows that $f(t_1, \cdots, t_n)$ is in n-$RT_\Sigma(Y)$. Next, observe that in $\mathscr{A} = n$-$RT_\Sigma(Y)$, for every finite tree t in $FT_\Sigma(v, s)$, $t_{\mathscr{A}}(t_1, \cdots, t_m) = (t_1, \cdots, t_m) \circ t$. To show that n-$RT_\Sigma(Y)$ is n-rationally complete, we have to show that every n-iteration $\{(t_1, \cdots, t_m) \circ \text{beta}^n\, (\mathscr{a}^{(i)} \circ \mathscr{b})\}$ (for some n-rational scheme $(\mathscr{a}, \mathscr{b})$ of type $b_{n+1}(v, s)$ and some $(t_1, \cdots, t_m)$ in $(n\text{-}RT_\Sigma(Y))^v$, with each t_i of sort v_i), has a least upper bound in n-$RT_\Sigma(Y)$. Letting $t = \bigsqcup \text{beta}^n\, (\mathscr{a}^{(i)} \circ \mathscr{b})$, we conclude using Lemma 3.8 and the continuity of (tree) composition.

Finally, to prove that n-$RT_\Sigma(Y)$ is an n-rational Σ-algebra, we also have to check condition (2) of Definition 3.5. Since f is ω-continuous in $CT_\Sigma(Y)$, f is indeed n-rationally continuous.

We now show n-$RT_\Sigma(Y)$ is in fact a free Σ-algebra on Y.

THEOREM 3.10. *n-$RT_\Sigma(Y)$ is the free n-rational Σ-algebra on Y, that is, for every S-indexed family of functions $h: Y \to A$ from Y to the carrier of any n-rational Σ-algebra $\mathscr{A}$, there is a unique n-rational morphism $h^\#: n\text{-}RT_\Sigma(Y) \to \mathscr{A}$ extending h. Furthermore, $h^\#$ is ω-continuous.*



Proof. Since $\mathscr{A}$ is an ordered Σ-algebra, there is a unique monotonic morphism $h_0: FT_\Sigma(Y) \to \mathscr{A}$ extending h. Since every element in n-$RT_\Sigma(Y)$ is a tree of the form $t = \bigsqcup\{\text{beta}^n\, (\mathscr{a}^{(i)} \circ \mathscr{b}) |$ for some scheme $(\mathscr{a}, \mathscr{b})$ over $D^n(\Sigma UY)$ of sort $b_{n+1}(e, s)\}$ and

since each tree beta^n $(a^{(i)} \circ b)$ is finite, if we want h^* to be an n-rational morphism, it is uniquely defined by

$$h^*(t) = \bigsqcup \{h_0(\text{beta}^n\ (a^{(i)} \circ b)) \mid i \in N\}.$$

We have to show that the definition of $h^*(t)$ is independent of the choice of the n-iteration defining t, and that it is ω-continuous. Let $E = \{a_i \mid i \in N\}$ and $E' = \{a'_i \mid i \in N\}$ be two n-iterations defining t. From Proposition 2.7, E and E' are mutually cofinal, that is, $\forall i \exists j$ such that $a_i \leqq a'_j$ and $\forall i \exists j$ such that $a'_i \leqq a_j$. Since h_0 is monotonic, $h_0(E)$ and $h_0(E')$ are also mutually cofinal, which implies $\bigsqcup h(E) = \bigsqcup h(E')$. To show that h^* is monotonic, one simply observes that given any two trees $t_1 = \bigsqcup E$, $t_2 = \bigsqcup E'$ in n-$RT_\Sigma(Y)$, $t_1 \leqq t_2$ implies that E' is cofinal in E, and by monotonicity of h_0, $h_0(E')$ is cofinal in $h_0(E)$, which implies $h^*(t_1) = \bigsqcup h(E) \leqq \bigsqcup h(E') = h^*(t_2)$.

Let $L = \{t_i \mid i \in N\}$ be any chain in n-$RT_\Sigma(Y)$ having least upper bound t. Let $E = \{s_j \mid j \in N\}$ be an n-iteration such that $t = \bigsqcup E$. Since $\bigsqcup E = \bigsqcup L$, by Proposition 2.7, for every s_i there is some t_j such that $s_i \leqq t_j$. Since h^* is monotonic, for every i there is some j such that $h^*(s_i) \leqq h^*(t_j) \leqq h^*(t)$. Hence, $h^*(t)$ is a least upper bound for $h^*(E)$ and $h^*(L)$ is cofinal with $h^*(E)$. Let c be any upper bound for $h^*(L) = \{h^*(t_i) \mid i \in N\}$. Since $h^*(L)$ is cofinal with $h^*(E)$, c is also an upper bound for $h^*(E)$ and we have $h^*(t) = h^*(\bigsqcup E) = \bigsqcup h^*(E) \leqq c$. Hence, $h^*(t)$ is the least upper bound of $h^*(L)$, and we have $h^*(t) = h^*(\bigsqcup L) = \bigsqcup h^*(L)$, establishing the continuity of h^*.

Finally, we have to check that h^* is a homomorphism. This follows because finite trees are n-rationally continuous. Let $t_1, \cdots, t_m$ be in n-$RT_\Sigma(Y)$ and assume that $t_j = \bigsqcup t_i^j$. Let f be a symbol in $\Sigma_{u,s}$ with $m = |u|$. Then, we have:

$$\begin{aligned}
h^*(f(t_1, \cdots, t_m)) &= h^*(f(\bigsqcup t_i^1, \cdots, \bigsqcup t_i^m)) \\
&= h^*(\bigsqcup f(t_i^1, \cdots, t_i^m)) \quad \text{(since } f \text{ is } n\text{-rationally continuous)} \\
&= \bigsqcup h_0(f(t_i^1, \cdots, t_i^m)) \quad \text{(by definition of } h^*\text{)} \\
&= \bigsqcup f_{\mathcal{A}}(h_0(t_i^1), \cdots, h_0(t_i^m)) \quad \text{(since } h_0 \text{ is a homomorphism)} \\
&= f_{\mathcal{A}}(\bigsqcup h_0(t_i^1), \cdots, \bigsqcup h_0(t_i^m)) \quad \text{(since } f_{\mathcal{A}} \text{ is } n\text{-rationally continuous)} \\
&= f_{\mathcal{A}}(h^*(t_1), \cdots, h^*(t_m)) \quad \text{(by definition of } h^*\text{)}.
\end{aligned}$$

This concludes the proof that n-$RT_\Sigma(Y)$ is the free n-rational Σ-algebra on Y.

We are now in a position to prove that the hierarchy $(n\text{-}R\text{-ALG}_\Sigma)_{n \geqq 0}$ of classes of n-rational Σ-algebras is strict. This result is obtained as a consequence of Damm's "hierarchy theorem" [13] for level-n OI-tree languages.

THEOREM 3.11. *For every $n \geqq 0$, the initial n-rational Σ-algebra n-RT_Σ is not $(n+1)$-rational.*

Proof. Recall that n-RT_Σ consists of the n-rational trees without variables (Definition 3.1). It is shown in Damm [13, Cor. 4.11, Thm. 9.8] that n-RT_Σ is properly contained in $(n+1)$-RT_Σ. We proceed by contradiction. Assume that n-RT_Σ is $(n+1)$-rational. By the initiality of $(n+1)$-RT_Σ, there is an unique morphism $h:(n+1)\text{-}RT_\Sigma \to n\text{-}RT_\Sigma$, and composing h with the inclusion i from n-RT_Σ to $(n+1)$-RT_Σ, there are morphisms $h \cdot i:(n+1)\text{-}RT_\Sigma \to (n+1)\text{-}RT_\Sigma$ and $i \cdot h: n\text{-}RT_\Sigma \to n\text{-}RT_\Sigma$. Since n-RT_Σ is initial in n-R-ALG$_\Sigma$ and $(n+1)$-RT_Σ is initial in $(n+1)$-R-ALG$_\Sigma$, both $i \cdot h$ and $h \cdot i$ are identity functions, and h is an isomorphism. But since n-RT_Σ is a subset of $(n+1)$-RT_Σ, the restriction of $h \cdot i$ to n-RT_Σ is the identity, and since i is the inclusion from n-RT_Σ to $(n+1)$-RT_Σ, the restriction of h to n-RT_Σ is the identity. Since h is surjective (because it is an isomorphism), the restriction of

h to $n\text{-}RT_\Sigma$ is the identity, and $n\text{-}RT_\Sigma$ is properly contained in $(n+1)\text{-}RT_\Sigma$, h cannot be injective, contradicting the fact that it is an isomorphism.

4. Generation of *n*-rational subalgebras. In this section, we consider the generation of the least subalgebra generated by a subset of an n-rational algebra. We obtain a characterization similar to that given by Tiuryn [44], [45] for 0-rational algebras. We also obtain a convenient characterization of n-rational morphisms which is a direct extension of Tiuryn's condition [44], [45].

First, we show that the homomorphic images of n-rational algebras under an n-rational morphism (even a weak morphism) are also n-rational. This is a major difference from continuous algebras, which are not closed under images of continuous morphisms. The reason is that we are only considering certain "constructive" chains.

LEMMA 4.1. *Given a morphism of n-rational Σ-algebras, $h:\mathcal{A}\to\mathcal{B}$, $h(\mathcal{A})$ is also n-rational. The lemma is also true for a weak n-rational morphism.*

Proof. First, observe that the following property can easily be established: for any finite tree t in $FT_\Sigma(v, s)$, for any (weak) morphism $h:\mathcal{A}\to\mathcal{B}$ of ordered algebras, for any c in A^v, we have

$$h(t_{\mathcal{A}}(c)) = t_{\mathcal{B}}(h^v(c)).$$

Since h is strict and a homomorphism, $h(\mathcal{A})$ is an ordered algebra. Using the property stated above, given any iteration $F=\{(\text{beta}^n\ (\alpha^{(i)}\circ \ell))_{\mathcal{B}}(d)|i\in N\}$ in $h(\mathcal{A})$ there is an iteration $E=\{(\text{beta}^n\ (\alpha^{(i)}\circ \ell))_{\mathcal{A}}(c)|i\in N\}$ in $\mathcal{A}$ such that $h(E)=F$, where $h(c)=d$. Since h is n-rational, $h(\bigsqcup E)=\bigsqcup h(E)=\bigsqcup F$. Hence, $h(\mathcal{A})$ is n-rationally complete. Using the same technique, we can also show that operators are n-rationally continuous. This shows that $h(\mathcal{A})$ is n-rational.

Given an ordered Σ-algebra $\mathcal{A}$ and any S-indexed family X of subsets of the carriers of $\mathcal{A}$, the ordered subalgebra of $\mathcal{A}$ generated by X is denoted by $[X]$.

DEFINITION 4.2. Given an n-rational Σ-algebra $\mathcal{A}$ and any S-indexed family X of subsets of the carriers of $\mathcal{A}$, for every sort s, let

$$E(X)_s=\{\bigsqcup\{(\text{beta}^n\ (\alpha^{(i)}\circ \ell))_{\mathcal{A}}(x)|i\in N\}|(\alpha,\ell) \text{ is a } n\text{-rational scheme of type } b_{n+1}(v,s),\ x \text{ is in } X^v\}.$$

Let $E(X)$ be the S-indexed family $\{E(X)_s|s\in S\}$.

THEOREM 4.3. *For $\mathcal{A}$ and X as in Definition 4.2, $E([X])$ is the least n-rational subalgebra of $\mathcal{A}$ generated by X.*

Proof. Let Y be a set disjoint from Σ and in bijection with X. Let $f: Y\to A$ be a family of injections such that $f_s(Y_s)=X_s$.

Let $f^{\#}: n\text{-}RT_\Sigma(Y)\to\mathcal{A}$ be the unique n-rational morphism extending f.

CLAIM 1. *$f^{\#}(n\text{-}RT_\Sigma(Y))$ is the n-rational subalgebra generated by X.*

For this, we shall need:

CLAIM 2. *The carrier Z_s of sort s of $f^{\#}(n\text{-}RT_\Sigma(Y))$ is equal to $E([X])_s$.*

Assuming that Claim 2 has been proved, Claim 1 is proved as follows. Since $f^{\#}$ is an n-rational morphism, by Lemma 4.1, $f^{\#}(n\text{-}RT_\Sigma(Y))$ is an n-rational algebra, and because $f^{\#}(n\text{-}RT_\Sigma(Y))_s=E([X])_s$ by Claim 2, $f^{\#}(n\text{-}RT_\Sigma(Y))$ is the least n-rational algebra containing X.

It remains to prove Claim 2. First, we prove that $E([X])_s$ is a subset of Z_s. Given an n-iteration $L=\{(\text{beta}^n\ (\alpha^{(i)}\circ \ell))_{\mathcal{A}}(x)|i\in N\}$, where x is in $[X]^v$, one can construct as in the proof of Lemma 4.1 an n-iteration $L'=\{\text{beta}^n\ (\alpha^{(i)}\circ \ell)(y)|i\in N\}$ in $n\text{-}RT_\Sigma(Y)$ such that $f^{\#}(L')=L$. Let $t=\bigsqcup L'$, so that t is in $n\text{-}RT_\Sigma(Y)$. Then, $f^{\#}(t)=f^{\#}(\bigsqcup L')=\bigsqcup f^{\#}(L')=\bigsqcup L$. Hence, $\bigsqcup L$ is in Z_s, so $E([X])_s$ is a subset of Z_s.

Conversely, we prove that Z_s is a subset of $E([X])_s$. If c is in Z_s, then $c = f^{\#}(t)$ for some t in $n\text{-}RT_\Sigma(Y)$. As above, there is an n-iteration L such that $t = \sqcup L$ and clearly, $f^{\#}(L)$ is an n-iteration in $E([X])_s$. Moreover, $c = f^{\#}(t) = f^{\#}(\sqcup L) = \sqcup f^{\#}(L)$. Hence, c is in $E([X])_s$, and Z_s is a subset of $E([X])_s$. Combining the two inclusions, we have $Z_s = E([X])_s$.

The following characterization of n-rational (weak) morphisms will be very useful. This extends the characterization given by Tiuryn from 0 to any $n \geqq 0$ [44], [45].

THEOREM 4.4. *Let $\mathcal{A}$ and $\mathcal{B}$ be n-rational Σ-algebras. An S-indexed family of functions $h: \mathcal{A} \to \mathcal{B}$ is a weak n-rational morphism if and only if, for every n-rational tree t in $n\text{-}RT_\Sigma(v, s)$, for every c in A^v, $h_s(t_{\mathcal{A}}(c)) = t_{\mathcal{B}}(h^v(c))$. It is a monotonic n-rational morphism if h is also monotonic.*

Proof. First, assume $h: \mathcal{A} \to \mathcal{B}$ is a (weak) n-rational morphism. Let t be in $n\text{-}RT_\Sigma(v, s)$ and let c be in A^v. Define $d: X_v \to \mathcal{B}$ by $d(x_i) = h(c_i)$, $1 \leqq i \leqq |v|$ (c in A^v is also interpreted as a function $c: X_v \to \mathcal{A}$). Then, $h(t_{\mathcal{A}}(c)) = h(c^{\#}(t)) = c^{\#} \cdot h(t)$. Since the restriction of the n-rational morphism $c^{\#} \cdot h$ to X_v is equal to d, we have $d^{\#} = c^{\#} \cdot h$ and so

$$h(t_{\mathcal{A}}(c)) = c^{\#} \cdot h(t) = d^{\#}(t) = t_{\mathcal{B}}(h^v(c)).$$

Next, assume that the condition of the theorem holds. Since it holds in particular for finite trees, h is a homomorphism. Also, substituting $\perp$ for t, h is strict. We show that for every n-iteration $E = \{(\text{beta}^n\ (a^{(i)} \circ b))_{\mathcal{A}}(c) | i \in N\}$, $h(\sqcup E) = \sqcup h(E)$.

Let $f: X_v \to \mathcal{A}$ be the function such that $f(x_i) = c_i$ $(1 \leqq i \leqq |v|)$ and let $x = (x_1, \cdots, x_{|v|})$. It is clear that $E' = \{\text{beta}^n\ (a^{(i)} \circ b)(x) | i \in N\}$ is an n-iteration in $n\text{-}RT_\Sigma(X_v)$ and that $f^{\#}(E') = E$. Define $g: X_v \to \mathcal{B}$ by $g(x_i) = h(c_i)$. Note that $f^{\#} \cdot h$ is a strict homomorphism and that the restrictions of $f^{\#} \cdot h$ and $g^{\#}$ to X_v are equal, and therefore their restrictions to $FT_\Sigma(X_v)$ are equal. Hence, $g^{\#}(E') = h(f^{\#}(E')) = h(E)$. Let $t = \sqcup E'$ in $n\text{-}RT_\Sigma(X_v)$. We have, $h(\sqcup E) = h(\sqcup f^{\#}(E')) = h(f^{\#}(\sqcup E')) = h(f^{\#}(t)) = h(t_{\mathcal{A}}(c)) = t_{\mathcal{B}}(h^v(c)) = g^{\#}(t) = g^{\#}(\sqcup E') = \sqcup g^{\#}(E') = \sqcup h(E)$.

This concludes the proof that h is n-rationally continuous. Note also that the monotonicity of h was not used, and that if h is monotonic, h is a monotonic n-rational morphism.

The category $n\text{-}R\text{-}\mathrm{ALG}_\Sigma$ has all n-rational Σ-algebras for objects, and all monotonic n-rational morphisms as morphisms.

5. *n*-rational derived operators. If $\mathcal{A}$ is an n-rational Σ-algebra, we can define the ordered $D\Sigma$-algebra of "n-rational derived operators", denoted as $r_n\mathcal{A}$. Then, it can be shown that this algebra is $(n-1)$-rational and that these operators are n-rationally continuous. Continuing in this fashion, we obtain an ordered algebra of level-n functionals $r_1 r_2 \cdots r_{n-1} r_n \mathcal{A}$, denoted for simplicity as $d_n\mathcal{A}$, which is 0-rational. In the terminology of Tiuryn, $d_n\mathcal{A}$ is a regular algebra [44], [45].

This is a significant result, since this implies that *every* 0-rational scheme over $D^n\Sigma$ has a fixed-point semantics when interpreted in $d_n\mathcal{A}$. In particular, schemes of type $b_{n+1}(v, s)$ have a well defined meaning in $d_n\mathcal{A}$.

DEFINITION 5.1. Every tree t in $n\text{-}RT_\Sigma(v, s)$ defines a function $t_{\mathcal{A}}: A^v \to A_s$ such that for every c in A^v, $t_{\mathcal{A}}(c) = c^{\#}(t)$, where $c^{\#}: n\text{-}RT_\Sigma(X_v) \to \mathcal{A}$ is the unique n-rational morphism extending the function $c: X_v \to \mathcal{A}$. Functions of the form $t_{\mathcal{A}}$ are called *n-rational derived operators.*

Note that if $t = \sqcup\{\text{beta}^n\ (a^{(i)} \circ b) | i \in N\}$, $t_{\mathcal{A}}(c) = c^{\#}(t) = \sqcup c^{\#}(\text{beta}^n\ (a^{(i)} \circ b)) = \sqcup(\text{beta}^n\ (a^{(i)} \circ b))_{\mathcal{A}}(c)$, by the definition of derived operators and since $c^{\#}$ is n-rationally continuous.

THEOREM 5.2. *For every n-rational Σ-algebra $\mathscr{A}$, every n-rational derived operator $t_{\mathscr{A}}$ is monotonic and n-rationally continuous.*

Proof. From the above remark, for every c in A^v, $t_{\mathscr{A}}(c)=\bigsqcup(\text{beta}^n\,(a^{(i)}\circ b))_{\mathscr{A}}(c)$. Since each $t_i=\text{beta}^n\,(a^{(i)}\circ b)$ is a finite tree, each $(t_i)_{\mathscr{A}}$ is monotonic, and thus $t_{\mathscr{A}}$ is also monotonic.

For any m-tuple of n-iterations $(E_1,\cdots,E_m)$, with $E_j=\{a_i^j|i\in N\}$, we have (with $m=|v|$):

$$t_{\mathscr{A}}(\bigsqcup a_i^1,\cdots,\bigsqcup a_i^m)=\bigsqcup_k(\text{beta}^n\,(a^{(k)}\circ b))_{\mathscr{A}}(\bigsqcup_i a_i^1,\cdots,\bigsqcup_i a_i^m)$$

$$=\bigsqcup_k\bigsqcup_i(\text{beta}^n\,(a^{(k)}\circ b))_{\mathscr{A}}(a_i^1,\cdots,a_i^m)$$

(because each derived operator induced by a finite tree is n-rationally continuous by Lemma 3.6)

$$=\bigsqcup_i\bigsqcup_k(\text{beta}^n\,(a^{(k)}\circ b))_{\mathscr{A}}(a_i^1,\cdots,a_i^m)$$

(because least upper bounds of chains commute)

$$=\bigsqcup_i t_{\mathscr{A}}(a_i^1,\cdots,a_i^m).$$

We now define the algebra $r_n\mathscr{A}$ of n-rational derived operators.

DEFINITION 5.3. Given an n-rational Σ-algebra $\mathscr{A}$, let $r_n\mathscr{A}_{v,s}$ be the set of all n-rational derived operators $t_{\mathscr{A}}:A^v\to A_s$, for t in $n\text{-}RT_\Sigma(v,s)$. Let $r_n\mathscr{A}$ be the family $\{r_n\mathscr{A}_{v,s}|(v,s)\in S^*\times S\}$.

We shall show that $r_n\mathscr{A}$ is in fact a $D\Sigma$-algebra under functional composition, and that it is $(n-1)$-rational. This means that every $(n-1)$-rational scheme of type $b_{DS,n}(\bar{v},\bar{s})$ (where $(\bar{v},\bar{s})$ is in D^2S) has a least fixed-point when interpreted in $r_n\mathscr{A}$. Such schemes allow variables ranging over functionals (of type 1). First, we prove that $r_n(n\text{-}RT_\Sigma)$ is the subalgebra of the tree-substitution algebra DCT_Σ consisting of the n-rational trees with variables (see Definition 2.17), and that it is $(n-1)$-rational.

LEMMA 5.4. *$r_n(n\text{-}RT_\Sigma)$ is isomorphic to the tree-substitution $D\Sigma$-algebra of n-rational trees with variables, and it is an $(n-1)$-rational $D\Sigma$-algebra.*

Proof. First, we show that $r_n(n\text{-}RT_\Sigma)$ is isomorphic to the tree-substitution $D\Sigma$-algebra of n-rational trees with variables. For this, we show that there is a bijection $D_{v,s}$ between $n\text{-}RT_\Sigma(v,s)$ and $r_n(n\text{-}RT_\Sigma)_{(v,s)}$. Each derived operator h of type (v,s) in $r_n(n\text{-}RT_\Sigma)$ is defined by a tree T in $n\text{-}RT_\Sigma(v,s)$, and for every $t=(t_1,\cdots,t_m)$ with each t_j in $n\text{-}RT_\Sigma(u,v_j)$, $h(t_1,\cdots,t_m)=t^{\#}(T)$, where $t^{\#}$ is the n-rational extension of t viewed as a function $t:X_v\to n\text{-}RT_\Sigma$. But $t^{\#}(T)$ is equal to $(t_1,\cdots,t_m)\circ T$, the result of composing t and T. Hence, we have a surjective function

$$D_{v,s}:n\text{-}RT_\Sigma(v,s)\to r_n(n\text{-}RT_\Sigma)_{(v,s)},$$

where $D_{v,s}(T)$ is the derived operator such that

$$D_{v,s}(T)(t_1,\cdots,t_m)=(t_1,\cdots,t_m)\circ T.$$

CLAIM 1. *The function $D_{v,s}$ is also injective.*

Proof. First, the following notation is introduced for subtrees. Given a tree T and a node u in dom (T), the subtree T/u rooted at u in T is the tree defined by the function whose graph is

$$\{(v,T(uv))|uv\in\text{dom}\,(T)\}.$$

Next, assume that for two trees T and T', $T \neq T'$ but $D_{v,s}(T) = D_{v,s}(T')$. Since $(t_1, \cdots, t_m) \circ T = (t_1, \cdots, t_m) \circ T'$ for all $t_1, \cdots, t_m$ (with each t_j in $n\text{-}RT_\Sigma(e, v_j)$), $T \neq T'$ implies that there is some tree address u both in dom (T) and dom (T') such that either $T(u)$ or $T'(u)$ is a variable, say x_i, and $T/u \neq T'/u$. Without loss of generality, we can assume that $T(u) = x_i$. Since we are assuming that ranked alphabets are nontrivial (remark after Definition 2.4) there exists a finite tree t distinct from $\perp$ in $FT_\Sigma^{v_i}$. Three cases arise.

1. $T'(u) = x_j$ with $i \neq j$. Then, letting $t_k = \perp$ for $k \neq i$ and $t_i = t$, $(t_1, \cdots, t_m) \circ T \neq (t_1, \cdots, t_m) \circ T'$, a contradiction.

2. $T'(u) = \perp$. As in case 1, a contradiction is reached using the tuple $(t_1, \cdots, t_m)$ defined there.

3. $T'(u) \neq x_i$ and $T'(u) \neq \perp$. Then, letting $t_k = \perp$ for all k, $1 \leqq k \leqq m$, $(t_1, \cdots, t_m) \circ T \neq (t_1, \cdots, t_m) \circ T'$, a contradiction. Hence, the family of functions $D = (D_{v,s})$ is a family of bijections.

Note that $((n\text{-}RT_\Sigma)_{(u,s)})_{(u,s) \in DS}$ is an ordered $D\Sigma$-algebra, interpreting each $C_{u,v,s}$ as tree composition, each π_i^u as x_i^u and $\text{abst}_{v,s}$ as the inclusion function from $n\text{-}RT_\Sigma(e, s)$ to $n\text{-}RT_\Sigma(v, s)$. Also observe that

$$
\begin{aligned}
&D_{v,s}((T_1, \cdots, T_n) \circ T)(t_1, \cdots, t_m) \\
&\quad = (t_1, \cdots, t_m) \circ ((T_1, \cdots, T_n) \circ T) \quad \text{(by definition of } D_{v,s}) \\
&\quad = ((t_1, \cdots, t_m) \circ (T_1, \cdots, T_n)) \circ T \quad \text{(by associativity of tree composition)} \\
&\quad = ((t_1, \cdots, t_m) \circ T_1, \cdots, (t_1, \cdots, t_m) \circ T_n) \circ T \quad \text{(by Definition 2.5)} \\
&\quad = ((D_{v,w_1}(T_1), \cdots, D_{v,w_n}(T_n)) \cdot D_{w,s}(T))(t_1, \cdots, t_m) \quad \text{(by definition of the } D_{v,s}).
\end{aligned}
$$

This proves that $(D_{v,s})_{(v,s) \in DS}$ is a homomorphism, and since it is bijective, it is an isomorphism between the tree-substitution algebra of n-rational trees and $r_n(n\text{-}RT_\Sigma)$.

Since tree composition is ω-continuous, tree-substitution is $(n-1)$-rationally continuous.

It remains to show that $r_n(n\text{-}RT_\Sigma)$ is $(n-1)$-rationally complete. As in § 2 (Definition 2.22), for $n \geqq 1$, we define BETA^{n-1} as the family of functions

$$\text{BETA}^{n-1} = (\text{BETA}_{u,s}^{n-1} : CT_{D^n\Sigma}(e, b_n(u, s)) \to (D^2CT_\Sigma)_{(e,(u,s))} | (u, s) \in DS).$$

Then, we have $\text{beta}^n = \text{BETA}^{n-1} \cdot \text{beta}$.

CLAIM 2. *For all $n \geqq 1$, for every tree t in $0\text{-}RT_{D^n\Sigma}$ of sort $b_n(v, s)$,*

$$(\text{BETA}^{n-1}(t))_{r_n(n\text{-}RT_\Sigma)} = \text{beta}^n\ (t).$$

Proof. Let $T = \text{BETA}^{n-1}(t)$. It is sufficient to show that for every tree T in $(n-1)\text{-}RT_{D\Sigma}(e, (v, s))$, $T_{r_n(n\text{-}RT_\Sigma)} = \text{beta}\ (T)$. By the definition of a derived operator (Definition 2.12) and since T has no variables, $T_{r_n(n\text{-}RT_\Sigma)} = h^\#(T)$, where $h^\# : (n-1)\text{-}RT_{D\Sigma} \to r_n(n\text{-}RT_\Sigma)$ is the unique homomorphism given by initiality of $(n-1)\text{-}RT_{D\Sigma}$. But $(n-1)\text{-}RT_{D\Sigma}$ is the subalgebra of $CT_{D\Sigma}$ consisting of the $(n-1)$-rational $D\Sigma$-trees, and $r_n(n\text{-}RT_\Sigma)$ is the subalgebra of DCT_Σ consisting of the n-rational Σ-trees. Hence, by Definition 2.20, $h^\#$ is the restriction of beta to $(n-1)\text{-}RT_{D\Sigma}$, and $h^\#(T) = \text{beta}\ (T)$.

Given a scheme $(\mathscr{a}, \mathscr{b})$ over $D^n\Sigma$ of type $b_{DS,n}(\bar{v}, \bar{s})$ with $\bar{v} = (v_1, s_1) \cdots (v_m, s_m)$, and $\bar{s} = (w, s)$, we have to show that for every tuple $(t_1, \cdots, t_m)$ of trees with each t_j in $n\text{-}RT_\Sigma(v_j, s_j)$, the $(n-1)$-iteration $\{(\text{BETA}^{n-1}(\mathscr{a}^{(i)} \circ \mathscr{b}))_{r_n(n\text{-}RT_\Sigma)}(t_1, \cdots, t_m) | i \in N\}$ has a least upper bound in $n\text{-}RT_\Sigma(w, s)$. Assume that each tree t_j is defined by an

n-rational scheme (a_j, b_j). Then,

$$(\mathrm{BETA}^{n-1}(a^{(i)} \circ b))_{r_n(n\text{-}RT_\Sigma)}(t_1, \cdots, t_m)$$

$$= (\mathrm{BETA}^{n-1}(a^{(i)} \circ b))_{r_n(n\text{-}RT_\Sigma)}(\mathrm{beta}^n(a_1^{(i)} \circ b_1), \cdots, \mathrm{beta}^n(a_m^{(i)} \circ b_m))$$

(by definition of the t_j)

$$= (\mathrm{beta}^n(a_1^{(i)} \circ b_1), \cdots, \mathrm{beta}^n(a_m^{(i)} \circ b_m)) \circ \mathrm{beta}^n(a^{(i)} \circ b)$$

(by Claim 2 and the fact that $r_n(n\text{-}RT_\Sigma)$ is a tree-substitution algebra). We conclude exactly as in the proof of Lemma 3.8 using the claim established there, and by constructing a scheme (g, d), such that

$$(\mathrm{BETA}^{n-1}(g^{(i)} \circ d))_{r_n(n\text{-}RT_\Sigma)} = \mathrm{beta}^n(g^{(i)} \circ d)$$

$$= (\mathrm{beta}^n(a_1^{(i)} \circ b_1), \cdots, \mathrm{beta}^n(a_m^{(i)} \circ b_m)) \circ \mathrm{beta}^n(a^{(i)} \circ b).$$

By continuity of tree composition, the result is established. This concludes the proof of Lemma 5.4.

Given an n-rational Σ-algebra $\mathscr{A}$, we define the function $r_n: r_n(n\text{-}RT_\Sigma) \to r_n\mathscr{A}$ such that for every t in $n\text{-}RT_\Sigma(v, s)$, $r_n(t) = t_{\mathscr{A}}$, the n-rational derived operator defined by t. The function r_n is clearly surjective. Using standard techniques (see Goguen, Thatcher, Wagner and Wright [3]), we can show that for all t in $n\text{-}RT_\Sigma(v, s)$ and $t' = (t'_1, \cdots, t'_m)$ with each t'_j in $n\text{-}RT_\Sigma(u, v_j)$, $m = |v|$, $(t' \circ t)_{\mathscr{A}} = t'_{\mathscr{A}} \cdot t_{\mathscr{A}}$. Hence, $r_n\mathscr{A}$ is an ordered $D\Sigma$-algebra, and r_n is in fact a homomorphism. We show that r_n is ω-continuous.

LEMMA 5.5. *The function $r_n: r_n(n\text{-}RT_\Sigma) \to r_n\mathscr{A}$ is an ω-continuous, strict, surjective homomorphism of $D\Sigma$-algebras.*

Proof. Let $\{t_j | j \in N\}$ be a chain in $n\text{-}RT_\Sigma(u, s)$ with least upper bound t. For every $c: X_u \to \mathscr{A}$,

$$r_n(t)(c) = t_{\mathscr{A}}(c) = c^*(t) = c^*(\bigsqcup t_i)$$

$$= \bigsqcup c^*(t_i) \quad (\text{since } c^* \text{ is } \omega\text{-continuous by Theorem 3.10})$$

$$= \bigsqcup (t_i)_{\mathscr{A}}(c).$$

Hence, since least upper bounds are obtained pointwise, $r_n(t)$ is an upper bound for $\{r_n(t_i) | i \in N\}$ in $r_n\mathscr{A}_{u,s}$. If g is any other upper bound, for every c as above, we have $r_n(t_i)(c) \leqq g(c)$ for all $i \geqq 0$. Since $r_n(t)(c) = \bigsqcup r_n(t_i)(c)$, we have $r_n(t)(c) \leqq g(c)$ for all c, and this shows that $r_n(t) \leqq g$ and the ω-continuity of r_n.

The other conditions of the lemma are easily verified.

LEMMA 5.6. *If $\mathscr{A}$ is an n-rational Σ-algebra, $\mathscr{B}$ is an ordered Σ-algebra and there is a surjective strict ω-continuous homomorphism $h: \mathscr{A} \to \mathscr{B}$, then $\mathscr{B}$ is also n-rational.*

Proof. The proof parallels that of Lemma 4.1 and is left to the reader.

Hence, we have the following result.

THEOREM 5.7. *For any n-rational Σ-algebra $\mathscr{A}$, the $D\Sigma$-algebra of n-rational derived operators $r_n\mathscr{A}$ is an $(n-1)$-rational $D\Sigma$-algebra.*

Proof. By Lemma 5.5 and 5.6.

Starting with an n-rational algebra $\mathscr{A}$ and applying the above theorem n times, we obtain the algebra $r_1 r_2 \cdots r_{n-1} r_n \mathscr{A}$ denoted for simplicity as $d_n\mathscr{A}$, which is 0-rational. For every 0-rational algebra, using 0-rationality and 0-continuity of parameterized derived operators given by Lemma 3.6, one can easily establish the following lemma.

LEMMA 5.8. *Given a* 0*-rational* Σ*-algebra* $\mathcal{A}$, *for every* (0*-rational*) *parameterized scheme* (a, b) *of type* (v, s), $a^+_{\mathcal{A}}$ *is the least fixed-point of the functional equation* $h = (I_v, h) \cdot a_{\mathcal{A}}$, *where* $a^+_{\mathcal{A}} = (\bigsqcup a^{(i)}_{\mathcal{A}}) \cdot \pi^{vu}_u$.

Then, since $d_n\mathcal{A}$ is 0-rational, every parameterized scheme of type $(\bar{v}, \bar{s})$ *in* $D^n\Sigma$ has a least fixed-point in $d_n\mathcal{A}$. In particular, if the type of the scheme is $b_{n+1}(v, s)$ with (v, s) in DS, this least fixed-point is in fact a function from A^v to A_s.

Note that $d_n\mathcal{A}$ consists of level n functionals, and that $d_n\mathcal{A}$ is closed under fixed-points of parameterized rational schemes. We also mention that $r_n\mathcal{A}$ and $d_n\mathcal{A}$ are *clones* satisfying closure conditions related to those satisfied by the "mu-clones" of Wand [54], [55].

6. Comments and questions on part I. A number of questions whose answer is unknown to the author are listed below and a number of comments regarding the above developments are made.

(i) Tiuryn developed a theory of regular algebras [44], [45] using a notion more general than that of an ordered algebra, namely the notion of a strict algebra. Can this be generalized to n-rational algebras?

(ii) The author has studied in [19] infinite trees arising from unfolding recursion schemes, by encoding their paths using "address languages." Given an infinite tree t using only finitely many different function symbols, for every such symbol f, define the *address language* $L(f) = \{u | u \in \mathrm{dom}\,(t),\ t(u) = f\}$ as the set of all tree addresses labeled with an occurrence of f in t. Then, t is completely determined by the finite collection of languages of the form $L(f)$. Ginali [22], [23] proved that these languages are regular for infinite trees arising from 0-rational schemes. Gallier [19] proved that these languages are deterministic context-free for trees arising from recursion schemes = 1-rational schemes.

Is there a characterization of these languages for $n \geqq 2$?

Conjecture. These languages are deterministic OI indexed languages for $n = 2$.

(iii) The author also conjectures that for all $n \geqq 0$, these languages are *primitive recursive.* Combined with the hierarchy result of Theorem 3.11, this last conjecture opens the possibility of extending the definition of derived alphabets to countably infinite ordinals. The author believes that this is possible, and that the hierarchy is strict. This would mean that there is a least ordinal such that the trees obtained by unfolding schemes of transfinite (countable) types and applying beta$^\alpha$ (where α is a countable ordinal) are not recursively enumerable. What would this ordinal be? In such an extension, it appears that a certain form of *type polymorphism* is naturally introduced.

(iv) Address languages also suggest another way of defining classes of infinite trees. For example, a deterministic context-free tree can be defined by requiring each $L(f)$ to be deterministic context-free. For regular trees, Ginali [22], [23] proved that these trees are in fact unfoldings of 0-rational schemes (see also Elgot, Bloom and Tindell [15]). However, the author does not know of a similar proof for the context-free case.

One can also investigate the infinite primitive recursive, recursive or recursively enumerable trees, by requiring each $L(f)$ to be primitive recursive, recursive or recursively enumerable. Such investigations remain to be done.

(v) What kinds of algebraic theories yield n-rational algebras as their algebras? Is there a relationship with the I-adic theories of Wagner [53] and Wagner, Wright and Thatcher [52] and the $M(T)$ construction?

Other problems related to the logic of inequalities are deferred to Part 2.

Acknowledgment. I wish to thank Eric Wagner for suggesting numerous improvements, both technical and stylistic, and for his encouragement.

REFERENCES

[1] J. A. GOGUEN, J. W. THATCHER, E. G. WAGNER AND J. B. WRIGHT, *An introduction to categories, algebraic theories and algebras*, IBM Rept. RC-5369, IBM T. J. Watson Lab., Yorktown Heights, New York, 1975.

[2] ———, *Rational algebraic theories and fixed-point solutions*, 17th IEEE Symposium on Foundations of Computer Science, Houston, TX, 1976, pp. 147–158.

[3] ———, *Initial algebra semantics and continuous algebras*, J. ACM, 24 (1977), pp. 68–95.

[4] S. BLOOM, *Varieties of ordered algebras*, J. Comput. System Sci., 13 (1976), pp. 200–212.

[5] P. M. COHN, *Universal Algebra*, Harper and Row, New York, 1965.

[6] B. COURCELLE, *A representation of trees by languages, Parts* 1, 2, Theoret. Comput. Sci., 6 and 7 (1978), pp. 25–55 and pp. 255–279.

[7] ———, *Infinite trees in normal form and recursive equations having a unique solution*, Math. Systems Theory, 13 (1979), pp. 131–180.

[8] B. COURCELLE AND I. GUESSARIAN, *On some classes of interpretations*, J. Comput. System Sci., 17 (1978), pp. 388–413.

[9] B. COURCELLE AND P. FRANCHI-ZANNETTACCI, *Attribute grammars and recursive program schemes*, Theoret. Comput. Sci., 17 (1982), pp. 163–192 and pp. 235–258.

[10] B. COURCELLE AND M. NIVAT, *Algebraic families of interpretations*, 17th IEEE Symposium on Foundations of Computer Science, Houston, TX, 1976, pp. 137–146.

[11] W. DAMM, *Languages defined by higher program schemes, Automata, Languages and Programming*, Lecture Notes in Computer Science 52, Springer-Verlag, New York, 1977.

[12] ———, *The* IO *and* OI *hierarchies*, Ph.D thesis, Rept. no. 41, RWTH Aachen, 1980.

[13] ———, *The* IO *and* OI *hierarchies*, Theoret. Comput. Sci., 20 (1982), pp. 95–206.

[14] C. C. ELGOT, *Monadic computation and iterative algebraic theories*, Logic Colloquium '73, Studies in Logic, Vol. 80, North-Holland, Amsterdam, 1975, pp. 175–230.

[15] C. C. ELGOT, S. BLOOM AND R. TINDELL, *On the algebraic structure of rooted trees*, J. Comput. System Sci., 16 (1978), pp. 362–399.

[16] J. ENGELFRIEDT AND E. SCHMIDT, IO *and* OI, *Part* I *and Part* II, J. Comput. System Sci., 15 (1977) and 16 (1978), pp. 328–353 and pp. 67–99.

[17] S. FEFERMAN, *Theories of finite type related to mathematical practice*, Handbook of Mathematical Logic, Studies in Logic, Vol. 90, North-Holland, Amsterdam 1977.

[18] E. P. FRIEDMAN, *The inclusion problem for simple languages*, Theoret. Comput. Sci., 1 (1976), pp. 297–316.

[19] J. H. GALLIER, DPDA*'s in 'atomic normal form' and applications to equivalence problems*, Theoret. Comput. Sci., 14 (1981), pp. 155–186.

[20] ———, *Recursion schemes and generalized interpretations*, Automata, Languages and Programming, Lecture Notes in Computer Science 71, Springer-Verlag, New York, 1979.

[21] ———, *Recursion-closed algebraic theories*, J. Comput. System Sci., 23 (1981), pp. 69–105.

[22] S. GINALI, *Iterative algebraic theories, infinite trees and program schemata*, Ph.D thesis, Univ. Chicago, Chicago, 1976.

[23] ———, *Regular trees and the free iterative theory*, J. Comput. System Sci., 18 (1979), pp. 228–242.

[24] S. GORN, *Explicit definitions and linguistic dominoes*, in Systems and Computer Science, J. Hart and S. Takasu, eds., 1975.

[25] G. GRATZER, *Universal Algebra*, Van Nostrand-Reinhold, New York, 1967.

[26] I. GUESSARIAN, *Schemas de programmes recursifs polyadiques: equivalence semantique et classes d'interpretations*, Thèse d'état, Université de Paris VII, 1975.

[27] ———, *Algebraic Semantics*, Lecture Notes in Computer Sciences, 99, Springer-Verlag, New York, 1981.

[28] J. R. HINDLEY, B. LERCHER AND J. P. SELDIN, *Introduction to Combinatory Logic*, Lecture Notes Series, No. 7, Cambridge Univ. Press, Cambridge, 1972.

[29] D. LEHMANN, *On the algebra of order*, J. Comput. System Sci., 21 (1980), pp. 1–23.

[30] T. S. E. MAIBAUM, *A generalized approach to formal languages*, J. Comput. System Sci., 8 (1974), pp. 409–439.

[31] J. MESEGUER, *Varieties of chain-complete algebras*, J. of Pure Appl. Algebra, 19 (1980), pp. 347–383.

[32] J. MEZEI AND J. B. WRIGHT, *Algebraic automata and context-free sets*, Inform. and Control, 11 (1967), pp. 3–29.

[33] M. NIVAT, *On the interpretation of recursive polyadic program schemes*, Symposia Mathematica, Vol. 15, Academic Press, New York, 1975, pp. 255–281.

[34] B. ROSEN, *Tree-manipulating systems and Church-Rosser theorems*, J. ACM, 20 (1973), pp. 160–187.

[35] K. SCHUTTE, *Proof Theory*, Springer-Verlag, New York, 1977.

[36] H. SCHWICHTENBERG, *Proof theory: some applications of cut-elimination*, Handbook of Mathematical Logic, Studies in Logic, Vol. 80, North-Holland, Amsterdam, 1977.

[37] D. SCOTT, *Outline of a Mathematical Theory of Computation*, Tech. Monograph PRG-2, Oxford Univ. Computing Laboratory, Programming Research Group, 1970.

[38] ———, *The Lattice of Flow Diagrams*, Tech. Monograph PRG-3, Oxford Univ. Computing Laboratory, Programming Research Group, 1971.

[39] ———, *Continuous Lattices*, Tech. Monograph PRG-7, Oxford Univ. Computing Laboratory, Programming Research Group, 1971.

[40] ———, *Data types as lattices*, this Journal, 5 (1976), pp. 522–587.

[41] J. W. THATCHER, E. G. WAGNER AND J. B. WRIGHT, *Programming languages as mathematical objects*, in Mathematical Foundations of Computer Science '78, Lecture Notes in Computer Science, 64, Springer-Verlag, New York, 1978.

[42] ———, *Free continuous theories*, Tech. Rept. RC-6906, IBM T. J. Watson Lab., Yorktown Heights, New York, 1977.

[43] ———, *Notes on algebraic fundamentals for theoretical computer science*, IBM Tech. Rept., 1979.

[44] J. TIURYN, *Fixed-points and algebras with infinitely long expressions, Part* I: *Regular algebras*, Mathematical Foundations of Computer Science '77, Lecture Notes on Computer Science 53, Springer-Verlag, New York, 1977.

[45] ———, *Fixed-points and algebras with infinitely long expressions, Part* II: *mu-clones of regular algebras*, Foundations of Computing Theory, Lecture Notes in Computer Science 56, Springer Verlag, New York, 1977.

[46] ———, *On a connection between regular algebras and rational algebraic theories*, 2nd Workshop on Categorical and Algebraic Methods in Computer Science and System Theory, Dortmund, 1978.

[47] ———, *Fixed points in the power-set algebra of infinite trees*, Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 74, Springer-Verlag, New York 1979.

[48] ———, *Unique fixed points vs. least fixed points*, Theoret. Comput. Sci., 12 (1980), pp. 229–254.

[49] E. G. WAGNER, *Languages for defining sets in arbitrary algebras*, 12th IEEE Symposium on Foundations of Computer Science, East Lansing, MI, 1971.

[50] ———, *From Algebras to Programming Languages*, 5th Annual ACM S.I.G.A.C.T. Symposium, Austin, TX, 1973.

[51] ———, *An algebraic theory of recursive definitions and recursive languages*, 3rd Annual ACM S.I.G.A.C.T. Symposium, Shaker Heights, OH, 1971.

[52] E. G. WAGNER, J. B. WRIGHT AND J. W. THATCHER, *Algebraic theories, I-adic theories and program semantics*, IBM Research Tech. Rept., IBM T. J. Watson Lab., Yorktown Heights, NY, 1981.

[53] E. G. WAGNER, *Functorial hierarchies of functional languages*, IBM Research Tech. Rept. RC 9327, IBM T. J. Watson Lab., Yorktown Heights, New York, 1982.

[54] M. WAND, *A concrete approach to abstract recursive definitions*, Automata, Languages and Programming, North-Holland, Amsterdam, 1973, pp. 331–341.

[55] ———, *Mathematical foundations of formal language theory*, PhD thesis, Massachusetts Institute of Technology, Cambridge, Project MAC Rept., MAC TR-108, 1973.

SIAM J. COMPUT.
Vol. 13, No. 4, November 1984
© 1984 Society for Industrial and Applied Mathematics
008

n-RATIONAL ALGEBRAS II. VARIETIES AND LOGIC OF INEQUALITIES*

JEAN H. GALLIER†

Abstract. This work is a continuation of the study of n-rational algebras initiated in the first part of this paper [SIAM J. Comput., 13 (1984), pp. 750–775]. In this second part, varieties of n-rational algebras satisfying a set of inequalities and the corresponding logic are investigated. It is shown that there is a bijective Galois connection between such varieties, called semi-varieties, and "fully invariant" n-rational precongruences. A deductive system for proving inequalities in which a proof is represented as a well-founded tree is shown to be sound and complete. This proof system uses one infinitary inference rule, the "lub rule". A "Birkhoff variety theorem" is also proved for semi-varieties. The relationship between this approach in which classes of interpretations are semi-varieties of n-rational algebras, and the approach in which ω-continuous algebras are used (Courcelle, Nivat, Guessarian) is also briefly explored.

Key words. algebraic semantics, varieties of algebras, recursion schemes, logic of inequalities, infinite trees, well-founded proof trees, classes of interpretations

7. Introduction. This is a continuation of the study of n-rational algebras started in Part I of this paper (this issue, pp. 750–755). In Part I, basic properties of n-rational algebras were investigated. In particular, the existence of free algebras was proved. In Part II, varieties of n-rational algebras satisfying a set of inequalities and the associated logic are investigated.

The main reason for restricting our attention to inequalities is that inequalities are the simplest kind of axioms having reasonably simple and yet interesting mathematical properties. In particular, properties of programs such as extension or equivalence can be expressed as inequalities between infinite trees. This is because infinite trees in the free n-rational algebra $n\text{-}RT_\Sigma(X_0)$ (where X_0 denotes a family of countable sets of variables) represent the result of unfolding recursion schemes. Hence, the validity of the inequality $t_1 \leqq t_2$ in the variety defined by the set of inequalities E expresses the property "t_2 extends t_1," for the class of programs defined by t_1 and t_2 in this variety.

Unfortunately, even for $n=1$, the set of inequalities valid in $1\text{-}RT_\Sigma(X_0)$ is not recursively enumerable. This follows from two facts. First, it is partially decidable whether $t_1 \neq t_2$ for two trees t_1 and t_2 in $1\text{-}RT_\Sigma(X_0)$. Second, as was shown in Courcelle [7], it is undecidable whether $t_1 \leqq t_2$ for recursion schemes. This follows from the undecidability of the inclusion problem for simple languages, see Friedman [18]. Hence, there is no hope for a recursive axiomatization. However, it is shown in Theorem 10.19 that there is a complete proof system with one infinitary inference rule. In this system, a proof is represented as a *well-founded* tree. The main results of this paper are:

(1) The characterization of inequational varieties in terms of the bijective Galois connection with "fully invariant" precongruences (Theorem 10.14).

(2) The completeness theorem for the logic of inequalities (Theorem 10.19).

(3) The Birkhoff variety theorem for inequational varieties (Theorem 11.2).

(4) The "algebraicity" of inequational varieties when the inequalities are between finite trees (Theorem 12.2).

In order to help the reader understand the motivation behind the study of n-rational algebras, some of the advantages of n-rational algebras over ω-continuous algebras

* Received by the editors June 24, 1981, and in final revised form June 21, 1983. This research was partially supported by the National Science Foundation under grant MCS-8111726.

† Department of Computer and Information Sciences, Moore School of Electrical Engineering D2, University of Pennsylvania, Philadelphia, Pennsylvania 19104.

(mentioned in Part I) are repeated below. The main advantage is that no completion operation is necessary. The constructions used in dealing with n-rational algebras are quite similar to those used with ordinary (unordered and ordered) algebras (e.g., constructions and proofs can be done with congruences and orderings), while in the ω-continuous case, these constructions do not work and one must use completions to get the corresponding (or nearest corresponding) result. In particular, two important properties which fail for ω-continuous algebras but hold for n-rational algebras are the following:

(1) The image of an n-rational algebra under an n-rational morphism is an n-rational algebra.

(2) Given a set E of inequalities and the least congruence $\cong$ induced by E, the Herbrand n-rational algebra of the variety defined by E is the quotient $n\text{-}RT_\Sigma(X_0)/\cong$.

An outline of Part II follows. Section 8 is devoted to precongruences and quotients of n-rational algebras. In § 9, presentations of n-rational algebras are defined and studied. The least precongruence containing a set of inequalities is characterized in terms of a deductive system using well-founded proof trees. Section 10 is devoted to a study of inequational varieties, called semi-varieties. The main results are the bijective Galois connection between semi-varieties and fully invariant precongruences, the completeness theorem, and the characterization of universal algebras. In § 11, the Birkhoff variety theorem for semi-varieties is proved: A class of n-rational algebras is a semi-variety if and only if it is closed under subalgebras, direct products and images under (monotonic) n-rational morphisms. Section 12 covers the case where the inequalities consist of finite trees. It is shown that every valid inequality has a proof of height at most ω. A brief comparison is made with the approach using ω-continuous algebras (Courcelle [7], Courcelle and Guessarian [8], Courcelle and Nivat [10] and Guessarian [26] and [27]).

8. Precongruences and quotients. In this section the concept of an n-rational precongruence is introduced. A precongruence is a preorder satisfying additional properties, so that the quotient of an n-rational Σ-algebra by the equivalence induced by the precongruence is also n-rational.

Recall from § 2 of Part I that a preorder is a reflexive and transitive relation.

In order to avoid notational ambiguities, the partial ordering on each carrier of an algebra or on trees will be denoted by $\sqsubset$.

DEFINITION 8.1. Given an ordered Σ-algebra $\mathscr{A}$ and an S-indexed family $\leqq$ of relations $\leqq_s$ on each carrier A_s, $\leqq$ is an *n-rational precongruence*, for short *a* precongruence, if the following conditions hold:

(1) Each $\leqq_s$ is a preorder.

(2) For all x, y in A_s, $x \sqsubset y$ implies $x \leqq y$.

(3) For every operation $f_{\mathscr{A}}: A^u \to A_s$, for all $(x_1, \cdots, x_m)$ and $(y_1, \cdots, y_m)$ in A^u (with $m = |u|$), if $x_i \leqq y_i$ for $1 \leqq i \leqq m$, then $f_{\mathscr{A}}(x_1, \cdots, x_m) \leqq f_{\mathscr{A}}(y_1, \cdots, y_m)$.

(4) For every n-iteration $E = \{a_i | i \in N\}$ in A_s, for every b in A_s, if $a_i \leqq b$ for all $i \in N$, then $\bigsqcup a_i \leqq b$.

A preorder satisfying condition (2) is called *admissible*. Condition (4) is referred to as the *n-rational continuity* of the preorder. Condition (3) is the *congruential property*. A relation satisfying only conditions (1) and (3) and (4) is called a *quasi n-rational precongruence*.

Given a (quasi) precongruence $\leqq$, the (quasi) *congruence associated with* $\leqq$, denoted as $\cong$, is the largest equivalence relation contained in $\leqq$ and is defined as: $x \cong y$ if and only if $x \leqq y$ and $y \leqq x$. Given a (quasi) ccongruence $\cong$, the equivalence class of x modulo $\cong$ is denoted by $\bar{x}$.

DEFINITION 8.2. Given a morphism $h: \mathcal{A} \to \mathcal{B}$ of n-rational Σ-algebras, the S-indexed family $\leqq_h$ such that $(\leqq_h)_s = \{(x, y) | (x, y) \in A_s \times A_s, h_s(x) \sqsubseteq h_s(y)\}$ is a preorder called the *precongruence associated with h.* The associated equivalence Ker (h), where Ker $(h)_s = \{(x, y) | h_s(x) = h_s(y)\}$ is the *kernel* of h.

Recall from Part I, Definitions 2.9 and 3.7, that a weak n-rational morphism is a not necessarily monotonic homomorphism of ordered Σ-algebras which preserves n-iterations.

LEMMA 8.3. *Given a morphism $h: \mathcal{A} \to \mathcal{B}$ of n-rational Σ-algebras, the preorder $\leqq_h$ is an n-rational precongruence. If h is a weak n-rational morphism, $\leqq_h$ is a quasi n-rational precongruence.*

Proof. It is obvious that $\leqq_h$ is a preorder and is congruential since h is a homomorphism. If h is monotonic, then $x \sqsubseteq y$ implies $h(x) \sqsubseteq h(y)$ and so $\leqq_h$ is admissible. Let $E = \{a_i | i \in N\}$ be an n-iteration in A_s and assume that $a_i \leqq_h b$ for all $i \in N$, for some b in A_s. This means that $h(a_i) \sqsubseteq h(b)$ for all $i \in N$. Since h is n-rationally continuous, $h(\bigsqcup a_i) = \bigsqcup h(a_i)$. Hence, $h(\bigsqcup a_i) \sqsubseteq h(b)$, that is $\bigsqcup a_i \leqq_h b$, showing that $\leqq_h$ is n-rationally continuous.

DEFINITION 8.4. Given a morphism of n-rational Σ-algebras $h: \mathcal{A} \to \mathcal{B}$, the *quotient algebra* $\mathcal{A}/\text{Ker}(h)$ is defined as usual as the Σ-algebra whose carrier of sort s is the set $A_s/\text{Ker}(h)_s$ of equivalence classes modulo Ker $(h)_s$. For every function symbol f of type (u, s), f is interpreted in $\mathcal{A}/\text{Ker}(h)$ as the operation such that, for all $\bar{c}_1, \cdots, \bar{c}_m$ in $\mathcal{A}/\text{Ker}(h)$,

$$f_{\mathcal{A}/\text{Ker}(h)}(\bar{c}_1, \cdots, \bar{c}_m) = \overline{f_{\mathcal{A}}(c_1, \cdots, c_m)}.$$

Each carrier $A_s/\text{Ker}(h)_s$ is given the partial ordering $(\leqq_h)_s$ such that, for any two equivalence classes $\bar{a}$ and $\bar{b}$, $\bar{a}(\leqq_h)_s\bar{b}$ if and only if $a(\leqq_h)_s b$. By definition of Ker (h), this partial ordering is well defined. For simplicity, we often omit the subscript s, or even h.

LEMMA 8.5. *$\mathcal{A}/\text{Ker}(h)$ is an n-rational Σ-algebra and the natural mapping $\pi: \mathcal{A} \to \mathcal{A}/\text{Ker}(h)$ is an n-rational morphism. Furthermore, $\mathcal{A}/\text{Ker}(h)$ is isomorphic to $h(\mathcal{A})$.*

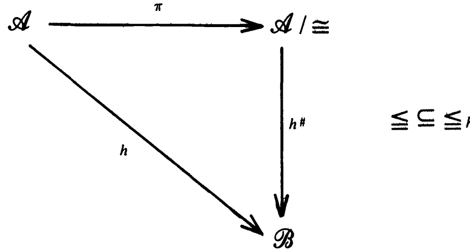
Proof. It is clear that $\mathcal{A}/\text{Ker}(h)$ is a (strict) ordered Σ-algebra. Let us show that it is n-rationally complete. First, observe that the following can easily be shown by induction on the depth of trees: For every finite tree t, for all $c_1, \cdots, c_m$ in $\mathcal{A}$, $\overline{t_{\mathcal{A}}(c_1, \cdots, c_m)} = t_{\mathcal{A}/\text{Ker}(h)}(\bar{c}_1, \cdots, \bar{c}_m)$. Let L be an n-iteration in $\mathcal{A}/\text{Ker}(h)$. Hence, $L = \{(\text{beta}^n(\alpha^{(i)} \circ \ell))_{\mathcal{A}/\text{Ker}(h)}(\bar{c}) | i \in N\}$. Using the above remark, $L = \bar{E}$ for the n-iteration $E = \{(\text{beta}^n(\alpha^{(i)} \circ \ell))_{\mathcal{A}}(c) | i \in N\}$. But E has a least upper bound $\bigsqcup E$ in $\mathcal{A}$, and by Lemma 8.3, the least upper bound of $L = \bar{E}$ is equal to $\overline{\bigsqcup E}$. Hence, $\mathcal{A}/\text{Ker}(h)$ is n-rationally complete. Similarly, using Lemma 8.3, it is easily shown that the operations in $\mathcal{A}/\text{Ker}(h)$ are n-rationally continuous. Now, if t is an n-rational tree, $t = \bigsqcup t_i$ for an n-iteration of finite trees and, by the above remark, for all c in A^u, $\overline{t_{\mathcal{A}}(c)} = \overline{\bigsqcup (t_i)_{\mathcal{A}}(c)} = \bigsqcup \overline{(t_i)_{\mathcal{A}}(c)} = \bigsqcup (t_i)_{\mathcal{A}/\text{Ker}(h)}(\bar{c}) = t_{\mathcal{A}/\text{Ker}(h)}(\bar{c})$. To show that π is n-rationally continuous, we use the characterization of Theorem 4.4. Let t be any n-rational tree in $n\text{-}RT_{\Sigma}(u, s)$ and let $c = (c_1, \cdots, c_m)$ be in A^u. Then, we have $\pi(t_{\mathcal{A}}(c_1, \cdots, c_m)) = \overline{t_{\mathcal{A}}(c_1, \cdots, c_m)} = t_{\mathcal{A}/\text{Ker}(h)}(\bar{c}_1, \cdots, \bar{c}_m) = t_{\mathcal{A}/\text{Ker}(h)}(\pi(c_1), \cdots, \pi(c_m))$. Hence π is n-rationally continuous. Also, if h is monotonic, $\leqq_h$ is admissible and π is clearly monotonic. Finally, define $f: \mathcal{A}/\text{Ker}(h) \to h(\mathcal{A})$ by $f(\bar{c}) = h(c)$. Then, we have $f(t_{\mathcal{A}/\text{Ker}(h)}(\bar{c}_1, \cdots, \bar{c}_m)) = f(\overline{t_{\mathcal{A}}(c_1, \cdots, c_m)}) = h(t_{\mathcal{A}}(c_1, \cdots, c_m)) = t_{\mathcal{B}}(h(c_1), \cdots, h(c_m)) = t_{\mathcal{B}}(f(\bar{c}_1), \cdots, f(\bar{c}_m))$. This shows that f is an n-rational morphism. Since f is clearly a bijection, it is an isomorphism between $\mathcal{A}/\text{Ker}(h)$ and $h(\mathcal{A})$.

We also have the following theorem.

THEOREM 8.6. *Given an n-rational Σ-algebra $\mathscr{A}$ and a precongruence $\leqq$ on $\mathscr{A}$ with associated congruence $\cong$, we have the following*:

(1) *The quotient Σ-algebra $\mathscr{A}/\cong$ is n-rational and the natural mapping $\pi: \mathscr{A} \to \mathscr{A}/\cong$ is an n-rational morphism which is monotonic if $\leqq$ is admissible.*

(2) *For every n-rational morphism $h: \mathscr{A} \to \mathscr{B}$, if the precongruence $\leqq$ is contained in $\leqq_h$, then there is a unique n-rational morphism $h^*: \mathscr{A}/\cong \to \mathscr{B}$ such that $h = \pi \cdot h^*$.*



Proof. The proof of (1) is similar to that of Lemma 8.5. For the second part of the theorem, define h^* by $h^*(\bar{c}) = h(c)$. The function h^* is well-defined because by hypothesis, $a \leqq b$ implies $a \leqq_h b$. If h is monotonic, h^* is clearly monotonic. It remains to show that h^* is n-rationally continuous. For any tree t in $n\text{-}RT_\Sigma(u, s)$, for any $(\bar{c}_1, \cdots, \bar{c}_m)$ in $(A/\cong)^u$, we have:

$$\begin{aligned} h^*(t_{\mathscr{A}/\cong}(c_1, \cdots, \bar{c}_m)) &= h^*(\overline{t_{\mathscr{A}}(c_1, \cdots, c_m)}) \\ &= h(t_{\mathscr{A}}(c_1, \cdots, c_m)) \\ &= t_{\mathscr{B}}(h(c_1), \cdots, h(c_m)) \\ &= t_{\mathscr{B}}(h^*(\bar{c}_1), \cdots, h^*(\bar{c}_m)). \end{aligned}$$

Hence, by Theorem 4.4, h^* is n-rationally continuous.

9. Presentations of n-rational algebras. In this section, presentations of n-rational Σ-algebras are discussed. We will be dealing with n-rational Σ-algebras satisfying a set of inequalities. Our goal is to generalize standard results valid for presentations of (unordered) algebras, as presented in Cohn [5], to the class of n-rational algebras.

The main technical tool in dealing with presentations, is the least congruence containing a set of equations. Therefore, it is first necessary to provide a characterization of the least n-rational precongruence containing a set of inequalities. Two equivalent characterizations will be given. The first one is an inductive definition, and the second is a proof system containing one infinitary inference rule.

DEFINITION 9.1. Given an S-indexed family X and an n-rational Σ-algebra $\mathscr{A}$, a function $g: X \to \mathscr{A}$ is a *generating function* for $\mathscr{A}$, if the unique morphism $g^*: n\text{-}RT_\Sigma(X) \to \mathscr{A}$ extending g is surjective. The family X is called a *generating family*.

From a syntactical point of view, there is no difference between an equation and an inequality: both are a pair of trees (t_1, t_2), with t_1, t_2 in $n\text{-}RT_\Sigma(X)$. The difference between an equation and an inequality is in the interpretation that they receive.

DEFINITION 9.2. An equation (t_1, t_2) *holds* in an n-rational Σ-algebra $\mathscr{A}$ with generating function $g: X \to \mathscr{A}$, if $g^*(t_1) = g^*(t_2)$. An inequality (t_1, t_2) *holds* in $\mathscr{A}$ if $g^*(t_1) \sqsubseteq g^*(t_2)$.

Frequently, in order to indicate the intended meaning of the pair (t_1, t_2), we denote an equation as $t_1 = t_2$, and an inequality as $t_1 \leqq t_2$.

Given a relation $E \subseteq (n\text{-}RT_\Sigma(X))^2$ to be interpreted as a set of inequalities, we first characterize the least n-rational precongruence containing E using an inductive definition.

DEFINITION 9.3. Given a relation $E \subseteq (n\text{-}RT_\Sigma(X))^2$, we define by transfinite induction up to ω_1 (the least uncountable ordinal) the sequence of relations $\xrightarrow[E]{\alpha}$ (or $\xrightarrow{\alpha}$, for simplicity) for every ordinal $\alpha < \omega_1$.

(1) $\xrightarrow{0} = E \cup \sqsubset$ (where $\sqsubset$ is the partial ordering on trees).

(2) For every ordinal $\alpha < \omega_1$,

$$\xrightarrow{\alpha+1} = \xrightarrow{\alpha} \cup \{(t_1, t_2) | \exists t_3, (t_1, t_3) \in \xrightarrow{\alpha} \text{ and } (t_3, t_2) \in \xrightarrow{\alpha}\}$$

$$\cup \{(ft_1 \cdots t_m, ft'_1 \cdots t'_m) | (t_i, t'_i) \in \xrightarrow{\alpha} \text{ for } 1 \leqq i \leqq m, \text{ where}$$

$$f \text{ and } t_1, \cdots, t_m, t'_1, \cdots, t'_m \text{ have compatible arity and sorts}\}$$

$$\cup \{(\bigsqcup s_i, t) | \text{ for an } n\text{-iteration } \{s_i | i \in N\} \text{ and } (s_i, t) \in \xrightarrow{\alpha} \text{ for } i \in N\}.$$

(3) For a limit ordinal $\alpha < \omega_1, \xrightarrow{\alpha} = \bigcup_{\beta<\alpha} \xrightarrow{\beta}$.

Finally, $\xrightarrow[E]{*}$ (also denoted $\xrightarrow{*}$) is defined as the union $\bigcup\{\rightarrow \alpha | \alpha < \omega_1\}$.

LEMMA 9.4. *The relation* $\xrightarrow[E]{*}$ *is the least n-rational precongruence containing E.*

Proof. Since no countable ordinal is cofinal to ω_1, it is easily seen that for every countable subset Z of $\xrightarrow[E]{*}$, there is a countable ordinal $\alpha < \omega_1$ such that Z is a subset of $\xrightarrow{\alpha}$. This fact together with the clauses of the inductive definition imply that $\xrightarrow[E]{*}$ contains E, is reflexive, transitive, congruential and n-rationally continuous. Hence, it is a precongruence containing E. Conversely, it is easily shown by transfinite induction, that every n-rational precongruence containing E contains each $\xrightarrow{\alpha}$. Therefore, $\xrightarrow[E]{*}$ is the least n-rational precongruence containing E.

Note that the least quasi n-rational precongruence containing E is obtained by replacing $\sqsubset$ by the identity relation on $n - RT_\Sigma(X)$ in clause (1) of the inductive definition.

The least precongruence $\xrightarrow[E]{*}$ containing E is now characterized using a proof system. In this proof system, a proof is a possibly countably infinite tree. However, even though these trees can have infinite breadth or height, they are *well-founded*, in the sense that they have *no infinite paths.* A rigorous definition can be given using the notion of a tree domain defined in Part I, § 2 (see Gorn [24]).

Recall that a *tree domain* D is a nonempty set of strings over N_+ (the positive integers) such that:

(1) For each u in D, every prefix of u is also in D.

(2) For each u in D, for every positive integer i, if ui is in D, then, for every j, $1 \leqq j \leqq i$, uj is also in D.

We can define a partial ordering $\leqq$ on D as follows: for u, v in D, $u \leqq v$ if and only if u is a prefix of v. The relation $u < v$ holds if $u \leqq v$ and $u \neq v$.

DEFINITION 9.5. A tree domain D is *well founded* if the relation $<$ on D is well founded. Equivalently, there are no infinite strictly increasing sequences $u_1 < u_2 < \cdots < u_n < u_{n+1} < \cdots$. A (countable) tree is well founded if its tree domain is.

Note that a tree is well founded if and only if it has no infinite paths. However, a well-founded tree is not necessarily finite branching.

The importance of well-founded trees lies in the fact that properties about them can be proved by *complete induction.* Also, given a well-founded tree domain, a *rank* function assigning an ordinal to each tree address can be defined. The rank of a node (or tree address) is the "height" of the subtree rooted at that node, and since well-founded trees can be infinite branching, it can be an infinite ordinal. The rank function is defined as follows. Given a (coutable) well-founded tree domain D:

For every leaf u in D, rank $(u)=0$ (recall that a leaf is a tree address such that $\{i|ui\in D\}=\emptyset$). For every other tree address u, rank $(u)=\bigsqcup\{\text{rank}(ui)+1|ui\in D\}$.

The rank or height of the tree (domain) itself is the rank of its root, rank (e). Note the "bottom-up" character of the recursive definition of rank, which is possible because the tree domain is well-founded. As pointed out by Saul Gorn, the concept of rank of a node is a generalization of the height of a node (the supremum of the lengths of all paths from the node to the leaves). If a well-founded tree domain is finite branching, by Koenig's lemma it is finite, and then the rank of node coincides with the usual concept of height.

Since we are only considering countable well-founded tree domains, the rank of any such tree domain is an ordinal strictly less than ω_1 (the least uncountable ordinal). Proof trees are trees whose domain is a countable well-founded domain, and whose labels are certain formulae. In the present case, formulae are inequalities between (possibly infinite) trees in $n\text{-}RT_\Sigma(X)$. However, these infinite trees are constructively specified by finite n-rational recursion schemes. Note also that the trees occurring in inequalities are not to be confused with proof trees. Proof trees are "meta-trees" belonging to the "meta-language", whereas trees occurring in inequalities belong to the "object-language". If an infinite proof tree has a finite constructive (recursive) specification, then it really corresponds to a constructive proof. In any case, the rank of a proof tree gives a certain "measure of complexity" of the proof. This is an important topic in proof theory, but this will not be investigated here.

We are now ready to present the axioms and inference rules of the proof system equivalent to the inductive definition of the least n-rational precongruence containing a given set E of inequalities.

DEFINITION 9.6. Let $E\subseteq(n\text{-}RT_\Sigma(X))^2$ be a set of inequalities.

(1) The *axioms* of the proof system are:

all inequalities in E,

all inequalities $t_1\sqsubset t_2$, where $\sqsubset$ is the partial ordering on $n\text{-}RT_\Sigma(X)$.

(2) The *rules of inference* of the proof system are:

Transitivity: $$\frac{t_1\leqq t_2,\ t_2\leqq t_3}{t_1\leqq t_3}$$

Substitution: $$\frac{t_1\leqq t_1'\cdots t_m\leqq t_m'}{ft_1\cdots t_m\leqq ft_1'\cdots t_m'}$$

provided that the arity of f and the sorts of the t_i, t_i' for $1\leqq i\leqq m$ are compatible.

Lub rule: $$\frac{\{s_i\leqq t\}_{i\in N}}{\bigsqcup s_i\leqq t}$$

for every n-iteration $\{s_i|i\in N\}$.

The lub rule takes ω premises.

A *proof tree* is a countable well-founded tree domain whose nodes are labeled with inequalities in such a way that:

(i) Every leaf is labeled with an axiom.

(ii) Every interior node is labeled with an inequality which is the conclusion of applying one of the above inference rules, the immediate descendants of that node being labeled with the inequalities occurring as premises.

It should be emphasized that when dealing with presentations, the set X is a set of *generators*, which are therefore treated as *constants* and not as variables. Hence, inequalities have no variables. The set X will be treated as a set of variables in the next section when we deal with varieties.

DEFINITION 9.7. Given a set of inequalities E, if there is a proof tree T for an inequality $t_1 \leqq t_2$, we say that *$t_1 \leqq t_2$ is provable from E*, and this is denoted by $E \vdash t_1 \leqq t_2$.

THEOREM 9.8. *Given a set of inequalities E, we have $E \vdash t_1 \leqq t_2$ if and only if $t_1 \overset{*}{\underset{E}{\to}} t_2$.*

Proof. The theorem is shown by proving the following two claims.

CLAIM 1. *For every ordinal $\alpha < \omega_1$, if $t_1 \overset{\alpha}{\to} t_2$, then there is a proof tree T for $t_1 \leqq t_2$ of rank* rank $(T) = \beta$ *for some ordinal $\beta < \alpha$.*

CLAIM 2. *For every ordinal $\alpha < \omega_1$, if there is a proof tree T for $t_1 \leqq t_2$, of rank* rank $(T) = \alpha$, *then there is some ordinal $\beta < \alpha + 1$ such that $t_1 \overset{\beta}{\to} t_2$.*

Both claims are proved by transfinite induction up to ω_1, using the one-to-one correspondence between the clauses of the inductive definition and the inference rules. The details are straightforward and left to the reader.

We can now define the notion of a presentation.

DEFINITION 9.9. Let $g : X \to \mathcal{A}$ be a generating function for an n-rational algebra $\mathcal{A}$ and let $E \subseteq (n\text{-}RT_\Sigma(X))^2$ be a set of inequalities. The pair (g, E) is a *presentation* for the Σ-algebra $\mathcal{A}$ if:

(1) Every inequality $t_1 \leqq t_2$ in E holds in $\mathcal{A}$, that is, $g^*(t_1) \sqsubseteq g^*(t_2)$.

(2) For any arbitrary pair (t_1, t_2) in $(n\text{-}RT_\Sigma(X))^2$, if $t_1 \leqq t_2$ holds in $\mathcal{A}$, that is, $g^*(t_1) \sqsubseteq g^*(t_2)$, then $t_1 \leqq t_2$ is provable from E, that is $E \vdash t_1 \leqq t_2$.

THEOREM 9.10. *Let $g : X \to \mathcal{A}$ be a generating function for an n-rational Σ-algebra $\mathcal{A}$ and let E be a set of inequalities. The following conditions are equivalent*:

(1) *(g, E) is a presentation of $\mathcal{A}$.*

(2) *The precongruence $\overset{*}{\underset{E}{\to}}$ generated by E is equal to the precongruence $\leqq_{g^*}$ induced by g^*.*

Proof. If (g, E) is a presentation for $\mathcal{A}$, every inequality $t_1 \leqq t_2$ in E holds in $\mathcal{A}$, that is, $g^*(t_1) \sqsubseteq g^*(t_2)$. Hence $t_1 \leqq_{g^*} t_2$ and E is a subset of $\leqq_{g^*}$. Since $\overset{*}{\underset{E}{\to}}$ is the least precongruence containing E and $\leqq_{g^*}$ is a precongruence by Lemma 8.3, $\overset{*}{\underset{E}{\to}}$ is a subset of $\leqq_{g^*}$. Conversely, for any arbitrary inequality $t_1 \leqq t_2$, $g^*(t_1) \sqsubseteq g^*(t_2)$ implies that $E \vdash t_1 \leqq t_2$ since (g, E) is a presentation of $\mathcal{A}$. Hence, $\leqq_{g^*}$ is a subset of $\overset{*}{\underset{E}{\to}}$, showing that $\overset{*}{\underset{E}{\to}} = \leqq_{g^*}$, as desired.

It is obvious that (2) implies (1).

COROLLARY. *Every n-rational Σ-algebra $\mathcal{A}$ admitting a presentation (g, E) is isomorphic to the quotient Σ-algebra $n\text{-}RT_\Sigma(X)/\overset{*}{\underset{E}{\leftrightarrow}}$, which is itself presented by (i, E), where $i : X \to n\text{-}RT_\Sigma(X)/\overset{*}{\underset{E}{\leftrightarrow}}$ is the restriction of the natural quotient morphism to X and $\overset{*}{\underset{E}{\leftrightarrow}}$ is the congruence associated with $\overset{*}{\underset{E}{\to}}$.*

Note that the second condition of the definition of a presentation, referred to as the *completeness condition*, implies that whenever $t_1 \sqsubseteq t_2$ (where $\sqsubseteq$ is the partial ordering on $n\text{-}RT_\Sigma(X)$), $g^*(t_1) \sqsubseteq g^*(t_2)$, since g^* is monotonic (see Theorem 3.10). Hence, the completeness condition forces the precongruence $\leqq_{g^*}$ to be admissible, and this is why we are using the least admissible precongruence containing E. However, note that we could define a purely equational version of a presentation, this time taking the least quasi precongruence containing E, denoted by $\overset{*}{\underset{E}{\to}}_w$ and its associated quasicongruence $\overset{*}{\underset{E}{\leftrightarrow}}$, and changing $\leqq$ to $=$ everywhere. We leave the details to the reader.

We also have the following version of "Dyck's lemma" (see Cohn [5, Thm. 8.3]).

LEMMA 9.11. *Let $E \subseteq (n\text{-}RT_\Sigma(X)))^2$ be a set of inequalities and let $\xrightarrow[E]{*}$ be the least precongruence containing E. Let $\pi : n\text{-}RT_\Sigma(X) \to n\text{-}RT_\Sigma(X)/\overset{*}{\underset{E}{\leftrightarrow}}$ be the natural quotient morphism. For any n-rational morphism $h : n\text{-}RT_\Sigma(X) \to \mathscr{A}$ to an n-rational Σ-algebra $\mathscr{A}$, if $h(t_1) \sqsubset h(t_2)$ for all $t_1 \leqq t_2$ in E, there exists a unique n-rational morphism $f : n\text{-}RT_\Sigma(X)/\overset{*}{\underset{E}{\leftrightarrow}} \to \mathscr{A}$ such that $h = \pi \cdot f$.*

Proof. Since $t_1 \leqq t_2$ implies $h(t_1) \sqsubset h(t_2)$, E is a subset of $\leqq_h$. Since $\xrightarrow[E]{*}$ is the least precongruence containing E, and $\leqq_h$ is a precongruence, $\xrightarrow[E]{*}$ is a subset of $\leqq_h$. We conclude by applying Theorem 8.6.

10. Inequational varieties. Before discussing varieties, the following result about substitution will be needed.

Let X and Y be two S-indexed families of sets. A function $f : X \to n\text{-}RT_\Sigma(Y)$ is called a *substitution*. The function f extends to an (unique) n-rational morphism

$$f^\# : n\text{-}RT_\Sigma(X) \to n\text{-}RT_\Sigma(Y)$$

also called a substitution.

LEMMA 10.1. *Let $\leqq$ be a precongruence on $n\text{-}RT_\Sigma(Y)$ with associated congruence $\cong$. Given any function $h : X \to n\text{-}RT_\Sigma(Y)/\cong$, let $f : X \to n\text{-}R_\Sigma(Y)$ be a substitution picking some arbitrary representative in every equivalence class, so that $h(x) = \overline{f(x)}$. Then, for every n-rational tree t in $n\text{-}RT_\Sigma(X)$, we have $h^\#(t) = \overline{f^\#(t)}$.*

Proof. First, the lemma is proved for finite trees by induction on the depth of trees.

(i) If t is a constant a, $f^\#$ and $h^\#$ being homomorphisms, $h^\#(a) = \bar{a} = \overline{f^\#(a)}$.

(ii) If $t = x$ for x in X, since $f^\#$ extends f and $h^\#$ extends h, and since $h(x) = \overline{f(x)}$, we have $h^\#(x) = \overline{f^\#(x)}$.

(iii) If $t = gt_1 \cdots t_m$, then $h^\#(t) = h^\#(gt_1 \cdots t_m) = \overline{gh^\#(t_1) \cdots h^\#(t_m)}$.

By the induction hypothesis, $h^\#(t_i) = \overline{f^\#(t_i)}$, so we have:

$$h^\#(gt_1 \cdots t_m) = \overline{g\overline{f^\#(t_1)} \cdots \overline{f^\#(t_m)}} = \overline{gf^\#(t_1) \cdots f^\#(t_m)} = \overline{f^\#(gt_1 \cdots t_m)} = \overline{f^\#(t)}.$$

For an infinite n-rational tree t, since $t = \bigsqcup t_i$ for an n-iteration $\{t_i \mid i \in N\}$, we have $h^\#(t) = \bigsqcup h^\#(t_i) = \bigsqcup \overline{f^\#(t_i)} = \overline{\bigsqcup f^\#(t_i)} = \overline{f^\#(t)}$. Hence, the lemma is proved.

In the sequel, we will denote by X_0 a fixed S-indexed family of countable sets of variables.

DEFINITION 10.2. An *inequality of sort s* is a pair of trees $t_1 \leqq t_2$ with t_1, t_2 in $n\text{-}RT_\Sigma(X_0)$. Let $\{x_1, \cdots, x_m\}$ be the set of variables occurring in t_1 or t_2. If x_i is of sort s_i, we write this as $x_i : s_i$. We also use the notation $(x_1 : s_1, \cdots, x_m : s_m \to s) \cdot t_1 \leqq t_2$ for the inequality $t_1 \leqq t_2$. We use a similar notation for equations, with $=$ in place of $\leqq$.

DEFINITION 10.3. Given an n-rational Σ-algebra $\mathscr{A}$, an inequality $(x_1 : s_1, \cdots, x_m : s_m \to s) \cdot t_1 \leqq t_2$ is *valid in $\mathscr{A}$*, denoted as $\mathscr{A} \vDash t_1 \leqq t_2$, if for every c in A^u (where $u = s_1 \cdots s_m$), $c^\#(t_1) \sqsubset c^\#(t_2)$ ($c^\#$ is the unique n-rational morphism extending c viewed as a function $c : \{x_1, \cdots, x_m\} \to \mathscr{A}$).

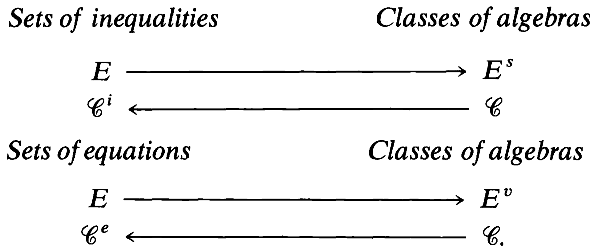
A similar definition can be given for equations by replacing inequalities by equalities. Note that since we are dealing with strictly ordered Σ-algebras, every carrier contains a least element and is therefore nonempty. Hence, the definition of validity given above is correct since carriers cannot be empty.

DEFINITION 10.4. Given a set $E \subseteq (n\text{-}RT_\Sigma(X_0))^2$ of inequalities, the *semi-variety* $\mathscr{C} = (\Sigma, E)$ is the class of all n-rational algebras $\mathscr{A}$ such that every inequality $t_1 \leqq t_2$ in E is valid in $\mathscr{A}$.

If every inequality in E is valid in $\mathcal{A}$, we say that $\mathcal{A}$ is a *model* of E. Hence, the semi-variety $\mathscr{C}$ is the class of all models of E. We shall also denote the semi-variety (Σ, E) by E^s. A similar definition can be given for equations. In this case, we obtain an *equational variety*, for short, a variety, denoted by E^v.

Note that semi-varieties of continuous Σ-algebras have been defined and studied by Guessarian [27], where they are called relational classes of interpretations. Results similar to ours hold for continuous precongruences and semi-varieties of continuous Σ-algebras.

Conversely, given a nonempty class of Σ-algebras $\mathscr{C}$, let $\mathscr{C}^i$ be the set of all inequalities valid in all Σ-algebras in $\mathscr{C}$, and let $\mathscr{C}^e$ be the set of all equations valid in $\mathscr{C}$. Hence, we have two correspondences between sets of inequalities and classes of Σ-algebras, and sets of equations and classes of Σ-algebras:



The following properties can easily be shown. Given any two sets of inequalities E_1, E_2 and two classes of Σ-algebras $\mathscr{C}_1$, $\mathscr{C}_2$, we have:

(1) $E_1 \subseteq E_2$ implies $E_2^s \subseteq E_1^s$;

(2) $\mathscr{C}_1 \subseteq \mathscr{C}_2$ implies $\mathscr{C}_2^i \subseteq \mathscr{C}_1^i$;

(3) $E_1 \subseteq E_1^{si}$ and $\mathscr{C}_1 \subseteq \mathscr{C}_1^{is}$.

Similar inclusions hold for equations, changing s to v and i to e. It is a routine exercise to check that the following identities are consequences of the above identities (see Cohn [5]):

(4) $E_1^s = E_1^{sis}$ and $\mathscr{C}_1^i = \mathscr{C}_1^{isi}$;

(4') $E_1^v = E_1^{vev}$ and $\mathscr{C}_1^e = \mathscr{C}_1^{eve}$.

Therefore, the mappings $si: E \to E^{si}$, $is: \mathscr{C} \to \mathscr{C}^{is}$, $ve: E \to E^{ve}$ and $ev: \mathscr{C} \to \mathscr{C}^{ev}$ are *closure operations* (that is, are idempotent; a mapping f is indempotent if $f \cdot f = f$). The pairs of mappings (i, s) and (v, e) define *Galois connections* between sets of inequalities and classes of n-rational algebras (over Σ), and between sets of equations and classes of Σ-algebras (see Cohn [5]). Note that $\mathscr{C}^{is}$ is the least semi-variety containing $\mathscr{C}$, and $\mathscr{C}^{ev}$ is the least variety containing $\mathscr{C}$.

Our next goal is to characterize the sets of inequalities for which the mapping $s: E \to E^s$ is a bijection. The importance of this characterization lies in the fact that the study of semi-varieties is reduced to the study of certain kinds of congruences. Unfortunately, the author was unable to find such a characterization for equations, but a characterization for inequalities will be given. Such sets of inequalities turn out to be "fully invariant" precongruences. Note that equations can actually be handled as inequalities as follows: given a set of equations E, let E' be the set of inequalities $E' = \{t_1 \leqq t_2, t_2 \leqq t_1 \mid t_1 = t_2 \in E\}$. One can easily check that the semi-variety E'^s is equal to the equational variety E^v. Hence, there is no "loss" in treating only inequalities.

First, a number of definitions and lemmas about classes of algebras are needed. The following lemma will be needed in the proof of Theorem 10.14.

LEMMA 10.5. *Let $\mathscr{F}$ be a free n-rational Σ-algebra generated by X, let $\mathcal{A}$, $\mathscr{B}$ be n-rational Σ-algebras, and let $f: \mathscr{F} \to \mathscr{B}$ and $g: \mathcal{A} \to \mathscr{B}$ be n-rational morphisms, with g*

surjective. There is a function $h: X \to \mathscr{A}$ such that $f = h^{\#} \cdot g$. (Note that h need not be unique.)

Proof. It is standard and is omitted. We refer the reader to Cohn [5, Prop. 5.7].

DEFINITION 10.6. A class $\mathscr{C}$ of n-rational Σ-algebras is *abstract*, if it is closed under n-rational isomorphisms. That is, for every n-rational isomorphism $h: \mathscr{A} \to \mathscr{B}$, if $\mathscr{A}$ is in $\mathscr{C}$, then $\mathscr{B}$ is also in $\mathscr{C}$.

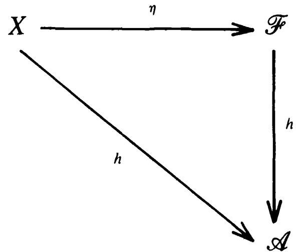
In the rest of this paper, only *nonempty abstract* classes will be considered.

DEFINITION 10.7. A class $\mathscr{C}$ of n-rational Σ-algebras is *nontrivial* if, for every sort s in S, there is a Σ-algebra in $\mathscr{C}$ whose carrier of sort s has at least two elements.

DEFINITION 10.8. Given a class $\mathscr{C}$ of n-rational Σ-algebras and an S-indexed family X, an n-rational Σ-algebra $\mathscr{F}$ is *universal over X* in $\mathscr{C}$ if and only if there exists a function $\eta: X \to \mathscr{F}$ from X to the carrier of $\mathscr{F}$, and the following properties hold:

(i) The *Σ-algebra* $\mathscr{F}$ is in $\mathscr{C}$.

(ii) For every Σ-algebra $\mathscr{A}$ in $\mathscr{C}$, for every function $h: X \to \mathscr{A}$, there is an unique n-rational (monotonic) morphism $h^{\#}: \mathscr{F} \to \mathscr{A}$ such that $h = \eta \cdot h^{\#}$.



If $h^{\#}$ is not monotonic, $\eta: X \to \mathscr{F}$ is called a *weakly universal* algebra. For the sake of briefness, we often say that $\eta: X \to \mathscr{F}$ is an universal algebra over X in $\mathscr{C}$.

The next two lemmas only depend on the universal property of $\mathscr{F}$ and are stated without proof. We refer the reader to Cohn [5] or Gratzer [25] for details.

LEMMA 10.9. *Let $\mathscr{C}$ be a (abstract) class of n-rational Σ-algebras. For any S-indexed family X, let $\eta: X \to \mathscr{F}$ be a universal Σ-algebra over X in $\mathscr{C}$. If the n-rational subalgebra $E([\eta(X)])$ of $\mathscr{F}$ generated by $\eta(X)$ is in $\mathscr{C}$, then $\mathscr{F} = E([\eta(X)])$, that is, $\eta(X)$ generates $\mathscr{F}$.*

Note that Theorem 4.3 is used in the proof.

LEMMA 10.10. *Let $\mathscr{C}$ be a nontrivial class of n-rational Σ-algebras. If $\eta: X \to \mathscr{F}$ is an universal Σ-algebra over X in $\mathscr{C}$, then η is a family of injections.*

When η is a family of injections, $\mathscr{F}$ is called a *free algebra* over X in $\mathscr{C}$. It is easily verified that semi-varieties (and varieties) are closed under subalgebras. Hence Lemma 10.9 applies to semi-varieties. If, in addition, a semi-variety is nontrivial, X can be identified with $\eta(X)$ and generates $\mathscr{F}$. This is the case when the class of all n-rational Σ-algebras is considered. Therefore, there is agreement between the notion of a free Σ-algebra for a class of Σ-algebras and the previous notion of a free Σ-algebra given in Theorem 3.10.

DEFINITION 10.11. Given an n-rational Σ-algebra $\mathscr{A}$, a precongruence $\leqq$ on $\mathscr{A}$ is *fully invariant* if it is preserved under endomorphisms. That is, for every n-rational morphism $h: \mathscr{A} \to \mathscr{A}$, for all x, y in $\mathscr{A}$, $x \sqsubset y$ implies $h(x) \sqsubset h(y)$. Similarly, a *fully invariant* congruence is defined by replacing inequality by equality.

Note that fully invariant precongruences on n-$RT_{\Sigma}(X)$ are those preserved under substitutions, since endomorphisms in n-$RT_{\Sigma}(X)$ are substitutions.

Two more lemmas are needed before stating and proving the theorem characterizing semi-varieties in terms of fully invariant precongruences.

LEMMA 10.12. *Let $<$ be a fully invariant precongruence on $n\text{-}RT_\Sigma(X_0)$ with associated congruence $\cong$. An inequality $t_1 \leqq t_2$ is valid in $n\text{-}RT_\Sigma(X_0)/\cong$ if and only if $t_1 < t_2$.*

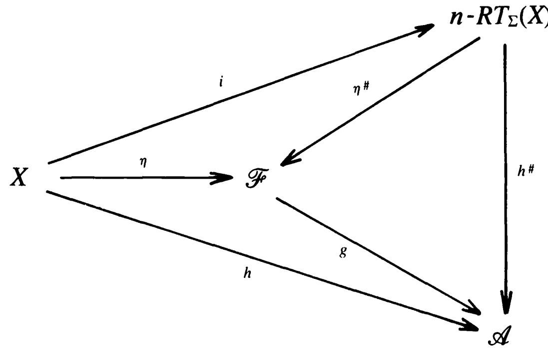
Proof. First, assume $t_1 < t_2$, and let $(x_1 : s_1, \cdots, x_m : s_m \to s) \cdot t_1 < t_2$ be its expanded notation. For any function $h : \{x_1, \cdots, x_m\} \to n\text{-}RT_\Sigma(X_0)/\cong$, by Lemma 10.1, there is a substitution $f : X_0 \to n\text{-}RT_\Sigma(X_0)$ such that $h^*(t) = \overline{f^*(t)}$ for every tree t in $n\text{-}RT_\Sigma(X_0)$. Since $<$ is fully invariant, $t_1 < t_2$ implies that $f^*(t_1) < f^*(t_2)$, and so, $h^*(t_1) = \overline{f^*(t_1)} < \overline{f^*(t_2)} = h^*(t_2)$. Hence, $t_1 \leqq t_2$ is valid in $n\text{-}RT_\Sigma(X_0)/\cong$.

Conversely, if $t_1 \leqq t_2$ is valid in $n\text{-}RT_\Sigma(X_0)/\cong$, let $h : X_0 \to n\text{-}RT_\Sigma(X_0)/\cong$ be such that $h(x) = \bar{x}$. We can take the identity function as the substitution f, and so $h^*(t) = \bar{t}$. Hence, $\bar{t}_1 = h^*(t_1) < h^*(t_2) = \bar{t}_2$ and $t_1 < t_2$, as desired.

The next lemma shows the crucial role played by universal algebras in classes of algebras.

LEMMA 10.13. *Given a class $\mathscr{C}$ of n-rational Σ-algebras closed under subalgebras, for any S-indexed family X, if $\eta : X \to \mathscr{F}$ is an universal algebra over X in $\mathscr{C}$, an inequality $(x_1 : s_1, \cdots, x_m : s_m \to s) \cdot t_1 \leqq t_2$ holds in $\mathscr{F}$, that is $\eta^*(t_1) \sqsubset \eta^*(t_2)$, if and only if $t_1 \leqq t_2$ is valid in $\mathscr{C}$.*

Proof. Assume that $\eta^*(t_1) \sqsubset \eta^*(t_2)$ holds in $\mathscr{F}$. Since $\{x_1, \cdots, x_m\}$ is a subset of X, any assignment $c : \{x_1, \cdots, x_m\} \to \mathscr{A}$ can be extended to a function $h : X \to \mathscr{A}$. Let $h : X \to \mathscr{A}$ be any function to any Σ-algebra in $\mathscr{C}$. Since $\mathscr{F}$ is universal on X, there is an unique morphism g such that $h = \eta \cdot g$. But $n\text{-}RT_\Sigma(X)$ is also free on X and so, there are unique morphisms η^* and h^* such that $i \cdot \eta^* = \eta$ and $i \cdot h^* = h$, where $i : X \to n\text{-}RT_\Sigma(X)$ is the inclusion map.



But then, $i \cdot h^* = i \cdot \eta^* \cdot g = h$, and so $h^* = \eta^* \cdot g$. Hence, $h^*(t_1) = g(\eta^*(t_1)) \sqsubset g(\eta^*(t_2))$ (since g is monotonic) $= h^*(t_2)$. Therefore, $t_1 \leqq t_2$ is valid in $\mathscr{A}$.

Conversely, if $t_1 \leqq t_2$ is valid in $\mathscr{C}$, since $\mathscr{F}$ is in $\mathscr{C}$, $t_1 \leqq t_2$ is valid in $\mathscr{F}$. Choosing $h : X \to \mathscr{F}$ equal to η, we have $\eta^*(t_1) \sqsubset \eta^*(t_2)$, that is, $t_1 \leqq t_2$ holds in $\mathscr{F}$.

Note that the monotonicity of g was crucial to the proof. A similar lemma holds for a weakly universal Σ-algebra and an equation $t_1 = t_2$ because the monotonicity of g is not needed in this case.

The following theorem relating semi-varieties and fully invariant precongruences can now be proved. This theorem is important in the sense that it reduces the study of semi-varieties to the study of fully invariant precongruences. A similar theorem was proved by Guessarian [27, Prop. 5.25] for relational classes of continuous algebras and continuous substitution-closed preorders. The techniques used in our proofs are quite different from those used in Guessarian [27], and more "constructive". An important consequence of Theorem 10.14 is that, if $\leqq$ is a fully invariant admissible n-rational precongruence on $n\text{-}RT_\Sigma(X)$, then the n-rational algebra $n\text{-}RT_\Sigma(X)/\cong$ is

universal on X in the variety $\leqq^s$. This last result is used to prove the soundness of our proof system in Theorem 10.19.

THEOREM 10.14. *The mapping $s: E \to E^s$ is a bijective Galois connection between fully invariant precongruences over n-$RT_\Sigma(X_0)$ and semi-varieties.*

Proof. First, let $\mathscr{C}$ be any (abstract) class of n-rational Σ-algebras. We prove that the set $\mathscr{C}^i$ of inequalities valid in $\mathscr{C}$ is a fully invariant precongruence. It is straightforward to check that $\mathscr{C}^i$ is an admissible congruential preorder, by the definition of validity. Given an inequality $(x_1: s_1, \cdots, x_m: s_m \to s) \cdot t_1 \leqq t_2$, for any assignment $c: \{x_1, \cdots, x_m\} \to \mathscr{A}$, where $\mathscr{A}$ is any Σ-algebra in $\mathscr{C}$, for any substitution $f: n\text{-}RT_\Sigma(X_0) \to n\text{-}RT_\Sigma(X_0)$, $f \cdot c^\#: n\text{-}RT_\Sigma(X_0) \to \mathscr{A}$ is a (monotonic) morphism, and if $t_1 \leqq t_2$ is valid in $\mathscr{A}$, then $c^\#(f(t_1)) \sqsubseteq c^\#(f(t_2))$. Hence, $f(t_1) \leqq f(t_2)$ is valid in $\mathscr{A}$, and so $\mathscr{C}^i$ is fully invariant. It remains to show that it is n-rationally continuous. Given an n-iteration $\{t_i | i \in N\}$ with least upper bound t, let t' be arbitrary in n-$RT_\Sigma(X_0)$ and assume that $t_i \leqq t'$ is valid in $\mathscr{C}$ for all $i \in N$. Let $\{x_1, \cdots, x_m\}$ be the set of variables occurring in t or t'. For every assignment $c: \{x_1, \cdots, x_m\} \to \mathscr{A}$, since $c^\#$ is n-rationally continuous, $c^\#(\bigsqcup t_i) = \bigsqcup c^\#(t_i)$, and since $t_1 \leqq t'$ is valid, we have $c^\#(t_1) \sqsubseteq c^\#(t')$, which implies $c^\#(\bigsqcup t_i) = \bigsqcup c^\#(t_i) \sqsubseteq c^\#(t')$. But this means that $\bigsqcup t_i \leqq t'$ is valid, as desired.

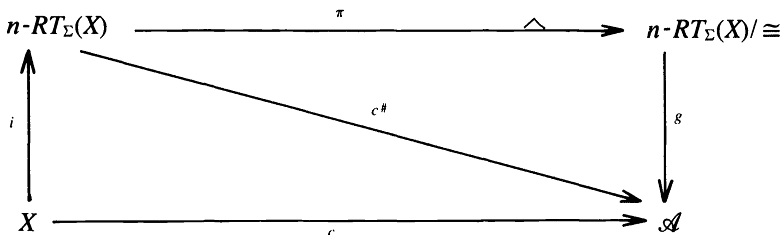
Next, assume that $\leqq$ is a fully invariant precongruence on n-$RT_\Sigma(X_0)$. First, observe that the mapping $s: \leqq \to \leqq^s$ is surjective. Indeed, for any semi-variety $\mathscr{C} = E^s$, since $\mathscr{C}^{is} = E^{sis} = E^s = \mathscr{C}$, $\mathscr{C}^i$ being a fully invariant precongruence, we have $\mathscr{C} = (\mathscr{C}^i)^s$.

We shall show that the mapping s is injective by showing that $\leqq^{si} = \leqq$. Since we know from the Galois connection that $\leqq$ is a subset of $\leqq^{si}$, it suffices to prove the converse. This will follow from the following claim.

CLAIM. *For any S-indexed family X (not necessarily equal to X_0), let $\leqq$ be a fully invariant precongruence on n-$RT_\Sigma(X)$. Then, $\eta: X \to n\text{-}RT_\Sigma(X)/\cong$ is an universal Σ-algebra over X in the semi-variety $\leqq^s$, where η is the restriction of the quotient morphism to X.*

Proof of claim. First, we check that every inequality in $\leqq$ is valid in n-$RT_\Sigma(X)/\cong$. This will show that n-$RT_\Sigma(X)/\cong$ is in $\leqq^s$. Let $(x_1: s_1, \cdots, x_m: s_m \to s) \cdot t_1 \leqq t_2$ be any inequality in $\leqq$. Every assignment $c: \{x_1, \cdots, x_m\} \to n\text{-}RT_\Sigma(X)/\cong$ extends to a (monotonic) n-rational morphism $c^\#$. Let $\pi: n\text{-}RT_\Sigma(X) \to n\text{-}RT_\Sigma(X)/\cong$ be the quotient morphism. Since π is surjective, by Lemma 10.5, there is a function $g: X \to n\text{-}RT_\Sigma(X)$ such that $c^\# = g^\# \cdot \pi$. But since $\leqq$ is fully invariant, $t_1 \leqq t_2$ implies $g^\#(t_1) \leqq g^\#(t_2)$ and so, $c^\#(t_1) = \pi(g^\#(t_1)) \leqq \pi(g^\#(t_2)) = c^\#(t_2)$ since π is monotonic. Hence, $t_1 \leqq t_2$ is valid in n-$RT_\Sigma(X)/\cong$.

For every assignment $c: \{X_1, \cdots, X_m\} \to \mathscr{A}$ to any n-rational algebra $\mathscr{A}$ in $\leqq^s$, consider



Since $\mathscr{A}$ is in $\leqq^s$, whenever $t_1 \leqq t_2$ is valid in $\mathscr{A}$ we have $c^\#(t_1) \sqsubseteq c^\#(t_2)$. Hence, $\leqq$ is a subset of $\leqq_{c^\#}$. By Theorem 8.6, there is an unique n-rational morphism g such that $c^\# = \pi \cdot g$. Then, it is clear that $\eta: X \to n\text{-}RT_\Sigma(X)/\cong$ is universal over X in the semi-variety $\leqq^s$.

To conclude the proof of Theorem 10.14, assume that $t_1 \leqq t_2$ is in $\leqq^{si}$. Since n-$RT_\Sigma(X)/\cong$ is in $\leqq^s$, $t_1 \leqq t_2$ is valid in n-$RT_\Sigma(X)/\cong$. But since n-$RT_\Sigma(X)/\cong$ is

universal, by Lemma 10.12, $t_1 \leqq t_2$ belongs to $\leqq$. Hence, $\leqq^{si}$ is a subset of $\leqq$. But then, $\leqq^{si} = \leqq$, concluding the proof.

COROLLARY. *For any S-induced family X, if $\leqq$ is a fully invariant admissible n-rational precongruence on $n\text{-}RT_\Sigma(X)$, then the n-rational algebra $n\text{-}RT_\Sigma(X)/\cong$ is universal on X in the variety $\leqq^s$.*

DEFINITION 10.15. Let X be any arbitrary S-indexed family and let $E \subseteq (n\text{-}RT_\Sigma(X_0))^2$ be a set of inequalities. The set of *substitution instances* of E in $n\text{-}RT_\Sigma(X)$ is defined as follows: $E(X) = \{c^*(t_1) \leqq c^*(t_2) \mid (x_1 : s_1, \cdots, x_m : s_m \to s) \cdot t_1 \leqq t_2 \in E \cup \sqsubset$ and $c : \{x_1, \cdots, x_m\} \to n\text{-}RT_\Sigma(X)\}$.

((Note that $\sqsubset$ denotes the partial ordering on $n\text{-}RT_\Sigma(X_0)$.)

LEMMA 10.16. *Let X be any S-indexed family and let E be any set of inequalities on $n\text{-}RT_\Sigma(X_0)$. Then $\xrightarrow[E(X)]{*}$ is the least fully invariant n-rational (admissible) precongruence containing $E(X)$.*

Proof. Recall that $\xrightarrow[E(X)]{*}$ is the least n-rational precongruence containing $E(X)$. Hence, it suffices to show that $\xrightarrow[E(X)]{*}$ is fully invariant. This is easily shown by transfinite induction up to ω_1 using the fact that substitutions are n-rationally continuous.

Note that for $X = X_0$, it is not necessary to include substitution instances of inequalities $t_1 \sqsubset t_2$. Hence, we have the following corollary.

COROLLARY. *The relation $\xrightarrow[E(X_0)]{*}$ is the least fully invariant n-rational precongruence containing E.*

The above corollary shows that an equivalent proof system is obtained if the axioms are replaced by all substitution instances of the axioms. Instead of replacing the axioms by their substitution instances, we can add the *instantiation rule* to the proof system of Definition 9.6.

Instantiation rule; i.e., If $\{x_1 : s_1, \cdots, x_m : s_m\}$ is the set of variables occurring in t_1 or t_2, for every substitution $f : \{x_1, \cdots, x_m\} \to n\text{-}RT_\Sigma(X)$,

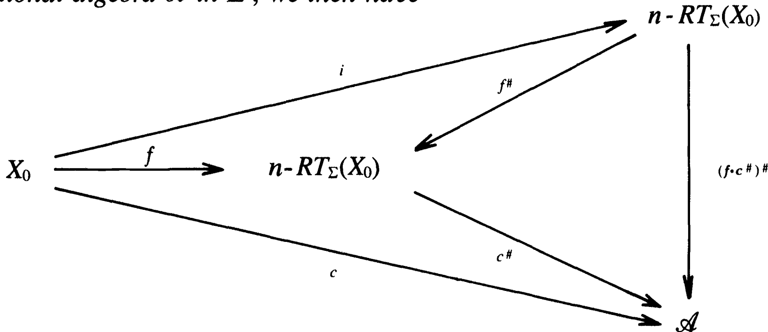
$$\frac{t_1 \leqq t_2}{f^*(t_1) \leqq f^*(t_2)}.$$

The following lemma shows the equivalence of the two proof systems. The proof proceeds by (transfinite) induction on the rank of proof trees and presents no difficulty. Details are left to the reader.

LEMMA 10.17. *For any inequality $t_1 \leqq t_2$, $E \vdash t_1 \leqq t_2$ in a proof possibly using the instantiation rule if and only if $E(X) \vdash t_1 \leqq t_2$ without using the instantiation rule.*

LEMMA 10.18. *Let E be a set of inequalities over $n\text{-}RT_\Sigma(X_0)$. Then, E and $E(X_0)$ define the same semi-variety E^s.*

Proof. Since E is a subset of $E(X_0)$, from the Galois connection, $E(X_0)^s$ is a subset of E^s. Conversely, for every inequality $(x_1 : s_1, \cdots, x_m : s_m \to s) \cdot t_1 \leqq t_2$ in E, for any substitution $f : \{x_1, \cdots, x_m\} \to n\text{-}RT_\Sigma(X_0)$ and any assignment $c : \{x_1, \cdots, x_m\} \to \mathcal{A}$ to any n-rational algebra $\mathcal{A}$ in E^s, we then have



Since both n-rational morphisms $(f\cdot c^{\#})^{\#}$ and $f^{\#}\cdot c^{\#}$ extend $f\cdot c^{\#}$, they must be equal. Then, we have, $c^{\#}(f^{\#}(t_1))=(f\cdot c^{\#})^{\#}(t_1)\sqsubseteq(f\cdot c^{\#})^{\#}(t_2))=c^{\#}(f^{\#}(t_2))$ since $t_1\leqq t_2$ is valid in $\mathcal{A}$. But every inequality $f^{\#}(t_1)\leqq f^{\#}(t_2)$ in $E(X_0)$ is valid in $\mathcal{A}$, and so, $\mathcal{A}$ belongs to $E(X_0)^s$. Hence, E^s is a subset of $E(X_0)^s$ and $E^s=E(X_0)^s$.

COROLLARY. *Let E be a set of inequalities. Then E and $\xrightarrow[E(X_0)]{*}$ both define the semi-variety E^s.*

Proof. From the corollary to Lemma 10.16, $\xrightarrow[E(X_0)]{*}$ is the least fully invariant n-rational precongruence containing E. But E^{si} is also a fully invariant precongruence containing E. Hence, $\xrightarrow[E(X_0)]{*}$ is a subset of E^{si}. Using the Galois connection, this implies that $E^{sis}=E^s$ is a subset of $\xrightarrow[E(X_0)]{*}{}^s$. But E is a subset of $\xrightarrow[E(X_0)]{*}$, so $\xrightarrow[E(X_0)]{*}{}^s$ is a subset of E^s. Therefore, $\xrightarrow[E(X_0)]{*}{}^s=E^s$.

The *soundness and completeness theorem* will now be proved.

THEOREM 10.19. *Given a set E of inequalities over n-$RT_\Sigma(X_0)$, an inequality $(x_1\colon s_1,\cdots,x_m\colon s_m\to s)\cdot t_1\leqq t_2$ is valid in the semi-variety E^s if and only if $E(X_0)\vdash t_1\leqq t_2$.*

Proof. First, we prove *soundness*. Assume that $E(X_0)\vdash t_1\leqq t_2$, that is, $t_1\xrightarrow[E(X_0)]{*}t_2$. Since $\xrightarrow[E(X_0)]{*}$ is the least fully invariant n-rational precongruence containing E, and since both E and $\xrightarrow[E(X_0)]{*}$ define the same semi-variety E^s, by the corollary to Theorem 10.14, letting $\cong$ denote the equivalence associated with $\xrightarrow[E(X_0)]{*}$, n-$RT_\Sigma(X_0)/\cong$ is universal over X_0 in E^s. By Lemma 10.12, $t_1\leqq t_2$ is valid in n-$RT_\Sigma(X_0)/\cong$ if and only if $t_1\xrightarrow[E(X_0)]{*}t_2$. By Lemma 10.13, since $t_1\leqq t_2$ is valid in n-$RT_\Sigma(X_0)/\cong$, it is valid in E^s, establishing soundness.

Completeness. Assume $t_1\leqq t_2$ is valid in E^s. By the above reasoning, n-$RT_\Sigma(X_0)/\cong$ is universal over X_0 in E^s and so, $t_1\leqq t_2$ is valid in n-$RT_\Sigma(X_0)/\cong$. By Lemma 10.12, $t_1\xrightarrow[E(X_0)]{*}t_2$, that is $E(X_0)\vdash t_1\leqq t_2$, establishing completeness.

The following theorem shows that universal algebras in semi-varieties are given by presentations.

THEOREM 10.20. *Given any set E of inequalities over n-$RT_\Sigma(X_0)$, for any S-indexed family X, $n\colon X\to n$-$RT_\Sigma(X)\cong$ is the universal Σ-algebra over X in the semi-variety E^s, where $\cong$ is the equivalence associated with $\xrightarrow[E(X)]{*}$. In particular, if E^s is nontrivial, n-$RT_\Sigma(X)/\cong$ is the free Σ-algebra over X in E^s.*

Proof. (1) First, we show that n-$RT_\Sigma(X)/\cong$ is in E^s. Let $(x_1\colon s_1,\cdots,x_m\colon s_m\to s)\cdot t_1\leqq t_2$ be any inequality in E. For any assignment $c\colon\{x_1,\cdots,x_m\}\to n$-$RT_\Sigma(X)/\cong$, by Lemma 10.1, there is a substitution $f\colon\{x_1,\cdots,x_m\}\to n$-$RT_\Sigma(X)$ such that $c^{\#}(t)=\overline{f^{\#}(t)}$. But then, since $t_1\leqq t_2$ is in E, $f^{\#}(t_1)\leqq f^{\#}(t_2)$ is in $E(X)$, and since $\xrightarrow[E(X)]{*}$ is a precongruence containing $E(X)$, we have

$$c^{\#}(t_1)=\overline{f^{\#}(t_1)}\leqq\overline{f^{\#}(t_2)}=c^{\#}(t_2).$$

Hence, $t_1\leqq t_2$ is valid in n-$RT_\Sigma(X)/\cong$.

(2) It is easy to show that the semi-variety E^s is contained in the variety $E(X)^s$.

(3) From Lemma 10.16, $\xrightarrow[E(X)]{*}$ is the least fully invariant precongruence containing $E(X)$ and, from the corollary to Theorem 10.14, n-$RT_\Sigma(X)/\cong$ is a universal algebra over X in $E(X)^s$. Hence, from (1), (2), (3) above, it follows that n-$RT_\Sigma(X)/\cong$ is universal over X in E^s.

Note that the universal Σ-algebra n-$RT_\Sigma(X)/\cong$ is presented by the set of generators X and by the set of inequalities $E(X)$.

11. A "Birkhoff variety theorem" for classes of n-rational algebras. In this section, semi-varieties of n-rational Σ-algebras are characterized in terms of closure

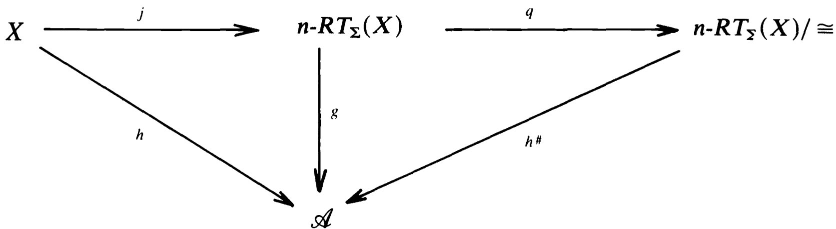
operations. First, we need the following theorem which shows the existence of universal Σ-algebras in abstract classes of n-rational Σ-algebras closed under subalgebras and direct products.

LEMMA 11.1. *Let $\mathscr{C}$ be any (abstract) class of n-rational Σ-algebras closed under subalgebras and direct products. Then, for every S-indexed family X, universal Σ-algebras over X exist in $\mathscr{C}$.*

Proof. Let $n\text{-}RT_\Sigma(X)$ be the free n-rational Σ-algebra on X, with injection $j: X \to n\text{-}RT_\Sigma(X)$. Let $(\leqq_i)_{i\in I}$ be the family of all n-rational precongruences on $n\text{-}RT_\Sigma(X)$ such that $n\text{-}RT_\Sigma(X)/\cong_i$ is isomorphic to an algebra in $\mathscr{C}$. Let $u_i = j \cdot q_i$ be the composition of u_i and the natural quotient morphism $q_i: n\text{-}RT_\Sigma(X) \to n\text{-}RT_\Sigma(X)/\cong_i$. The family $(u_i)_{i\in I}$ defines a function $u: X \to \prod(n\text{-}RT_\Sigma(X)/\cong_i)$ to the direct product of the $n\text{-}RT_\Sigma(X)/\cong_i$. Let F be the n-rational subalgebras of $\prod(n\text{-}RT_\Sigma(X)/\cong_i)$ generated by $u(X)$. Since $\mathscr{C}$ is closed under direct products and subalgebras, $\mathscr{F}$ is in $\mathscr{C}$. Let $\pi_i: \prod(n\text{-}RT_\Sigma(X)/\cong_i) \to n\text{-}RT_\Sigma(X)/\cong_i$ be the ith projection. Observe that, the q_i being surjective, $u_i(X)$ generates $n\text{-}RT_\Sigma(X)/\cong_i$ and so, the restriction of π_i to $\mathscr{F}$ is surjective. This shows that $\mathscr{F}$ is a subdirect product of the $n\text{-}RT_\Sigma(X)/\cong_i$. Let $\leqq$ be the intersection of the family $(\leqq_i)_{i\in I}$. It is clear that $\leqq$ is an n-rational precongruence. Furthermore, $\mathscr{F}$ is isomorphic to $n\text{-}RT_\Sigma(X)/\cong$, where $\cong$ is the equivalence associated with $\leqq$. Indeed, the function $u: X \to \prod(n\text{-}RT_\Sigma(X)/\cong_i)$ extends to a unique morphism $u^\#$, and it is immediately verified that the precongruence $\leqq_{u^\#}$ induced by $u^\#$ is equal to $\leqq$. Hence, by Lemma 8.5, $n\text{-}RT_\Sigma(X)/\cong$ and $u^\#(n\text{-}RT_\Sigma(X)) = \mathscr{F}$ are isomorphic. Let $\eta = j \cdot q$,

CLAIM. *$\eta: X \to \mathscr{F}$ is universal over X in $\mathscr{C}$.*

Let $\mathscr{A}$ be any Σ-algebra in $\mathscr{C}$ and let $h: X \to \mathscr{A}$ be any function. There is an unique n-rational morphism $g: n\text{-}RT_\Sigma(X) \to \mathscr{A}$ extending h. Since $\mathscr{C}$ is closed under subalgebras, $g(n\text{-}RT_\Sigma(X))$ is in $\mathscr{C}$ (note, Lemma 4.1 is used here). But $g(n\text{-}RT_\Sigma(X))$ is isomorphic to $n\text{-}RT_\Sigma(X)/\text{Ker}(g)$ and so, $\text{Ker}(g) = \cong_i$ for some $i \in I$. Let $\mathscr{B} = n\text{-}RT_\Sigma(X)/\cong_i$.



Since $\leqq$ is the intersection of the $\leqq_i$ and $\mathscr{B} = n\text{-}RT_\Sigma(X)/\cong_i$, $\leqq$ is a subset of $\leqq_g$. By Theorem 8.6, there is an unique morphism $h^\#$ such that $g = q \cdot h^\#$. Hence, $\eta: X \to n\text{-}RT_\Sigma(X)/\cong$ is universal over X in $\mathscr{C}$.

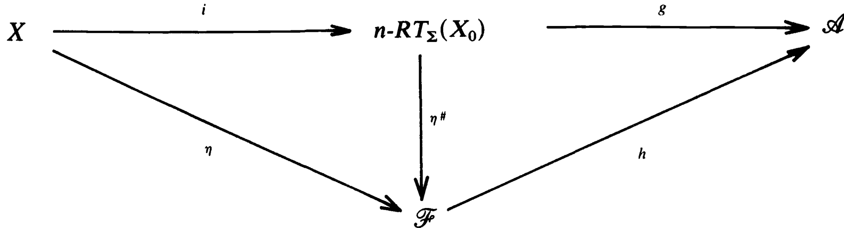
Note that $\eta(X)$ generates $\mathscr{F}$ and that $\mathscr{F}$ is a subdirect product of Σ-algebras in $\mathscr{C}$.

The following theorem gives a purely algebraic (model-theoretic) characterization of n-rational semi-varieties of n-rational Σ-algebras.

THEOREM 11.2. *A (nonempty abstract) class of n-rational Σ-algebras is a semi-variety if and only if it is closed under direct products, subalgebras and images under (monotonic) n-rational morphisms.*

Proof. We leave it to the reader to verify that semi-varieties are closed under direct products, subalgebras and images under n-rational morphisms.

Conversely, let $\mathscr{C}$ be a class of n-rational Σ-algebras closed under these operations. Let $\mathscr{A}$ be any Σ-algebra in $\mathscr{C}^{is}$. Let X be an S-indexed family of generators for $\mathscr{A}$ (X can be taken to be $\mathscr{A}$ itself). Let $g: n\text{-}RT_\Sigma(X) \to \mathscr{A}$ be the surjective n-rational morphism extending the inclusion $X \to \mathscr{A}$. By Theorem 11.1, there is a universal Σ-algebra $\eta: X \to \mathscr{F}$ over X in $\mathscr{C}$. Consider the diagram:



Let $\eta^{\#}$ be the unique extension of η. Note that $\eta(X)$ generates $\mathscr{F}$ since $\mathscr{C}$ is closed under subalgebras. We consider $\mathscr{F}$ as being presented by η and the precongruence $\leqq_{\eta^*}$. Every inequality $t_1 \leqq t_2$ holding in $\mathscr{F}$, that is $\eta^*(t_1) \sqsubseteq \eta^*(t_2)$, is valid in $\mathscr{C}$ by Lemma 10.13. Hence, whenever $t_1 \leqq_{\eta^*} t_2$, $t_1 \leqq t_2$ is valid in $\mathscr{A}$, which implies $g(t_1) \sqsubseteq g(t_2)$ since g is monotonic. Hence, $\leqq_{\eta^*}$ is a subset of $\leqq_g$. By Theorem 8.6, there is a unique n-rational morphism $h: \mathscr{F} \to \mathscr{A}$ since $\mathscr{F}$ is isomorphic to $n\text{-}RT_\Sigma(X)/\text{Ker}(\eta^*)$. Furthermore, g being surjective, h is also surjective and $h(\mathscr{F}) = \mathscr{A}$. Since $\mathscr{C}$ is closed under images of n-rational morphisms and $\mathscr{F}$ is in $\mathscr{C}$, $\mathscr{A}$ is also in $\mathscr{C}$. Hence, we have shown that $\mathscr{C}^{is}$ is contained in $\mathscr{C}$. From the Galois connection, $\mathscr{C}$ is a subset of $\mathscr{C}^{is}$, and therefore, $\mathscr{C} = \mathscr{C}^{is}$. Hence, the semi-variety $\mathscr{C}$ is defined by the set of inequalities $\mathscr{C}^i$.

12. Semi-varieties and "algebraic classes." This section points out connections between the algebraic classes of interpretations studied in [7], [8], [10], [26], [27] and semi-varieties. This topic should be investigated in more detail and is left for further research.

It is frequently the case that the set E of inequalities is a set of inequalities between finite trees. In particular, this is the case when inequational axioms are used to axiomatize classes of interpretations for recursion schemes as in Courcelle [7]. One might expect that in this case, proofs of inequalities among infinite trees representing the result of unfolding recursion schemes, are simpler than in the case where the axioms in E are infinite too. This is indeed the case. It is shown below that every inequality valid in the semi-variety generated by a set of finite inequalities has a proof tree of rank at most ω. The lub rule can be simplified, and every proof tree is equivalent to another tree in which this new rule occurs at the root of the tree, if it occurs at all. In this case, the preorder $\xrightarrow[E]{*}$ is "algebraic" in the sense of Courcelle and Nivat [10]. The part of a proof which does not involve the new lub rule can be carried out using E purely as a rewriting system, the lub rule being used only when passing to the limit. Hence, it may be expected that ideas and results in Courcelle [7] may be used to study semi-varieties generated by finite inequalities. This is left for further research.

DEFINITION 12.1. The *inductive lub rule* is defined as the following inference rule.

$$\frac{\forall i \exists j t_1^i \leqq t_2^j}{\bigsqcup t_1^i \leqq \bigsqcup t_2^j}$$

for any two n-iterations $\{t_1^i \mid i \in N\}$ and $\{t_2^j \mid j \in N\}$ in $n\text{-}RT_\Sigma(X_0)$.

In the theorem below, it is assumed that only inequalities $t_1 \sqsubseteq t_2$ between finite trees are included in the axioms, besides inequalities in E.

THEOREM 12.2. *Given a set E of inequalities between finite trees, if any inequality $t_1 \leqq t_2$ has a proof tree T, then it has a proof tree T' of rank at most ω, and the inductive lub rule appears at the root, if it appears at all.*

Proof. The proof proceeds by complete induction on the rank of proof trees.

(i) First, we have to show that every substitution instance of an inequality in E has a proof tree satisfying the condition of the theorem. Let $(x_1: u_1, \cdots, x_m: u_m \to u) \cdot t_1 \leqq t_2$ be any inequality in E. Let $f: \{x_1, \cdots, x_m\} \to n\text{-}RT_\Sigma(X_0)$ be any substitution. Let us denote $f^*(x_i)$ by s_i. For each s_i, there is an n-iteration $\{s_i^j \mid j \in N\}$ such that $s_i = \bigsqcup s_i^j$. Using the instantiation rule, one can easily show that $t_1(s_1^j, \cdots, s_m^j) \leqq t_2(s_1^j, \cdots, s_m^j)$ has a (finite) proof tree, for each $j \in N$. Then, by application of the inductive lub rule, we obtain the desired proof tree for

$$t_1(s_1, \cdots, s_m) = \bigsqcup t_1(s_1^j, \cdots, s_m^j) \leqq \bigsqcup t_2(s_1^j, \cdots, s_m^j) = t_2(s_1, \cdots, s_m).$$

Note also that the partial ordering on trees being algebraic, it is easily shown that every inequality $t_1 \sqsubset t_2$ has a proof.

The rest of the proof proceeds by cases. Only some of them are treated, the others being similar and so left to the reader.

(ii) The root of the proof tree T is an application of the transitivity rule. Then, T has two subtrees T_1 and T_2, and the inductive hypothesis applies to them. Hence, there are two proof trees T_1 for $t_1 \leqq t_3$ and T_2 for $t_3 \leqq t_2$, such that rank (T_1), rank $(T_2) \leqq \omega$ and if present, the inductive lub rule labels the root of T_1 and T_2. But then, for every i, for some j, we must have a subproof of $t_1^i \leqq t_3^j$ and for every j, for some k, there is a subproof of $t_3^j \leqq t_2^k$. Furthermore, these proof trees are finite. Using the transitivity rule, for every i, for some k, there is a finite proof tree for $t_1^i \leqq t_2^k$. Applying the inductive lub rule, the desired proof tree T' is obtained.

(iii) The root of the proof tree T is an application of the substitution rule. This case is similar to (i) and is omitted.

(iv) The lub rule labels the root of the proof tree T. In this case, the premises are inequalities $t_1^i \leqq t_2$ for finite trees t_1^i. By the inductive hypothesis, each such inequality has a proof tree in which, if the inductive lub rule is used, it is used at the root. But the trees t_1^i being finite, for each i, there is a j such that $t_1^i \leqq t_2^j$ is provable. Hence, an equivalent proof tree with a single application of the inductive lub rule at the root is obtained. This concludes the proof.

Note that alternatively, instead of including as axioms all inequalities between finite trees given by the partial ordering on trees, we could have used the instantiation rule and the reflexivity rule both restricted to finite trees.

We now briefly compare the approach using n-rational Σ-algebras with the more standard approach using continuous Σ-algebras. In order to simplify the notation, let us denote the preorder $\xrightarrow[E]{*}$ defined by the set of inequalities E as $\leqq$. Then, note that Theorem 12.2 shows that $\leqq$ is algebraic in the sense of Courcelle and Nivat [10]. As in Courcelle [7], the set E of finite inequalities induces a rewriting system which defines the rewrite relation $\xrightarrow[E]{}$. One can easily prove that for finite trees, $E \vdash t_1 \leqq t_2$ if and only if $t_1(\xrightarrow[E]{} \cup \sqsubset)^* t_2$. Hence, $\leqq$ is the "algebraic closure" of $(\xrightarrow[E]{} \cup \sqsubset)^*$. It is shown in Courcelle [7] that the universal Σ-algebra over X_0 in the semi-variety E is obtained as the ω-completion of the quotient $FT_\Sigma(X_0)/(\xrightarrow[E]{} \cup \sqsubset)^*$, and that the quotient $CT_\Sigma(X_0)/\cong$, where $\cong$, the equivalence associated with $\leqq$, is not always ω-continuous. However, $\leqq$ being an n-rational precongruence, the quotient $n\text{-}RT_\Sigma(X_0)/\cong$ is n-rational. Hence, not considering all chains but only n-iterations pays off, since no completion operation is needed.

There is however a negative counterpart. Courcelle [7] has shown that it is fruitful to investigate the notion of normal form of an infinite tree, under a rewrite relation. If $\{t_1^i | i \in N\}$ is an n-iteration, assuming that each t_1^i reduces to a normal form t_2^i under $(\underset{E}{\rightarrow} \cup \sqsubset)^*$, and assuming that $\{t_2^i | i \in N\}$ is also a chain, it is tempting to take $\bigsqcup t_2^i$ as a normal form for $\bigsqcup t_1^i$. However, although $\{t_2^i | i \in N\}$ may be a chain, it may *not* be an n-iteration, and $\bigsqcup t_2^i$ may not be in n-$RT_\Sigma(X_0)$. The following example due to Courcelle [7] illustrates this behavior for recursion schemes (1-$RT_\Sigma(X_0)$).

Example 12.1. The n-iteration $\{t_1^i | i \in N\}$ is obtained by unfolding the scheme

$$F(x) \Leftarrow h(x, F(f(g(x))))$$

and the inequalities are

$$f(g(x)) \leqq g(f(x)) \text{ and } f(\perp) \leqq \perp.$$

Figure 1 shows the tree t_1^i, and Fig. 2 shows the tree t_2^i. The tree $\bigsqcup t_2^i$ is not context-free.

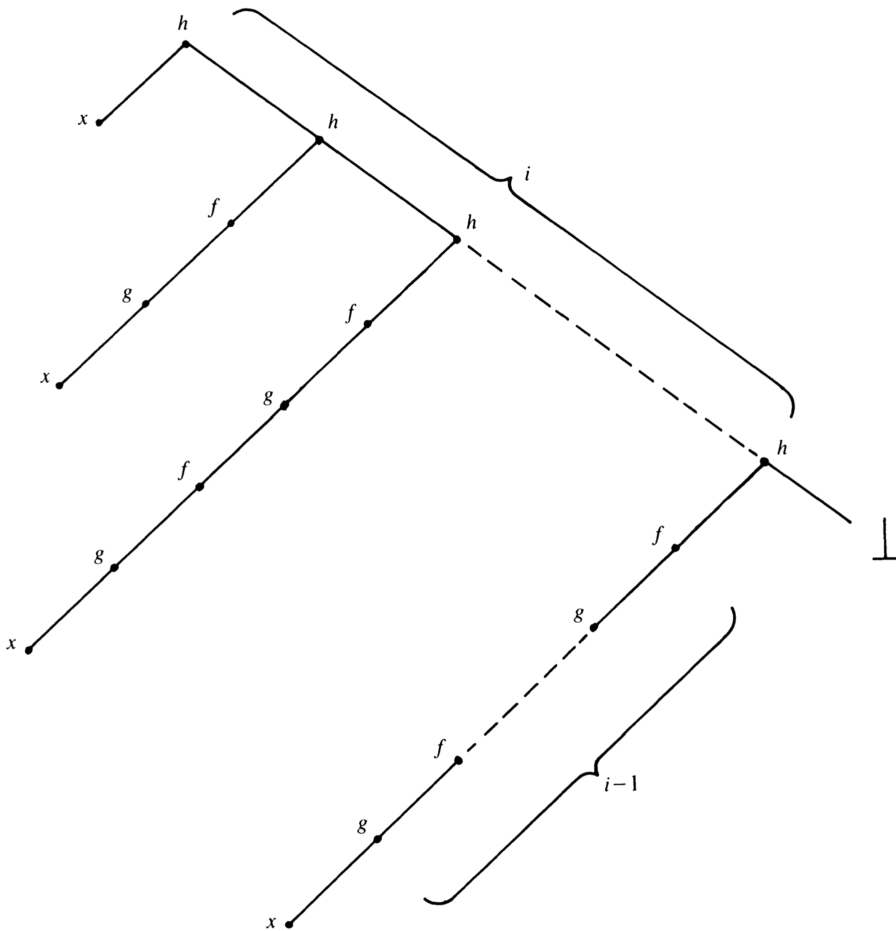


FIG. 1

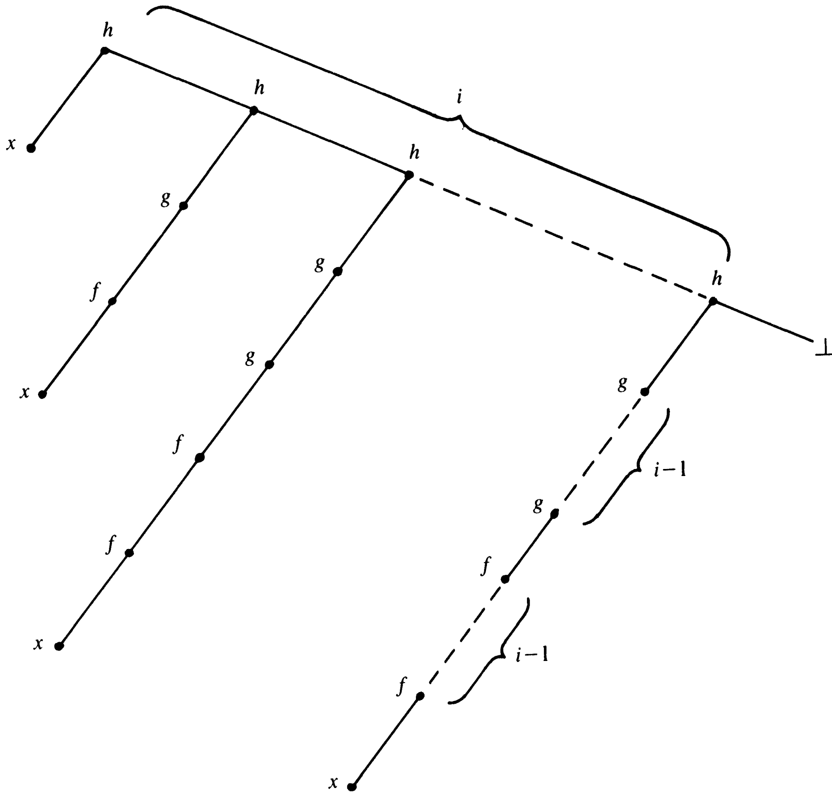


FIG. 2

This example shows a weakness of the approach using n-rational Σ-algebras. These investigations are left for further research.

Acknowledgments. I wish to thank Irene Guessarian, Saul Gorn, Scott Weinstein and especially Eric Wagner, for their helpful comments.

REFERENCES

[*See Part I, this issue, pp.* 774–775.]

SIAM J. COMPUT.
Vol. 13, No. 4, November 1984

© 1984 Society for Industrial and Applied Mathematics
009

THE INFORMATION-THEORETIC BOUND IS GOOD FOR MERGING*

NATHAN LINIAL†

Abstract. Let $A=(a_1>\cdots>a_m)$ and $B=(b_1>\cdots>b_n)$ be given ordered lists: also let there be given some order relations between a_i's and b_j's. Suppose that an unknown total order exists on $A\cup B$ which is consistent with all these relations (= a linear extension of the partial order) and we wish to find out this total order by comparing pairs of elements $a_t:b_s$. If the partial order has N linear extensions, then the Information Theoretic Bound says that $\log_2 N$ steps will be required in the worst case from any such algorithm. In this paper we show that there exists an algorithm which will take no more than $C\log_2 N$ comparisons where $C=(\log_2((\sqrt{5}+1)/2))^{-1}$. The computation required to determine the pair $a_t:b_s$ to be compared has length polynomial in $(m+n)$. The constant C is best possible. Many related results are reviewed.

Key words. theoretic bound, partially ordered sets, order ideals, lattice paths, convex polygons

1. Introduction and review. This paper is a part of an effort to answer the question "How good is the Information Theoretic Lower Bound." This question had already received considerable attention, e.g., [*Fr*][*GYY*1]. For many algorithmic problems, the quest of an answer is equivalent to searching a certain space whose elements are referred to as "compatible solutions" in the sense that they do not contradict the presently available information concerning the solution. Let us assume that our queries concerning the solution are such that they permit exactly two answers (the generalization to other cases is obvious). Thus the space of compatible solutions is split into two parts according to the answer. Assuming answers are given by an adversary, we may assume that the actual answers are always such that we are left with the majority of the compatible solutions after each query. The best one can do is to make such a query for which the space of compatible solutions is split into two equal parts. For this optimal strategy the number of steps will thus be $\log_2 N_0$ where N_0 is the initial number of compatible solutions. The problem is of course that in many situations such an efficient query which splits the compatible solutions into two sets of equal size does not exist. The purpose of this paper is to investigate the quality of the ITB under such circumstances.

This general model of a problem encompasses a great variety of search–sort problems and the situation varies from one problem to another. In many interesting families of problems which are included in this model the following situation occurs: although in general one cannot always find an optimal query which splits the space of compatible solutions into two equal parts, one can find a constant $\frac{1}{2}\geqq\alpha>0$ such that a query can always be found for which the smaller subspace has size at least α times the size of whole compatible solution space. (So the size of the large subspace is at most $(1-\alpha)$ times the size of the whole space.) In this case it is clear that the solution can be found in $\log N_0/\log(1-\alpha)$ steps (N_0 again being the initial size of the space of compatible solutions). In such cases the ITB gives the right order of magnitude for the optimal number of steps. Sometimes a somewhat more complicated result can be stated: there is an integer k and a constant $0<\beta<1$ so that one can always find k queries with the property that no matter what answers one receives the size of the remaining subspace is at most β times the size of the space before these queries were

* Received by the editors August 7, 1982, and in revised form July 14, 1983.

† Institute of Mathematics and Computer Science, The Hebrew University of Jerusalem, Jerusalem 91904 Israel.

made. Clearly the ITB gives here, too, the correct order of magnitude for the optimal number of steps.

Let us review some previous work in which this situation has been shown to occur;

(1) [LS] Let (T, r) be a tree rooted at r. The space of compatible solutions consists of all subtrees of T rooted at r. The queries are: a node x in T is picked and one asks whether x belongs to the chosen subtree. One proves here:

a) There is always a node which belongs to a fraction α of the compatible trees where $\frac{1}{3} \leqq \alpha \leqq \frac{2}{3}$.

b) One can always find $k \leqq 3$ vertices (=queries) so that after these queries are answered, the size of the compatible solution space drops to at most λ^k of its initial size where $\lambda = 5^{-1/3}$. The constant λ is best possible and the cases of equality are completely characterized.

2) Let $(P, \geqq)$ be a finite poset and consider the space of its nonempty ideals (= down sets; $A \subseteq P$ is an ideal if $x \in A$, $y < x$ implies $y \in A$). A query is made here by asking whether an element x of P belongs to the chosen ideal or not. This space is related to a large variety of search problems (see [LS]). Sands [Sa] has shown that if one restricts the attention to posets of height $\leqq k$, then there is a constant $\alpha_k < \frac{1}{2}$ so that one can always find an $x \in P$(= a query) such that the fraction of those order ideals containing x is between α_k and $1-\alpha_k$. A major open problem in this field is the following:

Problem 1 [Sa]. [LS]. Prove that there is a universal constant $0 < \alpha < \frac{1}{2}$ so that in any finite poset P there is an element x for which

$$1-\alpha > \frac{\text{no. of ideals in } P \text{ containing } x}{\text{no. of ideals in } P} > \alpha.$$

3) [KLS] Let $G = (V, E, r)$ be a connected graph roots at r. Let the space of compatible solutions be the collection of all connected subgraphs containing r. A query is made by picking a vertex $x \in V$ and asking if it belongs to the connected subgraph. If no further assumption is made on the graph G, then the ITB may totally fail. If for example one chooses G to be C_n—the circuit on n vertices—and r to be any designated vertex, then $N_0 = O(n^2)$ is the number of connected subgraphs of G containing r. However for certain connected subgraphs, like the whole graph minus one vertex, the search will require $n-1$ queries.

However, if one assumes that all vertices in G have degree at least three, then it can be shown [KLS] that $N_0 \geqq 2^{n/4}$ and so the ITB must be good (for example, make all n possible queries). Not much is known, though, about how to find the most efficient queries and how efficient they are.

2. The problem and the main theorem. In the standard sorting problem [Kn], as everyone knows, one is given n elements $x_1, \cdots, x_n$ and one has to find a total order on them by comparing pairs x_i: x_j. The ITB implies that at least $\log_2 N_0 = \log_2 n!$ steps are required and that this bound can be more-or-less achieved. Consider now the following more general problem:

The general sorting problem. The input consists of n elements $x_1, \cdots, x_n$ together with some order relations between them. One is to discover their total order which is known to be compatible with the input order relations.

Formal restatement of the problem. Let $(P, \geqq)$ be a finite poset. There is a linear order on P compatible with $\geqq$ (an extension of $\geqq$) which is unknown to us. This extension is to be discovered by querying the order relations between pairs of elements $x, y \in P$ where x, y are unrelated by $\geqq$.

The ITB implies that any algorithm which solves this problem requires at least $\log_2 N_0$ steps where N_0 is the number of extensions of $\geqq$. We conjecture that the ITB gives the right order of magnitude. Namely, we make the following

CONJECTURE 1. *There is a universal constant $c > 1$ such that the general sorting problem can be solved in $c \log_2 N_0$ steps where N_0 is the number of extensions of $(P, \geqq)$.*

We want to make an even sharper conjecture asserting that one can always find an efficient query. To this end we make the following

DEFINITION. Let $(P, \geqq)$ be a poset, $x, y \in P$.

$$\Pr(x > y) := \frac{\text{no. of extensions of } (P, \geqq) \text{ in which } x > y}{\text{no. of extensions of } (P, \geqq)}.$$

The quantities $\Pr(x > y)$ received much attention recently [Gr], [Sh], [GYY2], [KS]. We want to make:

CONJECTURE 2. *There is a universal constant $\frac{1}{2} > \alpha > 0$ such that if $(P, \geqq)$ is a finite poset in which the order $\geqq$ is not total, then there exists $x, y \in P$ such that*

$$1 - \alpha \geqq \Pr(x > y) \geqq \alpha.$$

In fact we know of no counterexample even for $\alpha = \frac{1}{3}$.

Now we can state and prove our main results. We can show the validity of conjectures 1 and 2 in the case where $(P, \geqq)$ can be covered by two chains. This special case is well known as the *merging problem*, see [Kn]. One is given two linearly ordered lists $A = (\alpha_1 > \cdots > \alpha_m)$ and $B = (b_1 > \cdots > b_n)$ and some order relations between elements of A and elements of B. We want to merge A and B into one ordered list where the linear order on $A \cup B$ is an extension of the partial order just described. So we have:

THEOREM 1. *Any algorithm which can merge A and B will require $\log_2 N_0$ steps in the worst case where N_0 is the number of extensions of the partial order on $A \cup B$. An algorithm exists which merges A and B in no more than $C_1 \log_2 N_0$ where $C_1 = (\log_2((1+\sqrt{5})/2))^{-1}$. This bound is best possible. The computation needed for finding the appropriate queries can be done in time polynomial in $|A \cup B|$.*

THEOREM 2. *With A, B as above one can always find $x \in A$, $y \in B$ for which*

$$\tfrac{2}{3} \geqq \Pr(x > y) \geqq \tfrac{1}{3}.$$

The constants $\frac{1}{3}$, $\frac{2}{3}$ are best possible. The elements x, y can be found in time polynomial in $|A \cup B|$.

Let us start with

Proof of Theorem 2. Let us show first why $\frac{1}{3}$, $\frac{2}{3}$ are best possible. Consider the case where $A = (a_1 > \cdots > a_m)$, $B = (b_1 > \cdots > b_{2m})$, $a_j > b_{2j+1}$ $(m-1 \geqq j \geqq 1)$ and $b_{2j-2} > a_j$ $(m \geqq j \geqq 2)$. It is easily verified that

$$\Pr(a_j > b_k) = \begin{cases} 0, & k \leqq 2j-2, \\ \frac{1}{3}, & k = 2j-1, \\ \frac{2}{3}, & k = 2j, \\ 1, & k \geqq 2j+1, \end{cases} \quad (m \geqq j \geqq 1, 2m \geqq k \geqq 1).$$

Now let us turn to the proof of the existence of $x \in A$, $y \in B$ for which $\frac{2}{3} \geqq \Pr(x > y) \geqq \frac{1}{3}$. We may assume w.l.o.g. that a_1 and b_1 are incomparable. If $a_1 > b_1$, say, then a_1 is the unique maximal element in $A \cup B$ and so it remains the maximal element in any extension of the partial order. Therefore, nothing will change if a_1 is deleted from the poset. We prove our claim by contradiction and we assume again

w.l.o.g. that

$$\Pr(a_1 > b_1) < \tfrac{1}{3}.$$

Define now the following quantities

$$q_1 = \Pr(a_1 > b_1),$$
$$q_i = \Pr(b_{i-1} > a_1 > b_i)(n \geqq i \geqq 2),$$
$$q_{n+1} = \Pr(b_n > a_1).$$

We prove the following:

LEMMA. *The real numbers* $q_i(n+1 \geqq i \geqq 1)$ *satisfy*:

1) $\frac{1}{3} \geqq q_1 \geqq \cdots \geqq q_{n+1} \geqq 0$,

2) $\sum_{t=1}^{n+1} q_i = 1$.

Proof. Since $q_1, \cdots, q_{n+1}$ is a probability distribution, all we have to show is that $q_1 \geqq \cdots \geqq q_{n+1}$. To show this we exhibit a 1:1 mapping from the event whose probability is q_{i+1} into the event with probability $q_i(1 \geqq i \geqq n)$. Notice that in an extension for which $b_{i-1} > a_1 > b_i$ not only does a_1 come after b_{i-1} but it must immediately follow it: Of course none of the a_j can precede a_1 and none of the b_j can come between b_{i-1} and b_i. The mapping from those extensions in which a_1 immediately follows b_i to those where $b_{i-1} > a_1 > b_i$ is obtained by permuting a_1 and b_{i-1}. This mapping clearly is well defined and 1:1.

The theorem can be proved now: let r be defined by

$$\sum_{i=1}^{r-1} q_i \leqq \tfrac{1}{2} < \sum_{i=1}^{r} q_i.$$

Since $\sum_{i=1}^{r-1} q_i = \Pr(a_1 > b_{r-1}) \leqq \frac{1}{2}$, it follows that $\sum_{i=1}^{r-1} q_i < \frac{1}{3}$. Similarly $\sum_{i=1}^{r} q_i = \Pr(a_1 > b_r)$ must be $> \frac{2}{3}$. Therefore $q_r > \frac{1}{3}$, but this contradicts $\frac{1}{3} > q_1 \geqq q_r$.

Complexity. The last claim of the theorem reduces now to proving that the index r of the above proof can be found in time which is polynomial in $|A \cup B|$. The reader should be aware that two separate complexity measures are being considered: the main one is a count of the number of queries that have to be asked in order to solve the merging problem, and the other one, which we address now, is the time complexity of the computations which are required to design the queries. Given a partially ordered set on n elements $(P, \geqq)$ which can be covered by two chains, there is a determinant formula giving the number of extensions of $\geqq$, see [Mo, p. 32]. Since these determinants are computable in polynomial time and we need to compute polynomially many such determinants to implement our algorithm, this proves our assertion. For completeness, let us recite the determinant counting formula: Let $P = A \cup B$, where $A = (a_1 > \cdots > a_m)$, $B = (b_1 > \cdots > b_n)$, and assume $m \geqq n$. Define integers $\alpha_1, \cdots, \alpha_m, \beta_1, \cdots, \beta_m$ as follows: $\beta_j = \min\{t | a_j < b_t\}$, $\alpha_j = \max\{t | b_t > a_j\}$ and where the minimum and maximum of an empty set are taken to be $n+1$ and zero respectively. The number of extensions of $(P, \geqq)$ is given by

$$\det\left[\binom{\beta_i - \alpha_j + 1}{j - i + 1}\right]_{m \geqq i, j \geqq 1}$$

see [Mo] for the details. □

The following theorem is equivalent with Theorem 1 but states the result in a more convenient way. We remind the reader about the definition of Fibonacci numbers: This is the sequence defined by: $F_0 = 1$, $F_1 = 2$, $F_{n+1} = F_n + F_{n-1}(n \geqq 1)$. The following

explicit formula also exists for these integers

$$F_n = A\lambda^n + B\cdot(-\lambda)^{-n},$$

where

$$\lambda = \frac{\sqrt{5}+1}{2}, \quad A = \frac{5+3\sqrt{5}}{10}, \quad B = \frac{5-3\sqrt{5}}{10}.$$

THEOREM 1.1. *A merging problem which cannot be solved by less than n queries must have at least F_n compatible solutions. For each $n \geqq 1$ there exists a unique merging problem which requires n queries and has exactly F_n compatible solutions The appropriate queries can be found in time polynomial in the size of the poset.*

Proof. Let us start by exhibiting the extreme cases. We describe the merging problems which are referred to as the *special merging problems. For $n = 2m-1$, let $A = (a_1 > \cdots > a_m)$, $B = (b_1 > \cdots > b_m)$ and* the relations $a_j > b_{j+1}(m-1 \geqq j \geqq 1)$, $b_k > a_{k+2}(m-2 \geqq k \geqq 1)$. For $n = 2m$ let $A = (a_1 > \cdots > a_{m+1})$, $B = (b_1 > \cdots > b_m)$ and $a_j > b_{j+1}(m-1 \geqq j \geqq 1)$, $b_k > a_{k+2}(m-1 \geqq k \geqq 1)$. In either case a_j is incomparable with only b_{j-1} and b_j. Whenever $a_j : b_j$ are compared, the answer is $a_j > b_j$ and the answer on $a_j : b_{j-1}$ is $b_{j-1} > a_j$. These answers supply no further information on incomparable pairs: therefore all n queries have to be made to solve these merging problems. To show that the number of compatible solutions in these merging problems are given by Fibonacci numbers, let us consider the case $n = 2m$. We split the compatible solutions into two parts according to whether $a_1 > b_1$ or $b_1 > a_1$. If $a_1 > b_1$, then a_1 is the unique maximal element and so can be deleted altogether. For the rest of the elements we make the following renaming $b_i' = a_{i+1}(i = 1, \cdots, m) a_i' = b_i(i = 1, \cdots, m)$ which shows that the remaining problem is the special problem for $n = 2m-1$. If $a_1 < b_1$, then a_1, b_1 are the maximal elements of the poset so they can be deleted. The remaining problem is again the special one for $n = 2m-2$. We have thus shown that $F_n = F_{n-1} + F_{n-2}$ for even $n \geqq 2$. The rest of the details can be easily filled in by the reader.

Now we turn to the actual proof of the theorem and of the uniqueness of the special problems: We'll show that if a merging problem is given with $N_0 \leqq F_n$ compatible solution and n steps are needed to solve it, then the problem is special. For $n \leqq 3$ the cases are few and can be checked each in itself. The general case is done by induction on n. Without loss of generality we assume that $q_1 = \Pr(a_1 > b_1) \leqq \frac{1}{2}$. As in the lemma we define q_i to be $\Pr(b_{i-1} > a_1 > b_i)(m+1 \geqq i \geqq 1)$. Consider the index r for which

$$\Pr(a_1 > b_{r-1}) = \sum_{i=1}^{r-1} q_i \leqq \tfrac{1}{2} < \sum_{i=1}^{r} q_i = \Pr(a_1 > b_r).$$

If $\Pr(a_1 > b_r) < F_{n-1}/F_n$, then comparing $a_1 : b_r$ we remain with a problem which has less than F_{n-1} compatible solutions and so can be solved in $n-2$ steps, contradiction. Similarly if $\Pr(a_1 > b_{r-1}) > F_{n-2}/F_n$, then on comparing $a_1 : b_{r-1}$ we remain with a problem with less than $F_n - F_{n-2} = F_{n-1}$ compatible solutions and the same argument applies. If follows, therefore, that $q_r \geqq F_{n-1}/F_n - F_{n-2}/F_n = F_{n-3}/F_n$. This implies now that $r \leqq 2$, because otherwise

$$\frac{F_{n-2}}{F_n} \geqq \sum_{1}^{r-1} q_i \geqq q_1 + q_2 \geqq 2q_r \geqq \frac{2F_{n-3}}{F_n},$$

a contradiction if $n \geqq 4$. On the other hand $r \neq 1$ because, by assumption $q_1 = \Pr(a_1 > b_1) \leqq \frac{1}{2}$.

So $r = 2$, $q_1 \leqq F_{n-2}/F_n$, $q_1 + q_2 \geqq F_{n-1}/F_n$. Make the comparison $a_1 : b_2$, to which we may assume the answer is $a_1 > b_2$. This is followed by the comparison $a_1 : b_1$ to which

we may assume a reply $a_1 > b_1$. The remaining problem has at most F_{n-2} compatible solutions and so can either be solved in $n-3$ queries making up a total of $n-1$ queries for the original problem, or else it is the special problem with F_{n-2} compatible solutions. One has to verify now that the problem we started with is special. This is an easy fact to verify and the details are omitted. The complexity argument is the same as in Theorem 2. □

3. Open problems. The major problem is, of course, to show that Theorems 1 and 2 hold for general sorting problems. These problems were stated above in conjectures 1, 2. To state other problems let us make the following definition: An extension of a partial order $(P, \geqq)$ can be described as 1:1 order-preserving map $\sigma: P \to \{1, \cdots, |P|\}$. For $x \in P$ we define $h(x)$ to be the average of $\sigma(x)$ over all extensions σ of $(P, \geqq)$. Let $|P| = n$ be the order of the poset, then $\sum_{x \in P} h(x) = n(n+1)/2$. We define the "second moment" of h as $V(P) = \sum_{x \in P} h^2(x)$. If $p, q \in P$ are incomparable elements, then denote by $P(p, q)$ the poset which is obtained by adding the relation $p > q$ to P (and, of course, taking transitive closure of the new relation). We have:

THEOREM 3. *Let P be a poset, $p, q \in P$ incomparable elements. Then*

$$V(P) \leqq \max \{V(P(p, q)), V(P(q, p))\}.$$

Proof. The most convenient way to view this inequality is geometrically: To any poset $(P, \geqq)$ we canonically assign an n-dimensional convex polyhedron $C(P)$ where $|P| = n$. The assignment is as follows: If P has no order relations, then $C(P)$ is the unit cube $\{(x_1, \cdots, x_n) | 1 \geqq x_i \geqq 0\}$. Let us say that $C(P)$ has been defined for posets with k order relations or less ($k \geqq 0$). Then on introducing the new relation $p_i > p_j$ the convex polytope of the new poset $P(p_i, p_j)$, namely $C(P(p_i, p_j))$, is obtained by taking that part of $C(P)$ which lies in the half-space $x_i > x_j$. Accordingly, for P which is totally ordered, $C(P)$ is a simplex $x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}$. Notice that these simplices have volume $1/n!$ each, and that if $(P, \geqq)$ is any partial order on $P = \{p_1, \cdots, p_n\}$, then there is a 1:1 correspondence between the extensions of $(P, \geqq)$ and the simplices that make up $C(P)$. In particular the volume of $C(P)$ equals $1/n!$ times the number of extensions of $(P, \geqq)$. Notice also that since all these simplices have equal volume, $\bar{h}(P) = 1/(n+1)(h(p_1), \cdots, h(p_n))$ is the center of gravity of $C(P)$. If follows that $V(P)$ is the square of the distance from the center of gravity of $C(P)$ to the origin. Now that we have established the geometric interpretation of $V(P)$, the validity of the theorem follows at once: $C(P)$ is the disjoint union of $C(P(p_i, p_j))$ and $C(P(p_j, p_i))$. Therefore the origin and the centers of gravity for $C(P(p_i, p_j))$ and $C(P(p_j, p_i))$ form a triangle and the center of gravity of $C(P)$ lies on the edge connecting the two centers of gravity. The theorem now follows from obvious facts of plane geometry. □

Now that we have established Theorem 3, we are ready to ask if a stronger statement holds.

CONJECTURE 3. *Let P be a poset and let $p, q \in P$ be incomparable. Then*

$$V(P(p, q)) \geqq V(P).$$

See the problem session of [OS, p. 806] for a related discussion.

Note added in proof. Problem 1 has been recently answered affirmatively by the author and M. Saks. The constant that was found is $\alpha = \frac{1}{4}(3 - \log_2 5)$.

An interesting problem in computational complexity is to show that it is hard to count the number of linear extensions of a finite poset. We conjecture that this problem is #P-complete. This conjecture has apparently been made also by R. Karp and by some other researchers. Using the construction made in the proof of Theorem 3, this

conjecture could be a concrete statement to the effect that evaluating the volumes of polyhedra is a hard computational problem.

Also, counting the number of order ideals in posets can be shown to be # P-complete. This was shown also by R. Karp (private communication, March 1983).

It has been brought to our attention that Conjecture 2 has been independently made by a number of researchers, some time ago. In particular we know that M. Fredman and R. Stanley had thought about it.

REFERENCES

[Fr] M. FREDMAN, *How good is the information theory bound in sorting*, Theoret. Comput. Sci., 1 (1976), pp. 355–361.

[Gr] R. L. GRAHAM, *Linear extensions of partial orders and the* FKG *inequality*, in [OS], pp. 213–236.

[GYY1] R. L. GRAHAM, A. C. YAO AND F. F. YAO, *Information bounds are weak in the shortest distance problem*, J. Assoc. Comput. Mach., 27 (1980), pp. 428–444.

[GYY2] ———, *Some monotonicity properties of partial orders*, SIAM J. Alg. Disc. Meth., 1 (1980), pp. 251–258.

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

[KS] D. J. KLEITMAN AND J. B. SHEARER, *A monotonicity property of partial order*, to appear.

[KLS] D. J. KLEITMAN, N. LINIAL AND D. STURTEVANT, *On extremal spanning trees*, unpublished manuscript.

[LS] N. LINIAL AND M. SAKS, *Searching ordered structures*, J. Algorithms, to appear.

[Mo] S. G. MOHANTY, *Lattice Path Counting and Applications*, Academic Press, New York, 1979.

[OS] *Ordered Sets*, I. Rival, ed., NATO Advanced Study ser. C. vol. 83, 1981.

[Sa] B. SANDS, *Counting antichains in finite partially ordered sets*, Discrete Math., 35 (1981), pp. 213–228.

[Sh] L. A. SHEPP, *The* FKG *inequality and some monotonicity properties of partial orders*, SIAM J. Alg. Disc. Meth., 1 (1980), pp. 295–299.

SIAM J. COMPUT.
Vol. 13, No. 4, November 1984

© 1984 Society for Industrial and Applied Mathematics
010

PARALLEL ALGORITHMS FOR ALGEBRAIC PROBLEMS*

JOACHIM VON ZUR GATHEN†

Abstract. Fast parallel algorithms are presented for the following problems in symbolic manipulation of univariate polynomials: computing all entries of the extended Euclidean scheme of two polynomials over an arbitrary field, gcd and lcm of many polynomials, factoring polynomials over finite fields, and the squarefree decomposition of polynomials over fields of characteristic zero and over finite fields.

For the following estimates, assume that the input polynomials have degree at most n, and the finite field has p^d elements. The Euclidean algorithm is deterministic and runs in parallel time $O(\log^2 n)$. All the other algorithms are probabilistic (Las Vegas) in the general case, but when applicable to $\mathbf{Q}$ or $\mathbf{R}$, they can be implemented deterministically over these fields. The algorithms for gcd and lcm use parallel time $O(\log^2 n)$. The factoring algorithm runs in parallel time $O(\log^2 n \log^2 (d+1) \log p)$. The algorithm for squarefree decomposition runs in parallel time $O(\log^2 n)$ for characteristic zero, and in parallel time $O(\log^2 n + (d-1) \log p)$ for finite fields. All Las Vegas algorithms have failure probability less than 2^{-n}. For all algorithms, the number of processors is polynomial in n.

Key words. parallel processing, algebraic computing, symbolic manipulation, Euclidean algorithm, factorization of polynomials, squarefree decomposition

1. Introduction. In Borodin–von zur Gathen–Hopcroft [1982] the following program is laid out: obtain a "theory package for parallel algebraic computations," i.e. fast parallel computations for the widely used problems of symbolic manipulation in an algebraic context. In that paper, two basic problems were considered: solving systems of linear equations and computing the gcd of polynomials, both over arbitrary ground fields.

The present paper continues this program, and fast parallel solutions to the following algebraic problems are given: computing all entries of the extended Euclidean scheme of two polynomials over an arbitrary field, computing the gcd and lcm of many polynomials over an arbitrary field, factoring polynomials over finite fields, and the squarefree decomposition of polynomials over fields of characteristic zero and over finite fields.

As our model of parallel computation, we can take an algebraic PRAM (with instructions $+$, $-$, $*$, $/$, constants) or the parallel algebraic computation (directed acyclic) graphs, PACDAG for short, which we describe informally below. We will describe the algorithms of this paper in "high-level language," and not give actual implementations on a PACDAG. A more formal description would follow the lines of the discussion in Strassen [1983] of (one-processor) algebraic computation trees and the collections that they compute.

A PACDAG has two kinds of processors and (shared) variables, "arithmetic" and "boolean" ones. At each node of the (rooted) directed acyclic computation graph, each arithmetic processor can either perform an arithmetic operation ($+$, $-$, $*$, $/$) on two arithmetic variables, or access an arithmetic input variable, or fetch a constant from the ground field. Each boolean processor can either compute the negation or conjunction of (one resp. two) boolean variables, or it can take an arithmetic variable x and set a boolean variable to "true" if $x \neq 0$, and to "false" otherwise. (One can in fact simulate these boolean computations in the ground field if a conditional division instruction of the form "if $x \neq 0$ then $y = 1/x$" is allowed.) No write-conflicts are

* Received by the editors February 8, 1983, and in revised form August 12, 1984. An extended abstract of this paper appeared in Proc. 15th ACM Symposium on Theory of Computing, Boston, 1983, pp. 17–23.

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

allowed. At each node of the graph, each of the branches of the graph emanating from that node is labeled with a boolean variable; if on a given input the computation reaches that node and exactly one of these boolean variables is true, then the corresponding branch is chosen; otherwise the computation is not defined at that input.

At each leaf, the output is given by a sequence of arithmetic variables. Thus a PACDAG computes a collection in the sense of Strassen [1983].

Any PACDAG can obviously be described by a string over a finite alphabet, provided the field constants used can be described in this way. A family $(P_n)_{n\in\mathbf{N}}$ of PACDAGs is uniform if the description of P_n can be generated by a deterministic Turing machine in space $O(\log n)$, given n in unary as input. (See Ruzzo [1981], Cook [1983].) All algorithms of this paper are uniform.

We are concerned with two cost measures of a PACDAG: its parallel time (= depth of the graph = length of longest path from root to any leaf) and its size (= number of processors). All the algorithms of this paper work in parallel time $\log^{O(1)} n$ (usually $O(\log^2 n)$) and use $n^{O(1)}$ processors, where n is the input size. One additional feature is needed for some algorithms: a random generator that produces elements of a finite subset of the ground field at random, i.e. new "random variables" from which these random elements can be read.

Among the problems for which we do not have a fast parallel algorithm are the gcd of two integers and the factorization of a polynomial with rational coefficients. We give a reduction from the first problem to a problem related to the second, namely computing short vectors in integer lattices.

All our results belong to the "asymptotic approach" to parallel algorithms, where one is interested in obtaining the fastest possible parallel algorithms, allowing an almost arbitrary (say, polynomially bounded) number of processors. This contrasts with the approach of getting a speed-up from sequential to parallel time close to the number of processors: $T_{\text{seq}}/T_{\text{par}}$ is approximately the number of processors.

One interesting phenomenon occurs: all the problems eventually reduce to solving systems of linear equations. Hence the paramount role of the latter problem. On the theoretical level, this is expressed to a certain extent by Valiant's [1980] universality of the determinant (disallowing branching and division).

The algorithms for linear algebra in Borodin–von zur Gathen–Hopcroft [1982] used a polynomial number of processors, but this number was impractically large. Soon after, Berkowitz [1984] presented an algorithm for computing the determinant with parallel time $O(\log^2 n)$ and $O(n^{3.5})$ processors.

2. The extended Euclidean scheme for polynomials. We start with an easy result about division with remainder of polynomials.

LEMMA 2.1. *One can compute the quotient and remainder of two polynomials of degree at most n in parallel time $O(\log^2 n)$.*

Proof. Let f, $g \in F[x]$ be given, where F is an arbitrary field, and $k = \deg f - \deg g + 1 \leq n$. Their quotient $q \in F[x]$ is uniquely determined by the condition $\deg(f - qg) < \deg g$, which can be expressed by a nonsingular system of k linear equations in the k coefficients of q. This system can be solved in parallel time $O(\log^2 k)$, as in Borodin–von zur Gathen–Hopcroft [1982]. Computation of the remainder then takes $O(\log n)$ parallel steps. □

Actually, quotient and remainder can be computed in parallel time $O(\log n)$. This was shown by Reif [1983] under the assumption that F supports a fast Fourier transform, and by Eberly [1984] in general.

Let f, $g \in F[x]$ with $0 \leqq \deg g = m \leqq n = \deg f$, where F is an arbitrary field. We set $a_0 = f$, $a_1 = g$ and consider the extended Euclidean scheme for (f, g):

$$
\begin{array}{ll}
a_0 = q_1 a_1 + a_2, & s_2 a_0 + t_2 a_1 = a_2, \\
\vdots & \vdots \\
a_{l-2} = q_{l-1} a_{l-1} + a_l, & s_{l-1} a_0 + t_{l-1} a_1 = a_{l-1}, \\
a_{l-1} = q_l a_l, & s_l a_0 + t_l a_l = a_l.
\end{array}
$$

where the following conditions are satisfied for $2 \leqq k \leqq l$: a_k, q_1, q_k, s_k, $t_k \in F[x]$, $\deg a_k < \deg a_{k-1}$, $s_0 = 1$, $t_0 = 0$, $s_1 = 0$, $t_1 = 1$, $s_k = s_{k-2} - q_{k-1} s_{k-1}$ and $t_k = t_{k-2} - q_{k-1} t_{k-1}$. Thus the q's are the quotients and the a's the remainders of Euclid's algorithm, gcd (f, g) is the unique monic scalar multiple of a_l by convention, all gcd's of polynomials in $F[x]$ are monic) and the s's and t's are the "cumulants", "convergents" or "continuants". (The terminology is rather unsatisfactory. The last term, proposed by a certain Muir, provoked an amusing controversy (see Muir [1878]).) Sequential algorithms for computing these polynomials are important and well-studied; see Knuth [1981, 4.6.1], for an overview. One interesting feature is that $q_1, \cdots, q_l$ can be computed faster than can $a_1, \cdots, a_l$ (see Strassen [1983]).

The above conditions imply that a_k has "small" degree and is a linear combination of f and g with coefficients s_k and t_k. Obviously one can multiply a_k, s_k, t_k by a polynomial of small degree and conserve these properties. The following lemma shows that this is the only way of obtaining these properties, and thus gives a characterization of a_k, s_k, t_k. Our proof follows Kronecker [1881]; see Knuth [1981, Exercise 4.6.1–26], for a different approach.

LEMMA 2.2. *Let f, g, a_k, s_k, $t_k \in F[x]$ be as above, and a, s, $t \in F[x]$ with a, $t \neq 0$. Then the following two conditions are equivalent*:

(i) $sf + tg = a$, *and* $\deg a + \deg t < n$.

(ii) *There exist $k \in \{1, \cdots, l\}$ and $b \in F[x]$ such that*

$$a = b a_k, \quad s = b s_k, \quad t = b t_k,$$
$$\deg a_k \leqq \deg a < (\deg a_{k-1} + \deg a_k)/2 < \deg a_{k-1}.$$

Furthermore, if the conditions are satisfied, then k and b in (ii) *are uniquely determined.*

Proof. By induction on k one proves that $\deg t_k = \sum_{1 \leqq i < k} \deg q_i$ and $\deg t_k + \deg a_{k-1} = n$, and then (i) obviously follows from (ii).

For the other implication, we define $k \in \{1, \cdots, l\}$ by

$$\deg a_k \leqq \deg a < \deg a_{k-1}.$$

This determines k uniquely, since

$$\deg a_l < \deg a_{l-1} < \cdots < \deg a_0,$$
$$\deg a_l = \deg \gcd (f, g) \leqq \deg a < n = \deg a_0.$$

Eliminating g from

$$s_k f + t_k g = a_k, \qquad sf + tg = a,$$

we get

$$(s_k t - s t_k) f = t a_k - t_k a,$$
$$\deg (t a_k - t_k a) \leqq \max \{n - 1 - \deg a + \deg a_k, n - \deg a_{k-1} + \deg a\} < n.$$

Since $\deg f = n$, we conclude that $s_k t - s t_k = 0$. By induction on k one easily sees that

$$s_k t_{k-1} - t_k s_{k-1} = (-1)^k$$

for $1 \leqq k \leqq l$, hence in particular $\gcd(s_k, t_k) = 1$. This together with $s_k t = s t_k \neq 0$ implies that there exists $b \in F[x]$ such that

$$t = b t_k.$$

Then also

$$\begin{aligned} &s = b s_k, \quad a = b a_k, \quad 0 \leqq \deg b, \\ &n > \deg a + \deg t = 2 \deg b + \deg a_k + \deg t_k \\ &\qquad = 2 \deg b + \deg a_k + n - \deg a_{k-1}, \\ &2 \deg a = 2 \deg b + 2 \deg a_k \\ &\qquad < \deg a_{k-1} - \deg a_k + 2 \deg a_k = \deg a_{k-1} + \deg a_k. \end{aligned}$$

□

Remark. If $a = 0$ then the statement of Lemma 2.2 becomes valid by introducing the following (natural) notation: $a_{l+1} = 0$, $s_{l+1} = s_{l-1} - q_l s_l$, $t_{l+1} = t_{l-1} - q_l t_l$. We have to allow $k = l+1$ in (ii), and interpret arithmetic expressions involving $\deg a = -\infty$ "correctly."

The theory of subresultants (Collins [1967], Brown–Traub [1971]) provided an important development for sequential algorithms computing gcd's of polynomials (or, more generally, for the entries of the extended Euclidean scheme). Euclid's algorithm for the gcd can suffer from exponential intermediate expression swell if the coefficients of the polynomials are integers or polynomials themselves (Brown [1971]). The subresultant algorithms avoid this difficulty—which makes the algorithm decidedly impractical—by translating the problem into systems of linear equations. For parallel algorithms, this strategy of employing linear equations was successfully exploited for the gcd in Borodin–von zur Gathen–Hopcroft [1982], and here we use it to compute all entries of the extended Euclidean scheme.

We first give a self-contained exposition of the relevant results about subresultants. For this simplified version with Lemma 2.2 as the cornerstone we only have to introduce the "principal subresultants" as follows.

We write $f = f_n x^n + \cdots + f_0$, $g = g_m x^m + \cdots + g_0$ with $0 \leqq m \leqq n$ and $f_n g_m \neq 0$. For $0 \leqq i \leqq m$ we consider the $(n+m-2i) \times (n+m-2i)$-submatrix P_i of the Sylvester matrix of (f, g) which consists of the first $m-i$ columns of f_j's and the first $n-i$ columns of g_j's:

$$P_i = \begin{bmatrix} f_n & & & g_m & & \\ f_{n-1} & f_n & & g_{m-1} & g_m & \\ \vdots & \ddots & & \vdots & \ddots & \\ f_{n-m+i+1} & \cdots & f_n & g_{m-n+i+1} & \cdots & g_m \\ \vdots & & \vdots & \vdots & & \vdots \\ f_{2i-m+1} & \cdots & f_i & g_{2i-n+1} & \cdots & g_i \end{bmatrix}.$$

Thus P_0 is the Sylvester matrix of (f, g), and $\det(P_0)$ their resultant. One might call P_i a "principal subresultant" since its rows are the highest among the matrices for subresultants of the same size. We denote by c_k the leading coefficient of a_k.

THEOREM 2.3. *Using the above notation, we have for all i, $1 \leqq i \leqq m$:*

(i) $\exists k \in \{1, \cdots, l\}$ $\deg a_k = i \Leftrightarrow \det(P_i) \neq 0$.

(ii) *If* $\deg a_k = i$ *and* $y_0, \cdots, y_{m-i-1}, z_0, \cdots, z_{n-i-1} \in F$ *are such that*

$$P_i \cdot \begin{bmatrix} y_{m-i-1} \\ \vdots \\ y_0 \\ z_{n-i-1} \\ \vdots \\ z_0 \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

then

$$s_k = c_k(y_{m-i-1}x^{m-i-1} + \cdots + y_0),$$

$$t_k = c_k(z_{n-i-1}x^{n-i-1} + \cdots + z_0).$$

Proof. Let $e_i = (0, \cdots, 0, 1) \in F^{m+n-2i}$, and for any $y_0, \cdots, y_{m-i-1}$, $z_0, \cdots, z_{n-i-1} \in F$ let

$$s = y_{m-i-1}x^{m-i-1} + \cdots + y_0,$$

$$t = z_{n-i-1}x^{n-i-1} + \cdots + z_0,$$

where $1 \leqq i \leqq m$. By abuse of notation, we write $P_i(s, t) = e_i$ for the system of linear equations as in (ii). Obviously for any i, $1 \leqq i \leqq m$, we have

$\det(P_i) \neq 0 \Leftrightarrow P_i(s, t) = e_i$ has exactly one solution

$\Leftrightarrow$ there exist unique $s, t \in F[x]$ such that $\deg s < m - i$, $\deg t < n - i$ and $sf + tg$ is a monic of degree i.

Using Lemma 2.2, we will show that the latter condition is satisfied iff there exists $k \in \{1, \cdots, l\}$ such that $\deg a_k = i$. Note that for any i with $\deg a_l \leqq i \leqq m$ there exists $k \in \{1, \cdots, l\}$ such that $\deg a_k \leqq i < \deg a_{k-1}$. We distinguish four cases.

Case 1. $\exists k \in \{1, \cdots, l\}$ such that $\deg a_k = i$. Setting $s = c_k^{-1}s_k$, $t = c_k^{-1}t_k$ we see that $sf + tg$ is monic of degree i. It follows from Lemma 2.2 that s, t are uniquely determined. This proves (i) in this case, and also the statement (ii).

Case 2. $\exists k \in \{1, \cdots, l\}$ such that $\deg a_k < i < (\deg a_k + \deg a_{k-1})/2$. For any $b \in F[x]$ of degree $i - \deg a_k$ and with leading coefficient c_k^{-1}, set

$$s(b) = bs_k, \quad t(b) = bt_k, \quad a(b) = ba_k.$$

Then $\deg(s(b)) < m - i$, $\deg(t(b)) < n - i$ and $a(b) = s(b)f + t(b)g$ is monic of degree i. Since there exists more than one such b, (i) is proven in this case.

Case 3. $\exists k \in \{1, \cdots, l\}$ such that $(\deg a_k + \deg a_{k-1})/2 \leqq i < \deg a_{k-1}$. Assume that $\deg s < m - i$, $\deg t < n - i$ and $a = sf + tg$ has degree i. From Lemma 2.2, we get $i < (\deg a_k + \deg a_{k-1})/2$. Hence no such s, t exist, and (i) is proven in this case.

Case 4. $i < \deg a_l$. No s, t satisfy the condition, since no nonzero polynomial of degree less than $\deg a_l = \deg(\gcd(f, g))$ is a linear combination of f and g. (This can be interpreted as Case 3 with $k = l + 1$.) □

We can now state the parallel algorithm that computes all entries of the extended Euclidean scheme of two polynomials.

ALGORITHM EUCLID.

Input: The coefficients of $f, g \in F[x]$ with $\deg g = m \leqq n = \deg f$.

Output: The coefficients of polynomials A_k, Q_k, S_k, T_k for $1 \leqq k \leqq l$, which are the entries of the extended Euclidean scheme of (f, g).

1. For all i, $0 \leqq i \leqq m$, compute $p_i = \det(P_i)$, where P_i is the ith principal subresultant as above.
2. Set

$$\{n_1, \cdots, n_l\} = \{i: 0 \leqq i \leqq m \text{ and } p_i \neq 0\}$$

with $m = n_1 > n_2 > \cdots > n_l$. ($n_k$ will be $\deg a_k$.)

3. For all k, $1 \leqq k \leqq l$, and $i = n_k$ compute $y_0, \cdots, y_{m-i-1}, z_0, \cdots, z_{n-i-1} \in F$ such that

$$P_i \cdot \begin{bmatrix} y_{m-i-1} \\ \vdots \\ y_0 \\ z_{n-i-1} \\ \vdots \\ z_0 \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ \vdots \\ 0 \\ 1 \end{bmatrix}.$$

(This is a nonsingular system of linear equations and has a unique solution.) Set

$$u_k = y_{m-i-1}x^{m-i-1} + \cdots + y_0,$$

$$v_k = z_{n-i-1}x^{n-i-1} + \cdots + z_0,$$

$$w_k = u_k f + v_k g.$$

(u_k, v_k, w_k will be scalar multiples of s_k, t_k, a_k. In steps 4 and 5 we compute these scalar factors.)

4. For all k, $2 \leqq k \leqq l$, compute d_k, $r_k \in F[x]$ such that

$$w_{k-2} = r_{k-1} w_{k-1} + d_k,$$

$$\deg d_k < \deg w_{k-1}.$$

(Dividing w_{k-2} by w_{k-1} with remainder. Use $w_0 = f$.)

5. For all k, $1 \leqq k \leqq l$, compute the following.

$$\delta_k = \text{leading coefficient of } d_k,$$

$$e_k = \begin{cases} \delta_k \delta_{k-2} \cdots \delta_2 & \text{if } k \text{ is even,} \\ \delta_k \delta_{k-2} \cdots \delta_1 & \text{if } k \text{ is odd,} \end{cases}$$

$$A_k = e_k w_k,$$

$$Q_k = (A_{k-1} - A_{k+1})/A_k,$$

$$S_k = e_k u_k,$$

$$T_k = e_k v_k.$$

(with δ_1 = leading coefficient of g, and $A_{l+1} = 0$.)

THEOREM 2.4. *Over any field, algorithm* EUCLID *computes the entries of the extended Euclidean scheme of* (f, g). *If* $\deg g \leqq \deg f \leqq n$, *then it can be performed in parallel time* $O(\log^2 n)$.

Proof. By Borodin–von zur Gathen–Hopcroft [1982] steps 1 and 3 can be performed in parallel time $O(\log^2 n)$. Using Lemma 2.1 for step 4 and the division in step 5, the timing estimate is clear. The proof of correctness below shows that the division in step 5 is exact.

It follows from Theorem 2.3(i) that

$$\#\{i: 0 \leqq i \leqq m \text{ and } p_i \neq 0\} = l = \text{length of Euclidean scheme},$$
$$(n_1, \cdots, n_l) = (\deg a_1, \cdots, \deg a_l),$$

and from Theorem 2.3(ii) that

$$s_k = c_k u_k, \quad t_k = c_k v_k, \quad a_k = c_k w_k,$$

for $1 \leqq k \leqq l$, where c_k is the leading coefficient of a_k. From

$$(c_{k-2} r_{k-1}) w_{k-1} + c_{k-2} d_k = c_{k-2} w_{k-2} = a_{k-2}$$
$$= q_{k-1} a_{k-1} + a_k = (c_{k-1} q_{k-1}) w_{k-1} + c_k w_k,$$
$$\deg (c_{k-2} d_k) < \deg w_{k-1},$$
$$\deg (c_k w_k) < \deg w_{k-1},$$

and the uniqueness of division with remainder we conclude that

$$c_{k-2} d_k = c_k w_k.$$

Since w_k is monic, we obtain

$$c_{k-2} \delta_k = c_k.$$

By induction on k it follows that $c_k = e_k$ and $a_k = A_k$. We also conclude that $q_k = Q_k$, $s_k = S_k$, $t_k = T_k$. □

Remark 2.5. Let us describe in some more detail how the branching in step 2 can be implemented on a PACDAG. Given the sequence $p_0, \cdots, p_m$, we want to store in variables q_{kj} $(0 \leqq k \leqq l,\ j \in \{1, \cdots, n\}^2)$ the matrices $\tilde{P}_i$, where $\tilde{P}_i$ is P_i extended by 1's on the diagonal to an $n \times n$-matrix, such that $Q_k = \tilde{P}_{n_k}$ for $1 \leqq k \leqq l$.

Using a binary splitting (starting with all values $0 \leqq u = v \leqq m$ and ending with $u = 0$, $v = m$), we compute larger and larger intervals $[u, \cdots, v]$ such that if $s = \#\{i: u \leqq i \leqq v \text{ and } p_i \neq 0\} - 1$, then $Q_u, \cdots, Q_{u+s}$ are the $\tilde{P}_i$ with $p_i \neq 0$ in the original order, and $Q_{u+s+1}, \cdots, Q_v = 0$. Let q_k be p_i for the i corresponding to k.

Given two such intervals $[u, \cdots, v]$ and $[v+1, \cdots, w]$, we first branch in depth $O(\log n)$ according to the value of s, $u \leqq s \leqq v$, such that $q_s \neq 0$ and $q_{s+1} = 0$ (or ($s = u - 1$ and $q_u = 0$) or ($s = v$ and $q_v \neq 0$)), and similarly for t, $v < t \leqq w$. Then set

$$Q_k = \begin{cases} Q_k & \text{if } u \leqq k \leqq s, \\ Q_{k+v-s} & \text{if } s < k \leqq s + t - v, \\ 0 & \text{if } s + t - v < k \leqq w. \end{cases}$$

Remark 2.6. As mentioned above, the entries of the Euclidean scheme of integer polynomials may be very large. However, the u_k, v_k, w_k computed in the algorithm will have reasonably small coefficients (with binary length polynomial in the input length, by Edmonds [1967]). Hence, as long as one wants a_k, q_k, s_k, t_k only up to a scalar multiple, steps 1, 2, 3 provide a solution with small coefficients (using the scalar multiple $(w_{k-1} - \delta_{k+1} w_{k+1})/w_k$ of q_k). This is quite satisfactory for all practical purposes. Only if we want to compute the entries of the extended Euclidean scheme exactly, then we have to perform steps 4 and 5 and may be faced with very large integers. A similar observation applies to multivariate polynomials

Remark 2.7. Algorithm EUCLID can be considered as an "NC^1-reduction" (i.e. one using only operations of parallel time $O(\log n)$) from the problem of computing

the extended Euclidean scheme to that of computing the determinant of matrices. In fact, all the algorithms of this paper can be considered as (probabilistic) NC^1-reductions to the determinant problem. (We have to assume that the field is fixed; see § 4 for a discussion of finite fields described in the input.) In later sections, algorithms will make use of the rank of matrices. The rank problem itself is NC^1-reducible to the determinant computation—probabilistically in general, and deterministically over any real field, in particular over $\mathbf{Q}$ or $\mathbf{R}$. Thus over $\mathbf{Q}$, all reductions are deterministic, but over an arbitrary ground field they are probabilistic.

Remark 2.8. If F is $\mathbf{Q}$ or a finite field, then we can consider the input as represented by bit strings, and ask for Boolean circuits, say, that compute the functions considered here in small depth and polynomial size. A basic result is that the determinant function is in Boolean NC^2 (Csanky [1976], Borodin–Cook–Pippenger [1983]). It follows from Remark 2.7 that all the computational problems considered in this paper are in NC^2 (again, provided that the ground field is fixed).

3. Gcd and lcm of many polynomials. In § 5 below, we want to compute the squarefree decomposition of polynomials. It turns out that we first have to present an ancillary result, which may be of independent interest: how to compute the gcd of many polynomials in parallel. Thanks go to Steve Cook for pointing out the algorithm below. We also need the least common multiple lcm of many polynomials, and give an algorithm for this problem.

Let F be an arbitrary field, $f_1, \cdots, f_n \in F[x]$ have degree at most n, and $g = \gcd(f_1, \cdots, f_n)$. We want to compute g from $f_1, \cdots, f_n$. It is easy to see that there exist $s_1, \cdots, s_n \in F[x]$ such that $\sum s_i f_i = g$. We claim that in addition one can have $\deg s_i < n$. To prove this claim, reorder the polynomials such that $\deg f_1 \geqq \deg f_i$ for all i. For each $i \geqq 2$, divide s_i by f_1 with remainder: $s_i = q_i f_1 + \bar{s}_i$, and $\deg \bar{s}_i < \deg f_1 \leqq n$. Set $\bar{s}_1 = s_1 + \sum_{i \geqq 2} q_i f_i$. Then $\sum_{1 \leqq i \leqq n} \bar{s}_i f_i = g$, and $\bar{s}_1 f_1 = g - \sum_{i \geqq 2} \bar{s}_i f_i$ has degree less than $n + \deg f_1$. Hence $\deg \bar{s}_i < n$ for all i, and the claim is proven.

Now let

$$d = \min\{\deg f : \exists s_1, \cdots, s_n \in F[x] \deg s_i < n \text{ for all } i \text{ and } f = \textstyle\sum s_i f_i \neq 0\}.$$

Then $d = \deg g$, by the claim above. We can now set up systems A_k of linear equations for $0 \leqq k \leqq n$, where A_k expresses "$\sum s_i f_i$ is monic of degree k." We write

$$f_i = \sum_{0 \leqq j \leqq n} f_{ij} x^j, \qquad s_i = \sum_{0 \leqq j < n} s_{ij} x^j,$$

and A_k consists of the linear equations

$$\sum_{\substack{1 \leqq i \leqq n \\ 0 \leqq j \leqq l}} s_{ij} f_{i,l-j} = \begin{cases} 0 & \text{for } k < l < 2n, \\ 1 & \text{for } l = k, \end{cases}$$

in the indeterminates s_{ij}. (Write zero whenever a subscript is out of range.) Thus A_k has $2n - k$ equations in at most n^2 variables. From the above we know that A_k has a solution iff $k \geqq d$. In particular, A_d has a solution, and from a solution of A_d we can easily compute g.

ALGORITHM GCD OF MANY POLYNOMIALS.
Input: a number $n \in \mathbf{N}$, and $f_1, \cdots, f_n \in F[x]$ with $\deg f_i \leqq n$ for all i.
Output: either $g = \gcd(f_1, \cdots, f_n)$ or "failure."

1. For all k, $0 \leqq k \leqq n$, determine whether A_k has a solution, and if it has, compute a solution $(s_{ij}(k))$ of A_k. (Using a Las Vegas algorithm in general, but a deterministic algorithm if F is real.)

2. Set $d=\min\{k: A_k \text{ has a solution}\}$.

3. Return $g=\sum_{\substack{1\leq i\leq n,\\ 0\leq j<n}} s_{ij}(d)x^j f_i$.

THEOREM 3.1. *Algorithm* GCD OF MANY POLYNOMIALS *either returns the* gcd g *of the input polynomials, or it reports "failure." The latter happens with probability less than* 2^{-n}. *If* F *is real, then the algorithm can be performed deterministically. The parallel time for the algorithm is* $O(\log^2 n)$.

Proof. From the discussion above it is clear that the algorithm correctly computes the (monic) gcd of the input polynomials. Steps 2 and 3 can be performed (deterministically) in parallel time $O(\log n)$. In step 1 we apply the Las Vegas algorithm of Borodin–von zur Gathen–Hopcroft [1982, Thm. 5(ii)]. This algorithm works in parallel time $O(\log^2 n)$. The coefficient matrix of A_k is considered to be of size $n^2\times n^2$, and the failure probability for each k is at most 2^{-n^2}. Thus the total failure probability in step 1 is not greater than $(n+1)2^{-n^2}<2^{-n}$.

If F is real, then we can use the deterministic version of the parallel algorithm for solving singular systems of linear equations. □

We have used n as a separate input rather than read it off the input polynomials in order to ensure failure probability less than 2^{-n} in applications where we do not know the exact degree of the input polynomials. We remark that by computing gcd's of pairs of polynomials along a binary tree, and using the algorithm from Borodin–von zur Gathen–Hopcroft [1982, Thm. 2], we get a deterministic algorithm for $\gcd(f_1,\cdots,f_n)$ over an arbitrary field running in parallel time $O(\log^3 n)$.

For two polynomials $f_1,f_2\in F[x]$ the relation $\gcd(f_1,f_2)\cdot \operatorname{lcm}(f_1,f_2)=f_1f_2$ holds, and both the gcd and lcm can be computed in parallel time $O(\log^2 n)$. For more than two input polynomials, one does not have such a simple relationship between gcd and lcm.

ALGORITHM LCM OF MANY POLYNOMIALS.
Input: A number $n\in \mathbf{N}$, and polynomials $f_1,\cdots,f_n\in F[x]$ of degree at most n.
Output: Either the monic least common multiple $g=\operatorname{lcm}(f_1,\cdots,f_n)$, or "failure."

1. Set $d_i=\deg f_i$, $m=\max_{1\leq i\leq n} d_i$, $s=\sum_{1\leq i\leq n} d_i$, and replace each f_i by its monic multiple $f_i/(\text{leading coefficient of } f_i)$.
2. For all k, $m\leq k\leq s$, do the following. Let B_k be the system of (inhomogeneous) linear equations that expresses
$$u_1f_1-u_2f_2=u_2f_2-u_3f_3=\cdots=u_{n-1}f_{n-1}-u_nf_n=0.$$
Here each $u_i=\sum_{0\leq j\leq k-d_i} u_{ij}x^j$ is a monic polynomial of degree $k-d_i$, and hence B_k consists of $(n-1)k$ linear equations in the $\sum (k-d_i)=nk-s$ indeterminate coefficients u_{ij} $(1\leq i\leq n, 0\leq j<k-d_i)$ of the u_i's. (Note that $(n-1)k\geq nk-s$.) Determine whether B_k has a solution, and compute a solution $u_{ij}(k)$, if it exists. (This can be done deterministically over a real field, and probabilistically in general.)
3. Set $d=\min\{k: B_k \text{ has a solution}\}$.
4. Set $u=\sum_{0\leq j<d-d_1} u_{1j}(d)x^j+x^{d-d_1}$, and return $g=uf_1$.

THEOREM 3.2. *Algorithm* LCM OF MANY POLYNOMIALS *either returns the* lcm *of the input polynomials, or it reports "failure." The latter happens with probability less than* 2^{-n}. *If* F *is real, then the algorithm can be performed deterministically. The algorithm runs in parallel time* $O(\log^2 n)$.

We remark that over an arbitrary field the lcm can be computed deterministically in parallel time $O(\log^3 n)$ along a binary tree.

4. Factoring polynomials over finite fields. The parallel complexity of the factorization of polynomials over finite fields was the original motivation for the work begun in Borodin–von zur Gathen–Hopcroft [1982]. We present here the Cantor–Zassenhaus [1981] probabilistic algorithm (some of whose ingredients go back to Berlekamp [1970]; see also Knuth [1981, 4.6.2]) with the appropriate modifications for parallel execution. So let F be a finite field with q elements, and $f \in F[x]$ monic with degree $n \geqq 2$. We want to factor f.

In order to get a better timing estimate in case q is not prime, we let $G \subseteq F$ be another field with p elements, and $g \in G[t]$ irreducible of degree d such that $F = G[t]/(g)$ and $q = p^d$. (For any field F, one can of course choose $G = F$ and $p = q$.) F is a vector space over G with basis $1, t, \cdots, t^{d-1}$, and $R = F[x]/(f)$ is a vector space over F with basis $1, x, x^2, \cdots, x^{n-1}$, and a dn-dimensional vector space over G with basis $\{t^i x^j : 0 \leqq i < d, 0 \leqq j < n\}$.

ALGORITHM FACTORIZATION OVER A FINITE FIELD.
Input: A polynomial $f \in F[x]$ of degree n.
Output: Either the complete factorization of f, or "failure."

1. *Frobenius matrix.* Replace f by its (unique) monic scalar multiple. Compute the matrix Q of the linear mapping $R \to R$ with $u \mapsto u^p$. (This is called the Frobenius mapping if p is prime.)
2. *Nullspace.* Compute the dimension r of the nullspace K of $Q - I$, and $g_1, \cdots, g_r \in F[x]$ of degree less than n such that $g_1 \bmod f, \cdots, g_r \bmod f$ form a basis for K. If $r = 1$, set $S = \{f\}$ and go to step 5. (r is the number of distinct irreducible monic factors of f.)
3. *Random nullspace elements.* Let $m = \lceil 5 \log_2 r \rceil$, choose $v_{ij} \in G$ for $1 \leqq i \leqq m$, $1 \leqq j \leqq r$ independently at random, and let $h_i = \sum_{1 \leqq j \leqq r} v_{ij} g_j \in F[x]$ for $1 \leqq i \leqq m$.
4. *Primary factorization.* For $1 \leqq i \leqq m$ compute $c_i = \gcd(f, h_i^{(p-1)/2} - 1) \in F[x]$. If p is even, say $p = 2^k$, use $c_i = \gcd(f, \sum_{0 \leqq j < k} h_i^{2^j})$. Compute the common refinement of these partial factorizations as follows. Let $M = \{0, 1\} \times \{1, \cdots, m\}$. For all $I \subseteq M$ compute

$$s_I = \gcd\left(\{c_i : (0, i) \in I\} \cup \left\{\frac{f}{c_i} : (1, i) \in I\right\}\right).$$

   Then compute the following set T of "minimal I's:"

$$T = \{I \subseteq M : s_I \neq 1 \text{ and } \forall J \subseteq M \ I \subseteq J \Rightarrow s_J = 1 \text{ or } s_J = s_I\}.$$

   (Note that $I \subseteq J \Rightarrow s_J \mid s_I$.) T can be computed by comparing each pair (s_I, s_J) with $I \subseteq J$ and marking s_I as "irrelevant" if ($s_I = 1$ or ($s_J \neq 1$ and $s_J \neq s_I$)). Then fan in to keep only those I for which s_I has never been marked "irrelevant." Set $S = \{s_I : I \in T\}$. (This eliminates duplicate occurences of $s_I = s_J$ with $I \neq J$. We expect S to contain all "primary" factors of f, i.e. all g^e where g is an irreducible factor of f and e its multiplicity in f.)
5. *Complete factorization.* If $\# S \neq r$, then return "failure". Otherwise, for each $a \in S$ do the following. Set $b = a$. While $b' = db/dx = 0$, replace $b = \sum_{0 \leqq k} b_k x^k$ by its p_0th root $\sum_{0 \leqq k} b_{kp_0}^{q/p_0} x^k$, where $p_0 = \operatorname{char} F$ is a prime number. If $b' \neq 0$, compute $g = b/\gcd(b, b')$. Now g is an irreducible factor of f, and $e = \deg a / \deg g$ its multiplicity.
6. Return the set of all (g, e) computed above as the complete factorizaton of f.

THEOREM 4.1. *Algorithm* FACTORIZATION OVER A FINITE FIELD, *applied to a polynomial of degree n over a finite field with q elements, either returns the complete factorization of the polynomial or reports "failure." The probability of the second case is at most* $\frac{1}{2}$. *The parallel time is* $O(\log^2 n \log q)$ *operations in F. If f is squarefree,* $G \subseteq F$ *a subfield with p elements and* $q = p^d$, *then the cost is* $O(\log^2 n \log^2 (d+1) \log p)$ *operations in G. The number of processors is polynomial in n* $\log q$.

Proof. We start by proving the first estimate for the parallel time of the algorithm. So we count operations in F, and consider the matrix Q as an $n \times n$-matrix over F. Step 1 takes $O(\log^2 n \log p)$ and step 2 $O(\log^2 n)$ operations using Lemma 2.1 and the fast parallel algorithm for nullspace from Borodin–von zur Gathen–Hopcroft [1982]. The nullspace algorithm is probabilistic and has a probability of failure at most $2^{-n} \leqq \frac{1}{4}$.

The cost for step 3 is $O(\log r)$. (Note that $r \leqq n$.) In step 4, each c_i can be computed with $O(\log^2 n \log p)$ operations. For each s_I, we apply GCD OF MANY POLYNOMIALS eight times in parallel, with the same number n and using parallel time $O(\log^2 n)$. Unless all these applications fail, we take any of the answers. (They will all agree.) Note that *card* $2^M \leqq 2^{2m} \leqq 2^{12 \log_2 r} = r^{12} \leqq n^{12}$. Thus T and S can be computed in parallel time $O(\log n)$. Finally, each g in step 5 can be computed in time $O(\log_{p_0} n \log q + \log^2 n \log p)$. Thus the total parallel time is $O(\log^2 n \log q)$.

For the second estimate of the parallel time, simply note that one operation in F can be simulated by operations in G in parallel time $O(\log^2 (d+1))$. (We write $d+1$ rather than d in order not to get 0 for $d=1$.) Since now all elements of R are represented by coefficients from G, we consider Q as a $dn \times dn$-matrix over G. The computation of g in step 5 is unnecessary, since f is assumed to be squarefree. We thus get the estimate $O(\log^2 n \log^2 (d+1) \log p)$.

For the proof of correctness, let us recapitulate the overall strategy of the algorithm. In steps 1, 2, 3 we compute $h_1, \cdots, h_m \in F[x]$ such that

$$f \mid h_i^p - h_i = (h_i^{(p-1)/2} - 1)(h_i^{(p+1)/2} + h_i)$$

(if p is odd). Thus each h_i provides a partial factorization $f = c_i \cdot f / c_i$. In step 4 we compute the common refinement of these partial factorizations, and then step 5 accounts for the possible presence of multiple factors. Unless all distinct irreducible factors have been correctly separated in step 4, the procedure will report "failure."

We will show below that the number r from step 2 is the number of distinct irreducible monic factors of f. Then it is clear that the algorithm either reports "failure" at some stage or correctly computes the complete factorization of f. All that remains to do is to estimate the probabilities of failure p_2, p_4, p_5 in steps 2, 4 and 5. We have $p_2 \leqq 2^{-n} \leqq \frac{1}{4}$, and $p_4 \leqq n^{12} 2^{-8n} \leqq \frac{1}{8}$.

In order to estimate p_5, let $f = f_1^{e_1} \cdots f_s^{e_s}$ be the complete factorization of f, where $f_1, \cdots, f_s$ are pairwise distinct irreducible monic polynomials and $e_1, \cdots, e_s \geqq 1$. Let

$$R' = F[x]/(f_1^{e_1}) \times \cdots \times F[x]/(f_s^{e_s}),$$

$$K' = \{(u_1, \cdots, u_s) \in R' \colon u_1, \cdots, u_s \in G\}.$$

K' is the linear space over G of "locally constant polynomials," i.e. those that are constant ($\in G$) modulo each $f_i^{e_i}$. We denote by $\alpha : R \to R'$ the isomorphism of the Chinese remainder theorem, and by $\bar{h}$ the image of $h \in F[x]$ in R. We first prove that $\alpha(K) = K'$. Let $h \in F[x]$ with $\bar{h} \in K = \{g \in R \colon g^p - g = 0\}$, and $\alpha(\bar{h}) = (u_1, \cdots, u_s)$. Since $\bar{h}^p - \bar{h} = 0$ in R, we have

$$f \mid h^p - h = \prod_{w \in G} (h - w).$$

These factors are pairwise relatively prime, and it follows that

$$\forall i \leqq s \; \exists w_i \in G \quad f_i^{e_i} \mid h - w_i.$$

Then $u_i = w_i \in G$. Hence $\alpha(\bar{h}) \in K'$. The reverse inclusion is clear, and $\alpha(K) = K'$ follows. In particular, $r = s$, and f is a power of an irreducible polynomial iff $r = 1$. For the remainder of the proof we can assume that $r \geqq 2$.

We first consider the case where p is odd. For $1 \leqq i < j \leqq r$ and $h \in F[x]$ with $\bar{h} \in K$ we say that h separates f_i and f_j iff exactly one of $f_i^{e_i}$ and $f_j^{e_j}$ divides $h^{(p-1)/2} - 1$. Writing $\alpha(\bar{h}) = (u_1, \cdots, u_r)$, this is equivalent to the condition that exactly one of u_i and u_j satisfies $u^{(p-1)/2} - 1 = 0$. The number of pairs $(u_i, u_j) \in G^2$ satisfying this condition is

$$2 \cdot \frac{p-1}{2} \cdot \frac{p+1}{2} = \frac{1}{2}(p^2 - 1),$$

and thus the probability that a randomly chosen $\bar{h} \in K$ does not separate f_i and f_j is $\frac{1}{2}(1 + p^{-2}) \leqq \frac{5}{9}$. The probability that f_i and f_j get separated by none of the randomly chosen $h_1, \cdots, h_m$ is $\leqq (\frac{5}{9})^m$, and the probability that some pair of factors does not get separated is

$$\leqq \binom{r}{2} \left(\frac{5}{9}\right)^m \leqq \frac{r^2}{2} \left(\frac{5}{9}\right)^{5 \log r} \leqq \frac{r^4}{8} \cdot r^{5 \log (5/9)} = \frac{1}{8} r^{4 + 5 \log (5/9)} \leqq \frac{1}{8}.$$

Thus the total probability of failure is $\leqq p_2 + p_4 + p_5 \leqq \frac{1}{2}$, and Theorem 4.1 is proven for odd characteristic.

If $p = 2^k$ is even, the argument applies with the following modifications: h separates f_i and f_j iff exactly one of $f_i^{e_i}$ and $f_j^{e_j}$ divides $\sum_{0 \leqq j < k} h_i^{2^j}$ iff exactly one of u_i and u_j satisfies $\sum_{0 \leqq j < k} u^{2^j} = 0$. The latter polynomial has exactly 2^{k-1} zeros in G, and the number of pairs (u_i, u_j) that satisfy the above condition is 2^{2k-1}. The probability that $\bar{h}$ does not separate f_i and f_j is $\frac{1}{2}$. As above, it follows that the probability of failure is at most $\frac{1}{2}$. $\square$

Of course, we can execute n instances (say) of the algorithm in parallel and obtain failure probability at most 2^{-n}, with the timing estimate of Theorem 4.1 remaining true. In von zur Gathen–Kaltofen [1983], a fast parallel algorithm for factoring a multivariate polynomial $f \in F[x_1, \cdots, x_r]$ is presented, where F is finite and f is given in a dense encoding.

In our model it would be natural to consider F as fixed, and then we have an $O(\log^2 n)$ algorithm. However, for the factoring problem it is important to consider the field F not as fixed but as somehow described in the input. We might think of the input as consisting of a prime number p, an irreducible polynomial $g \in \mathbf{Z}_p[t]$ of degree d such that $q = p^d$ and $F = \mathbf{Z}_p[t]/(g)$, and $f \in F[x]$, all represented by numbers in binary notation. Then we want an algorithm whose parallel time is polynomial in log (input size), i.e. polynomial in $\log \log q$ and $\log n$. Unfortunately, the version with $G = F$ of our algorithm falls far short from this goal, with its running time depending on $\log q$ rather than $\log \log q$. A corresponding unpleasant phenomenon occurs with the sequential deterministic algorithms using time $O(q)$ rather than $O(\log q)$, and the sequential probabilistic methods avoiding this do not translate into a fast parallel algorithm.

The stumbling block is the computation of powers of polynomials modulo f in steps 1 and 4. A related problem — which occurs in step 5 — is the modular computation of powers of numbers: given integers a, n, p compute $a^n \bmod p$. The "repeated

squaring" method for computing $a^n \bmod p$ does not make efficient use of parallelism. In particular, even with arbitrarily many processors it is not clear how to obtain parallel running time polynomial in $\log\log n$.

Open question 4.1. Is the above problem of computing powers in finite fields in NC?

For nonprime fields with $\# F = q = p^d$ the second version above goes a step in the right direction, essentially replacing $\log q$ by $\log^2 d \log p$. As an application, consider the algebraic BCH codes, where one wants to factor polynomials over large finite fields of characteristic 2; the parallel time of our algorithm is indeed polynomial in log (input size) for this problem. In particular, if we want to factor polynomials of degree n over $GF(2^n)$, then our Las Vegas algorithm provides an $O(\log^4 n)$ solution.

It seems to be a fundamental question what one can say about factors of polynomials over finite fields with the parallel time depending polynomially on log log (field size). Obviously a composite polynomial has a certificate (namely, a factorization) that can be verified in parallel time $O(\log n \log\log q)$. Also an irreducible polynomial f has such a certificate: for each $i < n = \deg f$, give $g_i \in F[x]$ with $x^{ip} \equiv g_i \bmod f$ and $\deg g_i < n$. The first condition can be certified by a sequence of intermediate results in a computation for $x^{ip} \bmod f$. From the coefficients of the g_i's one can verify that the rank of {fix points of the Frobenius mapping} is $n-1$, thus proving irreducibility of f. (This is a special case of the general observation that problems in the Boolean class NP seem to be computable by $\log n$-depth polynomial-size nondeterministic circuits.)

Open question 4.2. Are the irreducible polynomials and the factoring problem, both over finite fields, in NC?

In the sequential analogue of this situation, step 1 and a test "$r=1$?" in step 2 of the above algorithm provide a polynomial-time deterministic irreducibility test for polynomials over finite fields, going back to Berlekamp [1967]. The parallel time for this algorithm is $O(\log^2 nd \log p)$.

The recent breakthrough by Lenstra–Lenstra–Lovász [1982] provides a polynomial-time factorization procedure for univariate integer polynomials; it has been extended by Kaltofen [1982] to multivariate integer polynomials. A basic subroutine is to compute short vectors in integer lattices. From the point of view of this paper, it remains to adapt these algorithms to the parallel setting. In § 7 we will see that they give rise to reductions to the problem of finding short vectors; however, we do not have a good parallel algorithm for the latter problem.

5. Squarefree decomposition. Let F be an arbitrary field and $f \in F[x]$. f is called squarefree if there does not exist an $h \in F[x] \setminus F$ such that h^2 divides f. Let c be the leading coefficient of f, and $g = (g_1, \cdots, g_t)$ be a sequence of monic squarefree polynomials from $F[x]$ with $g_t \neq 1$. We call g the monotone squarefree decomposition of f if $f = c g_1 g_2 \cdots g_t$ and g_{i+1} divides g_i for $1 \leqq i < t$. This decomposition is unique, and g_1 is called the "squarefree part" of f. We call g the distinct power decomposition of f if $f = c g_1 g_2^2 \cdots g_t^t$ and $\gcd(g_i, g_j) = 1$ for $1 \leqq i < j \leqq t$. This decomposition is unique, too. The next two lemmas show that these two decompositions are closely related, and how to compute the decomposition of a product from the decompositions of the factors.

LEMMA 5.1. *If $(g_1, \cdots, g_t)$ is the monotone squarefree decomposition and $(h_1, \cdots, h_s)$ the distinct power decomposition of f, then $s = t$, $h_i = g_i / g_{i+1}$ (with $g_{t+1} = 1$) and $g_i = h_i h_{i+1} \cdots h_t$ for $1 \leqq i \leqq t$. In particular, one decomposition can be computed from the other in parallel time $O(\log^2 n)$.* □

LEMMA 5.2. *Let* $(f_1, \cdots, f_r)$ *and* $(g_1, \cdots, g_s)$ *be the monotone squarefree decompositions of* $f, g \in F[x]$. *For* $j, k \geqq 0$, $1 \leqq i \leqq r+s$ *set*

$$u_{jk} = \gcd(\operatorname{lcm}(f_j, g_j), \gcd(f_k, g_k)),$$

$$h_i = \operatorname{lcm}(\{u_{jk}: 0 \leqq j, k \text{ and } i \leqq j+k\}),$$

$$t = \max\{i: h_i \neq 1\}$$

(with $f_j = 1$ *for* $j > r$, $g_k = 1$ *for* $k > s$, *and* $f_0 = g_0 = fg$*). Then* $(h_1, \cdots, h_t)$ *is the monotone squarefree decomposition of* fg.

Proof. Each u_{jk} with $(j, k) \neq (0, 0)$ is squarefree, hence also each h_i. Also $h_{i+1} \mid h_i$. Let $a \in F[x]$ be irreducible, p, q the multiplicities of a in f, g, respectively. Thus, e.g. $a|f_j$ iff $j \leqq p$. It is sufficient to show that $a \mid h_{p+q}$ and $a \nmid h_{p+q+1}$. We can assume that $p \geqq q$, and then $a \mid \operatorname{lcm}(f_p, g_p)$ and $a \mid \gcd(f_q, g_q)$. Hence $a \mid u_{pq}$ and $a \mid h_{p+q}$. If $k > q$, then $a \nmid \gcd(f_k, g_k)$. Hence if $j + k > p + q$, then $a \nmid u_{jk}$. It follows that $a \nmid h_{p+q+1}$. □

Algorithms for computing the squarefree decomposition are of interest since they may yield an (incomplete) factorization with little effort, and since most factorization algorithms over **Q** and finite fields require squarefree polynomials as inputs (not, however, the first version of the algorithm of the previous section). Recall that one loop of the standard sequential squarefree factoring algorithms computes the squarefree part $f/\gcd(f, f')$ of f and passes $\gcd(f, f')$ to the next loop (see Knuth [1981, 4.6.2]). For a parallel algorithm over a field of characteristic zero, we use the following approach.

ALGORITHM SQUAREFREE.

Input: A number $n \in \mathbf{N}$, and a polynomial $f \in F[x]$ of degree at most n, where F is a field of characteristic zero.

Output: either the monotone squarefree decomposition $g = (g_1, \cdots, g_t)$ of f, or "failure."

1. For all i, j, $1 \leqq i < j \leqq n$, input $\prod_{i \leqq k \leqq j} k \in F$.
2. For all i, $0 \leqq i \leqq n$, compute

$$u_i = \frac{d^i f}{dx^i},$$

$$v_i = \gcd(u_0, \cdots, u_i),$$

$$g_i = v_{i-1}/v_i \quad (\text{for } i \geqq 1).$$

3. Set $t = \max\{i: 1 \leqq i \leqq n \text{ and } g_i \neq 1\}$.
4. Return $g = (g_1, \cdots, g_t)$.

THEOREM 5.3. *Algorithm* SQUAREFREE *either computes the monotone squarefree decomposition of the input polynomial, or reports "failure." The probability of the second case is less than* 2^{-n}. *The algorithm runs in parallel time* $O(\log^2 n)$. *If the field is real, then the algorithm can be performed deterministically.*

Proof. We first prove the bounds on the running time and failure probability, and then correctness. In step 2, we apply for each $i \geqq 1$ algorithm GCD OF MANY POLYNOMIALS twice to compute v_i, using the same number n as degree bound. The parallel time is $O(\log^2 n)$. SQUAREFREE reports "failure," if for some i both applications fail. (If neither fails, they will return the same value for v_i.) The total failure probability is at most $n2^{-2n} < 2^{-n}$.

For the proof of correctness, we can assume that f is monic, and let $f = f_1^{e_1} \cdots f_r^{e_r}$ be the factorization of f, where $f_1, \cdots, f_r$ are pairwise distinct irreducible monic

polynomials, and $e_1, \cdots, e_r \leqq 1$. Fix some $k \geqq r$, and let

$$a_i = (f'_k)^i \prod_{e_k - i < l \leqq e_k} l \prod_{\substack{1 \leqq j \leqq r \\ j \neq k}} f_j^{e_j}.$$

Then $f_k \nmid a_i$, and it is easy to see by induction on i that

$$f_k^{e_k - i + 1} \mid u_i - f_k^{e_k - i} a_i$$

for $0 \leqq i \leqq e_k$. It follows that the multiplicity of f_k in u_i and v_i is exactly $e_k - i$ for $0 \leqq i \leqq e_k$, and in g_i exactly 1 for $1 \leqq i \leqq e_k$ and 0 for $i > e_k$. Hence $g = (g_1, \cdots, g_t)$ is the monotone squarefree decomposition of f. □

Again, for an arbitrary field of characteristic zero, the algorithm can be performed deterministically in parallel time $O(\log^3 n)$.

6. Squarefree decomposition over finite fields. Let F be a field with q elements, $p = \operatorname{char} F$, $q = p^d$, and $f \in F[x]$ of degree at most n. We want to compute the squarefree decomposition of f. If our goal are $\log^2$ (input size)-algorithms and $q > p$ are both large, then a hard case is an innocent-looking polynomial like $f = x^p - a$. We only know how to compute $b = a^{q/p}$ (so that $f = (x - b)^p$) in parallel time $O(\log(q/p))$ which, just as in § 4, is in general not polynomial in log (input size) (see Open Question 4.1). For the estimates below, we let T_F be an upper bound on the parallel time to compute $a^p, a^{p^2}, \cdots, a^q$ for $a \in F$. Thus $T_F = O(\log(q/p))$. The algorithms of this section can be considered as reductions to this powering problem and systems of linear equations.

We first present an adaptation, called SQUAREFREE VIA DERIVATIVES, of the algorithm of § 5, which works for polynomials over a finite field. The only complication that requires some extra care is that dv_t/dx may be zero without v_t being constant. We will then have to compute $h = (v_t)^{1/p}$ at parallel cost $O(\log(q/p))$ and restart the algorithm with input h. Since this situation may occur several times consecutively, we only get $O((\log^2 n + \log(q/p)) \log_p n)$ as estimate for the parallel time. We then describe a second algorithm, SQUAREFREE VIA TAYLOR COEFFICIENTS, that avoids this factor $\log_p n$.

We thus have four parallel Las Vegas algorithms A_1, A_2, A_3, A_4 to compute the squarefree factorization. A_1 is the factoring algorithm of § 4 with $G = F$. A_2 is the second version of that factoring algorithm, with $G = \mathbf{Z}_p$, and step 5 of A_1 executed on the primary factors. A_3 and A_4 are SQUAREFREE VIA DERIVATIVES and SQUAREFREE VIA TAYLOR COEFFICIENTS, repectively. The parallel computing times are as follows:

$$\begin{aligned} T(A_1) &= O(\log^2 n \log q), \\ T(A_2) &= O(\log^2 n \log^2 (d+1) \log p + T_F), \\ T(A_3) &= O((\log^2 n + T_F) \log_p n), \\ T(A_4) &= O(\log^2 n + T_F). \end{aligned}$$

Thus, A_4 has the best uniform timing estimate among these algorithms, comparing particularly well with A_3 if p is small, and with A_1 and A_2 if $F = \mathbf{Z}_p$ for large p. (Of course, A_1 and A_2 provide much more information than A_3 or A_4 do.)

A somewhat surprising example is provided by polynomials of the form $x^2 - a \in F[x]$, where $\# F = 2^n$; then T_F seems to be $\Omega(n)$, and all four algorithms use parallel time $\Omega(n)$. Therefore we have the counterintuitive consequence that "squarefree decomposition" can be harder in parallel than "factoring squarefree polynomials"

over finite fields, while in characteristic zero, the first problem is always easy and the second one not even known to be in NC.

ALGORITHM SQUAREFREE VIA DERIVATIVES.

Input: A number $n \in \mathbf{N}$, and $f \in F[x]$ of degree at most n, where F is a finite field of characteristic p.

Output: Either the monotone squarefree decomposition g of f, or "failure."

1, 2, 3. Execute steps 1, 2, 3 of algorithm SQUAREFREE.

4. If $v_t' \neq 0$, then return $g = (g_1, \cdots, g_t)$. If $v_t' = 0$, then compute $h \in F[x]$ such that $v_t = h^p$. (Note that such an h exists.)
5. Call SQUAREFREE VIA DERIVATIVES recursively with input $(\lceil n/p \rceil, h)$ to obtain the monotone squarefree decomposition $(h_1, \cdots, h_s)$ of h.
6. Use Lemma 5.2 to merge the two monotone squarefree decompositions $(g_1, \cdots, g_t)$ of f/h^p and $(h_1, \cdots, h_1, \cdots, h_s, \cdots, h_s)$ of h^p (with each h_i written p times), and return the monotone squarefree decomposition of f.

We say that a polynomial $f \in F[x]$ is p-power-free if there does not exist an $h \in F[x] \backslash F$ such that h^p divides f. Thus 2-power-free is the same as squarefree.

THEOREM 6.1. *Let F be a finite field with q elements and* $\operatorname{char} F = p$, *and* $f \in F[x]$ *of degree at most n. Algorithm* SQUAREFREE VIA DERIVATIVES *either returns the monotone squarefree decomposition of f, or reports "failure." The latter occurs with probability less than 2^{-n}. The algorithm runs in parallel time* $O((\log^2 n + \log (q/p)) \log_p n)$. *If f is p-power-free, then it runs in parallel time* $O(\log^2 n)$. *The number of processors is polynomial in n.*

Proof. We leave the proof to the reader. Observe that steps 2, 5 and 6 have to be executed several times in parallel in order to get the estimate on the failure probability. □

To highlight the difference between the two algorithms presented in this section, let us first trace SQUAREFREE VIA DERIVATIVES on input $x^8 + x^7 + x^4 + x^3 \in \mathbf{Z}_2[x]$.

Step 2. $u_0 = x^8 + x^7 + x^4 + x^3$,
$u_1 = x^6 + x^2$,
$u_i = 0$ for $2 \leqq i \leqq 8$,

$v_0 = x^8 + x^7 + x^4 + x^3$,
$v_i = x^6 + x^2$ for $1 \leqq i \leqq 8$,

$g_1 = x^2 + x$,
$g_i = 1$ for $1 \leqq i \leqq 8$.

Step 3. $t = 1$.
Step 4. $h = x^3 + x$.
Step 5. $(h_1, h_2) = (x^2 + x, x + 1)$ by recursive application.
Step 6. Merge $(x^2 + x)$ with $(x^2 + x, x^2 + x, x + 1, x + 1)$ to get the output $(x^2 + x, x^2 + x, x^2 + x, x + 1, x + 1)$.

We now want to discuss a different approach to squarefree decomposition. In SQUAREFREE VIA DERIVATIVES, we compute pth roots from time to time, and apply the procedure recursively. Algorithm SQUAREFREE VIA TAYLOR COEFFICIENTS first computes the p-power-parts $w_0, \cdots, w_m$ of f, where $f = w_0 w_1^p w_2^{p^2} \cdots w_m^{p^m}$ and each w_i is p-power-free. This approach avoids the factor $\log_p n$ in the parallel time.

For i, $n \geqq 0$ define $(x^n)^{[i]} = \binom{n}{i} x^{n-i}$. Extending this by linearity, we get a mapping $^{[i]}: F[x] \to F[x]$. Thus $d^i f/dx^i = i! f^{[i]}$, $f^{[0]} = f$, and e.g. $(x^8 + x^7 + x^4 + x^3)^{[2]} = x^5 + x \in \mathbf{Z}_2[x]$. Clearly the $f^{[i]}$'s are more appropriate for computations than the derivatives, since for any $f \in \mathbf{Z}_2[x]$, $d^2 f/dx^2$ and all further derivatives are zero and thus carry no information at all. The $f^{[i]}$'s are the "universal Taylor coefficients" of f, since from them we get the Taylor expansion of f at an arbitrary point a as follows:

$$f = \sum_{0 \leqq i \leqq \deg f} f^{[i]}(a)(x-a)^i.$$

ALGORITHM SQUAREFREE VIA TAYLOR COEFFICIENTS.

Input: A number $n \in \mathbf{N}$, and a polynomial $f \in F[x]$ of degree at most n, where F is a finite field of characteristic p.

Output: Either the monotone squarefree decomposition $(g_1, \cdots, g_t)$ of f, or "failure".

1. For all i, j, $0 \leqq j \leqq i \leqq n$, input $\binom{i}{j} \in F$.
2. For all i, $0 \leqq i \leqq n$, compute $f^{[i]}$.
3. Set $l = \lfloor \log_p n \rfloor$. For all k, $0 \leqq k \leqq l$, compute

$$u_k = \gcd(f^{[0]}, \cdots, f^{[p^k - 1]}),$$

$$v_k = \frac{u_k}{u_{k+1}},$$

(with $u_{l+1} = 1$. v_k will be the "p^k-power-part of f" in the following sense: v_k will divide f and be equal to $w_k^{p^k}$ for some w_k, and for no irreducible factor v of v_k will $v^{p^{k+1}}$ divide f.)

$$w_k = v_k^{1/p^k},$$

$$y_k = (y_{k1}, \cdots, y_{k,t_k}),$$

the monotone squarefree decomposition of w_k. (Using steps 1, 2, 3 of algorithm SQUAREFREE. Note that w_k is "p-power-free" and the complication mentioned at the beginning of this section does not turn up.)

4. For all i, $1 \leqq i \leqq n$, do the following. Consider the p-adic representation $i = i_0 + i_1 p + \cdots + i_l p^l$ of i, where $0 \leqq i_k < p$. Compute

$$z_i = \gcd(y_{0,i_0}, y_{1,i_1}, \cdots, y_{l,i_l}),$$

(with $y_{kj} = 1$ if $j > t_k$, and $y_{k,0} = f$ for all k.)

$$g_i = \operatorname{lcm}(z_i, z_{i+1}, \cdots, z_n).$$

5. Set $t = \max\{i : g_i \neq 1\}$ and return $g = (g_1, \cdots, g_t)$.

Before we analyse the algorithm, let us look at the example considered above. With $f = x^8 + x^7 + x^4 + x^3 \in \mathbf{Z}_2[x]$ as input, SQUAREFREE VIA TAYLOR COEFFICIENTS produces the following:

Step 2. $f^{[0]} = x^8 + x^7 + x^4 + x^3$, $f^{[5]} = x^2$,
$f^{[1]} = x^6 + x^2$, $f^{[6]} = x$,
$f^{[2]} = x^5 + x$, $f^{[7]} = 1$,
$f^{[3]} = x^4 + 1$, $f^{[8]} = 1$.
$f^{[4]} = x^3 + 1$,

Step 3. Set $l=3$, and

$$\begin{array}{ll} u_0=x^8+x^7+x^4+x^3, & v_0=x^2+x, \\ u_1=x^6+x^2, & v_1=x^2, \\ u_2=x^4+1, & v_2=x^4+1, \\ u_3=1, & v_3=1, \\ w_0=x^2+x, & y_0=(x^2+x), \\ w_1=x, & y_1=(x), \\ w_2=x+1, & y_2=(x+1), \\ w_3=1, & y_3=(1). \end{array}$$

Step 4.

$$\begin{array}{ll} z_1=x^2+x, & \\ z_2=z_3=x, & g_1=g_2=g_3=x^2+x, \\ z_4=z_5=x+1, & g_4=g_5=x+1, \\ z_6=z_7=z_8=1, & g_6=g_7=g_8=1. \end{array}$$

Step 5. Return $g=(x^2+x, x^2+x, x^2+x, x+1, x+1)$.

THEOREM 6.2. *Let F be a finite field with q elements, $p=\mathrm{char}\, F$, and $f\in F[x]$ of degree n. With this input, algorithm* SQUAREFREE VIA TAYLOR COEFFICIENTS *either returns the squarefree decomposition of f or it reports "failure". The probability of the second case is less than 2^{-n}. The algorithm runs in parallel time $O(\log^2 n+\log_2(q/p))$. If f is p-power-free, then it runs in parallel time $O(\log^2 n)$. The number of processors is polynomial in n.*

Proof. We first describe the execution of the algorithm in some more detail, simultaneously proving the time bound. Then we establish the bound on the failure probability, and finally correctness.

Steps 1 and 2 can be performed in parallel time $O(\log n)$. In step 3, we apply algorithm GCD OF MANY POLYNOMIALS three times for each k to compute u_k. Below we prove that v_k is a p^kth power, and hence can be written as $v_k=\sum_{0\le i} v_{ki}x^{ip^k}$. Then $w_k=\sum_{0\le i} v_{ki}^{q^m/p^k}x^i$ can be computed in parallel time $O(\log(q/p))$, where $q^{m-1}<p^k\le q^m$, using $O(n)$ processors. For y_k, we use steps 1, 2, 3 of SQUAREFREE, applying it three times for each k. The algorithm runs in parallel time $O(\log^2 n)$. In step 4, for each i we apply GCD OF MANY POLYNOMIALS three times, and LCM OF MANY POLYNOMIALS three times for each g_i. The overall parallel time is $O(\log^2 n+\log(q/p))$, as claimed. If f is p-power free, then $v_1=\cdots=v_l=1$, and no p^kth roots with $k\ge 1$ have to be computed.

Failure may occur only in the computation of u_k, y_k, z_i, and g_i. In every single such computation, the failure probability is $\le 2^{-3n}$. The total failure probability is at most $(2l+2n)2^{-3n}\le 4n2^{-3n}<2^{-n}$.

For the proof of correctness, we assume that f is monic and let $f=h_0h_1^ph_2^{p^2}\cdots h_l^{p^l}$ be the "p-power-decomposition" of f, where each $h_i\in F[x]$ is monic and p-power-free. This decomposition exists and is unique. Fix some k, $0\le k\le l$, and write $r=h_0\cdots h_{k-1}^{p^{k-1}}$, $s=h_k^{p^k}\cdots h_l^{p^l}$. Then $f=rs$, r is p^k-power-free, and s is a p^kth power. We now show that $u_k=s$. For $0\le i<p^k$, we have

$$f^{[i]}=(rs)^{[i]}=\sum_{0\le j\le i} r^{[j]}s^{[i-j]}=r^{[i]}s^{[0]}=r^{[i]}s,$$

using Lemma 6.3(i) and (ii). Hence $u_k=s\cdot\gcd(r^{[0]},\cdots,r^{[p^k-1]})$. Lemma 6.3(v) implies that $\gcd(r^{[0]},\cdots,r^{[p^k-1]})=1$, and hence $u_k=s$. It follows that for all k we have $v_k=h_k^{p^k}$ and $w_k=h_k$.

Now each $y_{k,j}$ (with $j \geqq 1$) is squarefree, hence also every z_i and g_i is squarefree. Also g_{i+1} divides g_i. In order to show that g is the monotone squarefree decomposition of f, we only have to prove that $f = g_1 \cdots g_t$. So let $a \in F[x]$ be an irreducible monic factor of f, and m its multiplicity in f (so that $a^m | f$ and $a^{m+1} \nmid f$). We have to show that a divides g_i iff $i \leqq m$. Using the p-adic representation $m = m_0 + m_1 p + \cdots + m_l p^l$ of m, where $0 \leqq m_k < p$, it follows from the above that a divides h_k iff $m_k \neq 0$. More precisely, a divides $h_k = w_k$ exactly with multiplicity m_k, and hence

$$a \,|\, y_{k,j} \Leftrightarrow j \leqq m_k.$$

It follows that a divides z_m and hence g_m. Then a also divides each of $g_1, \cdots, g_m$, and it is sufficient to prove that a does not divide g_{m+1}. But for any $i = i_0 + i_1 p + \cdots + i_l p^l > m$ some p-adic coefficient i_k is greater than m_k, hence a does not divide y_{k,i_k}, and also not z_i or g_i. It follows that the multiplicity of a in $g_1 \cdots g_t$ is exactly m, and hence $f = g_1 \cdots g_t$. □

We remark that in our model the binomial coefficients computed in step 1 are considered as constants in F, and hence are given for free. However, they can also be computed in F, just using the constants 0 and 1 and field operations, either by computing $(x+1)^i$ or by Lucas' [1877] formula (see also Fine [1947]):

$$\binom{i}{j} \equiv \binom{i_k}{j_k} \cdots \binom{i_0}{j_0} \bmod p,$$

if $i = i_0 + i_1 p + \cdots + i_l p^l$ with $0 \leqq i_k < p$ is given in p-adic representation, and similarly for j. Note that the definition via factorials may fail to give a computation in positive characteristic.

In the following lemma we collect the facts concerning $f^{[i]}$ that were used in the preceding proof. It may be interesting to compare with the properties of the ith derivative $f^{(i)}$. Statements (ii) and (iii) are also true in that case, (iv) and (v) are false in general, and (i) has to be replaced by the Leibniz rule

$$(fg)^{(i)} = \sum \binom{i}{j} f^{(j)} g^{(i-j)}.$$

LEMMA 6.3. *Let F be a field of characteristic $p > 0$, $f, g \in F[x]$, and $i \geqq 0$. Then*

(i) $(fg)^{[i]} = \sum_{0 \leqq j \leqq i} f^{[j]} g^{[i-j]}$.
(ii) $(f^{p^i})^{[j]} = 0$ *for* $0 < j < p^i$.
(iii) $f \,|\, (f^i)^{[j]}$ *for* $0 \leqq j < i$.
(iv) *If f is irreducible and $f' \neq 0$, then* $\gcd(f, (f^i)^{[i]}) = 1$.
(v) *If F is finite and f is p^i-power-free, then* $\gcd(f^{[0]}, f^{[1]}, \cdots, f^{[p^i-1]}) = 1$.

Proof. (i) The case $f = x^m$, $g = x^n$ follows from comparing coefficients of x^{n+m-i} in

$$\sum_{0 \leqq l \leqq m+n} (fg)^{[l]} = \sum_{0 \leqq l \leqq m+n} \binom{n+m}{l} x^{n+m-l} = (1+x)^{n+m}$$

$$= (1+x)^n (1+x)^m = \sum_{j,k} \binom{n}{j} x^{n-j} \binom{m}{k} x^{m-k} = \sum_{j,k} f^{[j]} g^{[k]}.$$

Since both sides are bilinear, (i) also follows for arbitrary f, g.

(ii) For $f = ax^n$ and $0 < j < p^i$ with $a \in F$, (ii) follows from $\binom{np^i}{j} = 0$. For general f, (ii) now follows from the additivity of the left-hand side.

(iii) We use induction on i, the case $i = 0$ or $i = 1$ being trivial. For $i > 0$ we write

$$(f^i)^{[j]} = (f^{i-1} f)^{[j]} = \sum_{0 \leqq l \leqq j} (f^{i-1})^{(l)} f^{[j-l]},$$

using (i). By the induction hypothesis, f divides each summand with $l < i-1$. The only summand that is possibly left is $(f^{i-1})^{[i-1]}f^{[0]} = (f^{i-1})^{[i-1]}f$, which is also divisible by f.

(iv) We use induction on i and write

$$h = (f^i)^{[i]} = (f^{i-1}f)^{[i]} = \sum_{0 \leqq j \leqq i} (f^{i-1})^{[j]} f^{[i-j]}.$$

Then f divides all summands except the one for $j = i-1$, and hence

$$\gcd(f, h) = \gcd(f, (f^{i-1})^{[i-1]} f^{[1]}) = \gcd(f, f') = 1.$$

(v) Assume that there exists an irreducible $g \in F[x]$ such that g divides $f^{[j]}$ for $0 \leqq j < p^i$. We show by induction on j for $0 \leqq j < p^i$ that $g^{j+1} | f$. This yields the desired contradiction for $j = p^i - 1$. The claim is clear for $j = 0$. For $j \geqq 1$, we can write $f = g^j h$ with some $h \in F[x]$ by induction hypothesis. Then g divides $f^{[j]} = \sum_{0 \leqq l \leqq j} (g^j)^{[l]} h^{[j-l]}$ and also by (iii) each summand with $0 \leqq l < j$. Hence g divides $(g^j)^{[j]} h^{[0]}$, and from (iv) we conclude that $g | h$ and $g^{j+1} | f$. □

We remark that in (v) it is sufficient to assume F perfect, and in (iv) that F is perfect and f squarefree. Without some such assumption, the conclusions may not hold: If $a \in F$ is not a pth power, then for $f = x^{p+1} - ax$ and $i = 1$ we have gcd $(f^{[0]}, f^{[1]}, \cdots, f^{[p^i - 1]}) = x^p - a \neq 1$.

Algorithm SQUAREFREE VIA TAYLOR COEFFICIENTS will work over any perfect field of characteristic $p > 0$, provided that we have an effective procedure for extracting pth roots. The question of squarefreeness over arbitrary fields is undecidable (von zur Gathen [1984]).

7. Some reductions. The previous sections have left open parallel versions for a number of factorization problems that have good sequential solutions. The two most important ones—concerning boolean circuits—are the gcd of two integers (see Open Question 1 in Borodin–von zur Gathen–Hopcroft [1982]) and:

Open question 7.1. Can univariate and multivariate integer polynomials be factored fast in parallel?

The univariate factorization problem would probably be attacked along the lines of Lenstra–Lenstra–Lovász [1982] via computing short vectors in **Z**-modules. For multivariate polynomials, one might use Kaltofen's [1982], [1983a] reductions. We now give the parallel versions of these reductions, and also reduce the integer gcd's to a special case of the short vector problem.

In this section, we are concerned with the circuit (=parallel boolean) complexity (rather than algebraic complexity over an arbitrary field) of functions; see Cook [1983] for an excellent overview of this theory. Our notion of reduction comes from Cook's paper: A boolean function f is NC-reducible to another function g if there exists a U_{E^*}-uniform family $(\alpha_n)_{n \in \mathbf{N}}$ of boolean circuits—of depth $O(\log^k n)$ for some $k \in \mathbf{N}$—where α_n computes f on inputs of length n and is allowed to have oracle nodes for g. An oracle node for g has input edges $x_1, \cdots, x_r$ and output edges $y_1, \cdots, y_s$ whose values satisfy $g(x_1, \cdots, x_r) = (y_1, \cdots, y_s)$. Such an oracle node contributes $\lceil \log r \rceil$ to the depth of the circuit.

The functions INTEGER GCD, UNIVARIATE FACTORIZATION OVER **Q**, and MULTIVARIATE FACTORIZATION OVER **Q** compute the (nonnegative) gcd of two integers, the complete factorization of a polynomial in $\mathbf{Q}[x]$, and of a polynomial in $\mathbf{Q}[x_1, \cdots, x_n]$, respectively. For the last problem, the input size is given by the length of a dense encoding of the polynomial. For the sparse encoding, a probabilistic sequential polynomial-time algorithm is known (von zur Gathen [1983a]), but a fast parallel version of that reduction to the bivariate case is open. A function

is called SHORT VECTORS if it takes as input vectors $a_1, \cdots, a_n \in \mathbf{Z}^n$, linearly independent over $\mathbf{Q}$, and returns a vector x in the $\mathbf{Z}$-module ("lattice") $M = \sum a_i\mathbf{Z} \subseteq \mathbf{Z}^n$ such that

$$\forall y \in M\setminus\{0\} \quad |x| \leqq 2^{(n-1)/2}|y|.$$

Thus x is a shortest vector in M, up to the factor $2^{(n-1)/2}$ arising in the work of Lenstra–Lenstra–Lovász, with respect to the L_2-norm $|y| = (\sum y_i^2)^{1/2}$. A function is called SHORT VECTORS IN DIMENSION TWO if it takes $a_1, a_2 \in \mathbf{Z}^2$ as inputs and produces an x as above; instead of the factor $2^{(2-1)/2}$ we allow an arbitrary constant c.

THEOREM 7.1. 1. UNIVARIATE FACTORIZATION OVER $\mathbf{Q}$ *is* NC-*reducible to* SHORT VECTORS.

2. MULTIVARIATE FACTORIZATION OVER $\mathbf{Q}$ *is* NC-*reducible to* UNIVARIATE FACTORIZATION OVER $\mathbf{Q}$.

3. INTEGER GCD *is* NC-*reducible to* SHORT VECTORS IN DIMENSION TWO.

Proof. 1. and 2. follow from the reductions of Lenstra–Lenstra–Lovász [1982] and Kaltofen [1983a], using the results of the previous sections. Note that one has to use a quadratic Hensel iteration instead of the more common linear one. One might either lift each factor separately and discard duplicate ones at the end, or one might lift all irreducible factors mod p simultaneously to factors mod p^k (p, k as usual). (See e.g. von zur Gathen [1984] for formulas for this lifting.)

For the reduction in 3., let c be the constant of the algorithm SHORT VECTORS IN DIMENSION TWO. We can assume $c \in \mathbf{N}$. In order to compute the gcd of $a, b \in \mathbf{Z}$, we can assume that a is positive, and associate to a, b the two vectors $u = (a(ac+1), 0)$, $v = (b(ac+1), 1) \in \mathbf{Z}^2$. If $x = (x_1, x_2)$ is a short vector in the $\mathbf{Z}$-module $M = u\mathbf{Z} + v\mathbf{Z} \subseteq \mathbf{Z}^2$, i.e.

$$\forall y \in M\setminus\{0\} \quad |x| \leqq c|y|,$$

then we compute $g = \gcd(a, b)$ as follows.

1. Set $S = \{i \in \mathbf{Z}: 1 \leqq i \leqq c,\ x_2/i \in \mathbf{Z},\ x_2 b/ai \in \mathbf{Z}\}$.
2. Set $m = \max S$. (We will show that $S \neq \varnothing$.)
3. Return $|am/x_2|$ as $\gcd(a, b)$.

We now make two claims:

(i) $\exists k \in \mathbf{Z}$, $0 < |k| \leqq c$ and $x = (0, ka/g)$. (This k is unique.)

(ii) $k = m$.

Then $g = am/x_2$, and the correctness of the algorithm follows. To prove claim (i), let

$$z = -\frac{b}{g}u + \frac{a}{g}v = \left(0, \frac{a}{g}\right) \in M\setminus\{0\}.$$

It is sufficient to show that for $w = (w_1, w_2) = su + tv \in M$ we have

$$|w| \leqq c|z| \Leftrightarrow \exists k \in \mathbf{Z}, 0 \leqq |k| \leqq c \text{ and } w = kz.$$

(In particular, $\pm z$ are the two shortest nonzero vectors in M, and any element of M in the circle around 0 of radius $c|z|$ is an integral multiple of z.) "$\Leftarrow$" is clear. For "$\Rightarrow$", assume $|w| \leqq c|z|$. Then

$$|(sa + tb)(ac+1)| = |w_1| \leqq |w| \leqq c|z| = c\frac{a}{g},$$

and $ac + 1 > ca/g$, hence $sa + tb = 0$. The only solutions s, t of this equation are of the form

$$s = k - \frac{b}{g}, \qquad t = k\frac{a}{g},$$

for some $k \in \mathbf{Z}$. Then $w = su + tv = (0, ka/g)$. From

$$|k|\frac{a}{g} = |w_2| = |w| \leqq c|z| = c\frac{a}{g}$$

we get $|k| \leqq c$.

In order to prove claim (ii), we use from claim (i) that there exists $k \in \mathbf{Z}$ such that $0 < |k| \leqq c$ and $x = (0, ka/g)$. It is sufficient to show that $S = \{i \in \mathbf{Z}: i \mid k\}$. The inclusion "$\supseteq$" is clear. For "$\subseteq$", let $i \in S$ and $p \in \mathbf{N}$ prime, and e_a, e_b, e_g, e_i, e_k be the multiplicities of p in a, b, g, i, k resp. Then $e_g = \min\{e_a, e_b\}$. If $e_g = e_a$, then from $ka/ig = x_2/i \in \mathbf{Z}$ we get $e_k + e_a \geqq e_i + e_g$ and $e_k \geqq e_i$. If $e_g = e_b$, then from $kb/ig = x_2 b/ai \in \mathbf{Z}$ we get $e_k + e_b \geqq e_i + e_g$ and $e_k \geqq e_i$. In either case, the multiplicity of p in k is not less than its multiplicity in i. It follows that i divides k. □

We remark that in the reduction for 3. only two special cases of integer division were used: for a, $b \in \mathbf{Z}$, test whether a divides b, and if it does, compute the quotient. Reif [1983] has shown that the quotient and remainder of two n-bit integers can be computed in $O(\log n \log^2 \log n)$ parallel bit operations, and Beame–Cook–Hoover [1984] have improved the parallel time to $O(\log n)$ with a slightly weaker uniformity property.

Open question 7.2. Can short vectors in $\mathbf{Z}$-modules be computed fast in parallel?

8. Conclusion. We have shown that a number of algebraic problems with polynomial-time sequential solutions have polynomial-log-time parallel solutions. The basic routines are those of linear algebra; all other problems get reduced to these.

The most important open questions are the factorization of integer polynomials, and the analogous sequential vs. parallel behaviour for integer problems, e.g. computing the gcd of two n-bit integers with $\log^{0(1)} n$ bit operations in parallel. We have reduced this problem to a subroutine that is likely to be employed in factoring integer polynomials.

In von zur Gathen [1983b] we show in a general framework that problems like Padé approximation, partial fraction decomposition (with factored denominators), and various interpolation problems also have a fast parallel solution. Ongoing work at Toronto has resulted in a fast parallel factorization procedure for multivariate polynomials over finite fields (von zur Gathen–Kaltofen [1983]) and an irreducibility test for multivariate polynomials over $\mathbf{C}$ (Kaltofen [1983b]).

REFERENCES

P. W. Beame, S. A. Cook and H. J. Hoover, *Log depth circuits for division and related problems*, Proc. 25th IEEE Symposium on Foundations of Computer Science, Singer Island, FL, 1984.

S. J. Berkowitz, *On computing the determinant in small parallel time using a small number of processors*, Inform. Process. Lett. 18 (1984), pp. 147–150.

E. R. Berlekamp, *Factoring polynomials over finite fields*, Bell System Tech. J., 46 (1967), pp. 1853–1859.

———, *Factoring polynomials over large finite fields*, Math. Comp., 24 (1970), pp. 713–735.

A. Borodin, S. Cook and N. Pippenger, *Parallel computation for well-endowed rings and space bounded probabilistic machines*, Tech. Rep. 162/83, Dept. Computer Science, Univ. Toronto, Inform. and Control, to appear.

A. Borodin, J. von zur Gathen and J. Hopcroft, *Fast parallel matrix and gcd computations*, Inform. and Control, 52 (1982), pp. 241–256.

W. S. Brown, *On Euclid's algorithm and the computation of polynomial Greatest Common Divisors*, J. Assoc. Comput. Mach., 18 (1971), pp. 478–504.

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.

D. G. CANTOR AND H. ZASSENHAUS, *On algorithms for factoring polynomials over finite fields*, Math. Comp., 36 (1981), pp. 587–592.

G. E. COLLINS, *Subresultants and reduced polynomial remainder sequences*, J. Assoc. Comput. Mach., 14 (1967), pp. 128–142.

S. A. COOK, *The classification of problems which have fast parallel algorithms*, Proc. International Conference on Foundations of Computation Theory, Borgholm, 1983, pp. 78–93.

L. CSANKY, *Fast parallel matrix inversion algorithms*, this Journal, 5 (1976), pp. 618–623.

W. EBERLY, *Very fast parallel matrix and polynomial arithmetic*, Proc. 25th IEEE Symposium on Foundations of Computer Science, Singer Island, FL, 1984.

J. EDMONDS, *Systems of distinct representatives and linear algebra*, J. of Res. Nat. Bureau of Standards, 71B (1967), pp. 241–245.

N. J. FINE, *Binomial coefficients modulo a prime*, Amer. Math. Monthly, 54 (1947), pp. 589–592.

J. VON ZUR GATHEN, *Hensel and Newton methods in valuation rings*, Tech. Report 155 (1981), Dept. Computer Science, Univ. Toronto, Math. Comp. to appear.

——— [83a], *Factoring sparse multivariate polynomials*, Proc. 24th IEEE Symposium on Foundations of Computer Science, Tucson, AZ, 1983, pp. 133–137.

——— [83b], *Representations of rational functions*, Proc. 24th IEEE Symposium on Foundations of Computer Science, Tucson, AZ, pp. 172–179; this Journal, to appear.

J. VON ZUR GATHEN AND E. KALTOFEN, *Polynomial-time factorization of multivariate polynomials over finite fields*, Proc. 10th ICALP, Barcelona, 1983, pp. 250–263.

E. KALTOFEN, *A polynomial-time reduction from bivariate to univariate integral polynomial factorization*, Proc. 23rd Annual IEEE Symposium on Foundations of Computer Science, Chicago, 1982, pp. 57–64.

——— [83a], *Polynomial-time reduction from multivariate to bivariate and univariate integer polynomial factorization*, this Journal, to appear.

——— [83b], *Fast parallel absolute irreducibility testing*, manuscript, November 1983.

D. E. KNUTH, *The Art of Computer Programming*, Vol. 2, 2nd ed., Addison–Wesley, Reading MA, 1981.

E. KRONECKER, *Zur Theorie der Elimination einer Variabeln aus zwei algebraischen Gleichungen*, Monatsberichte der Akademie der Wissenschaften, Berlin, 1881, pp. 535–600.

A. K. LENSTRA, H. W. LENSTRA AND L. LOVÁSZ, *Factoring polynomials with rational coefficients*, Math. Ann., 261 (1982), pp. 515–534.

E. LUCAS, *Sur les congruences des nombres eulériens et des coefficients différentiels des fonctions trigonométriques, suivant un module premier*, Bull. Soc. Math. France, 6 (1877/78), pp. 49–54.

J. MUIR, Letter from Mr. Muir to Professor Sylvester on the word continuant, Amer. J. Math., 1 (1878), p. 344.

J. REIF, *Logarithmic depth circuits for algebraic functions*, Proc. 24th Annual IEEE Symposium Foundations of Computer Science, Tucson, 1983, pp. 138–145.

W. L. RUZZO, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365–383.

V. STRASSEN, *The computational complexity of continued fractions*, this Journal, 12 (1983), pp. 1–27.

L. G. VALIANT, *Reducibility by algebraic projections*, in: Logic and Algorithmic, Zürich 1980, Monographie No. 30, Enseignement Mathématique, pp. 365–380.

SIAM J. COMPUT.
Vol. 13, No. 4, November 1984
© 1984 Society for Industrial and Applied Mathematics
011

IS THE INTERESTING PART OF PROCESS LOGIC UNINTERESTING?: A TRANSLATION FROM PL TO PDL*

R. SHERMAN†, A. PNUELI† AND D. HAREL†

Abstract. With the (necessary) condition that atomic programs in process logic (PL) be binary, we present an algorithm for the translation of a PL formula p into a program $\zeta(p)$ of propositional dynamic logic (PDL) such that a finite path satisfies p iff it belongs to $\zeta(p)$. This reduction has two immediate corollaries: 1) validity in this PL can be tested by testing validity of formulas in PDL; 2) all state properties expressible in this PL are expressible in PDL. The translation, however, is of nonelementary time complexity.

The significance of the result to the search for natural and powerful logics of programs is discussed.

Key words. process logic, propositional dynamic logic, temporal logic

1. Introduction. The formalism of dynamic logic [Pr1] has been successfully proposed as a unifying framework for the formal reasoning about programs. It generalizes, and at the same time simplifies, previous systems such as Hoare's axiomatic system [Ho], Dijkstra's predicate transformers [D], etc. It appears that as long as it is the input-output relation of a program (in contrast with its ongoing behavior) that we wish to study, dynamic logic provides us with a mathematically complete and elegant system of reasoning.

However, it was soon pointed out that if one is interested in the continuous behavior of programs and not only their in-out behavior, then dynamic logic seems inadequate. The need for reasoning about such behavior arises naturally in the study of nonterminating programs such as operating systems, and in the investigation of concurrent systems. One logic proposed in response to this need, temporal logic, has been used successfully in the analysis of concurrent systems [MP]. However, there is a desirable property, compositionality, which temporal logic does not enjoy, but which dynamic logic and other formalisms, such as Hoare logic and Dijkstra's predicate transformers, do. The principle of compositionality decrees that the formalism be syntax directed in the sense that it should derive its treatment of well-structured programs from its treatment of their immediate components. In contrast, the current formalism of temporal logic refers to instructions and labels in a fixed program and requires the analysis of the program as a whole without the possibility of studying its subparts. An additional drawback of temporal logic is its inability to express combined properties of many programs.

In view of this apparent dichotomy—a compositional system which does not deal with mid-execution properties, and a noncompositional system which does—there were many attempts to extend one or the other to yield a system, generically called *process logic*, enjoying the advantages of both. Pratt's original process logic [Pr2], Parikh's SOAPL [PA] and Nishimura's language [N], were preliminary efforts in this direction. A recently proposed system which seems to have unified the basic concepts of both dynamic and temporal logic is the system of process logic (PL) presented in [HKP]. It borrows the program constructs and modal operators [ ] and ⟨ ⟩ from dynamic logic, and the temporal connectives **f** and **suf** from temporal logic and combines them into a single system.

* Received by the editors June 4, 1982, and in final revised form August 19, 1983. The research reported here was supported in part by a grant of the Israeli Academy of Science, Basic Research Foundation.

† Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot 76100, Israel.

The declared purpose of this new system is to enable compositional reasoning about continuous behavior of programs. As such, one would expect it to be able to express properties on the propositional level inexpressible by either PDL or TL, the propositional versions of dynamic and temporal logic, respectively. Indeed, the PL expression $[\alpha]\mathbf{some}p$, for example, states that in every execution of the program α there must be at least one state satisfying p. It can be shown [H] that this property cannot be expressed in PDL. If p is some property of a second program, say $\langle\beta\rangle q$, then it is not expressible in TL either.

Having demonstrated this greater expressibility, PL certainly becomes an attractive system for study. It is shown in [HKP] that validity in (the propositional version of) PL is decidable by reduction to *SnS*, the second order theory of n successors [R]. This yields a nonelementary decision procedure in general, and it is still unknown whether an alternative elementary decision procedure exists. It has also been shown that various small extensions to the system lead to undecidability [CHMP], [S]. So much for background.

The investigation reported upon here was prompted by the following simple observation. Let us consider the PL statement $[\alpha]\mathbf{some}p$. As mentioned, it is inexpressible in PDL for an abstract α. But suppose we knew the internal structure of α: for example, let $\alpha=(ab)^*c$ where a, b, c are *atomic* instructions. By assuming that they are atomic, we imply that there are no observable states *during* the execution of any of them. Thus if p, a state property, ever arises during a computation of α it may arise either before or after an atomic instruction but never during one. Similarly, if p holds before and after every atomic instruction of α it may be considered as holding continuously throughout the execution of α. Thus the property $[(ab)^*c]\mathbf{some}p$, stating that p must hold somewhere in every execution of $\alpha=(ab)^*c$ is equivalent to the PDL statement:

$$[((\neg p)?a(\neg p)?b)^*(\neg p)?c(\neg p)?]\mathbf{false}.$$

This PDL formula states that there can be no computation such that p is false before and after each of its atomic instructions. We immediately note the difference between the two modes of expressing this property. In PL we say that somewhere within α, p is realized; in the equivalent PDL formalization we have to explicitly state that it is not true that p is not realized at any of the locations possible during α.

In this paper we show how to apply this basic idea systematically to produce a translation algorithm from PL to PDL. However, no such translation is possible without ensuring the compatibility between models of PL and PDL. To this end one must adopt the *locality* of atomic formulas, i.e., that they are basically state formulas, true at states rather than on paths. Locality was adopted in [HKP] too and, indeed, by [CHMP], [S], the decidability of PL is lost without it. Moreover, one must adopt the *atomicity* of atomic programs as discussed above, i.e., that they consist of binary relations rather than arbitrary paths. Atomicity too is necessary since without it by [H] no such translation from PL to PDL exists. Accordingly we consider BPL, for *binary process logic*, the formalism obtained from PL by adopting the above restrictions. One then notices that the same mathematical objects serve as models for both PDL and BPL, the difference being only in the way satisfiability is extended from atomic formulas to general ones.

Our translation, to be specific, assigns to each formula p of BPL a PDL program $\zeta(p)$ such that for all finite paths x in any model, x satisfies p in BPL iff x is a possible computation path of $\zeta(p)$ in PDL, i.e., $x\in\zeta(p)$. Note that we treat finite paths only, and indeed we consider only finite paths in the notion of validity in BPL. It would

perhaps be possible to consider infinite paths if there were a programming construct, say α^{ω}, which would generate in PDL infinite computations from atomic programs. However, we feel that very little pragmatic power is lost due to the locality, atomicity and finiteness assumptions.

An additional technical matter concerns the presence in PL of paths which do not arise as computations of any program, in contrast to PDL where paths are implicitly present only as program computations. To overcome this incompatibility $\zeta(p)$ will employ a new atomic program u, understood to stand for the *universal* program which connects any two states. This idea involves no real loss of generality since, after all, the paths one is usually interested in are ones which arise from executing programs. Moreover, the two corollaries to our result, which we discuss next, make no use of this understanding regarding u. Also, it is easy to show that the addition of u to either BPL or PDL does not destroy their decidability.

The translation of BPL formulas to PDL programs can be utilized in the following two ways. First, we show how validity (over finite paths) in BPL may be reduced to validity in PDL; this is done by showing that p is satisfiable in BPL iff $\langle\zeta(p)\rangle$**true** is satisfiable in PDL. Secondly, we show that in a certain precise sense BPL and PDL are actually equivalent in expressive power. Specifically, if formulas of BPL are evaluated in zero-length paths, i.e., ones consisting of a single state, then BPL formulas express precisely the properties expressible by PDL, provided all states are connected by programs.

Returning to the title of the paper, we believe that in spite of the restrictions imposed on BPL, it retains all the important features of PL, rendering it an advanced system for reasoning compositionally about the continuous behavior of programs. Yet, we have succeeded in showing that properties expressible in BPL are in fact expressible in PDL too. Does this fact therefore detract from our interest in process logic?

Our argument is that rather than detract from it, the existence of such a translation should even enhance our interest in systems such as PL. One reason for this is the fact that the translation actually emphasizes the difference in modes of expression in the two logics. As already pointed out above, we simply state $[\alpha]$**some**p for the natural utterance: "in all executions of α, p is true somewhere". To state the same in PDL we must have full information about the structure of α and p, and in expressing the statement we must exhaust all possible ways of partitioning them. This need for detailed information about the structure of α and p, violates the principles of encapsulation and information hiding, and implies that the PDL style of expressing this property is by necessity a low level nonuniform one in comparison with the PL style.

Section 2 contains the definitions of PDL and BPL, § 3, the main one of the paper, contains our technical results, and in § 4 we discuss the issue of complexity. The proofs of some of the lemmas in § 3 are tedious but straightforward manipulations of finite automata considered as path acceptors, and are therefore gathered in an Appendix. They do not seem, however, to follow from standard closure properties of finite automata. The reader should be able to understand the idea of the proof of the theorem without having to resort to these detailed proofs, which are provided for completeness.

2. PDL and BPL.

Notation. Φ_0—the set of atomic formulas, Π_0—the set of atomic programs.

Syntax and semantics of PDL ([FL]):

A model is a triple $(S, \models, \rho)$ where:

S—is a set of states.

$\models$—is a satisfiability relation for atomic formulas.

$\rho : \Pi_0 \mapsto 2^{S\times S}$—is an interpretation for atomic programs.

The set of PDL formulas Φ, the set of PDL programs Π and their extended interpretations $\models$ and ρ are defined by:

1. $\Phi_0 \subseteq \Phi$
   $\mathbf{true} \in \Phi$, $\forall s \in S$, $s \models \mathbf{true}$
   $\mathbf{false} \in \Phi$, $\forall s \in S$, $s \not\models \mathbf{false}$
2. If $p, q \in \Phi$ then $p \vee q \in \Phi$, $\neg p \in \Phi$
   $s \models p \vee q$ iff $s \models p$ or $s \models q$
   $s \models \neg p$ iff $s \not\models p$
3. If $\alpha \in \Pi$, $p \in \Phi$ then $\langle \alpha \rangle p \in \Phi$
   $s \models \langle \alpha \rangle p$ iff $\exists t$, $(s, t) \in \rho(\alpha)$ and $t \models p$
4. $\Pi_0 \subseteq \Pi$,
   $\theta \in \Pi$, $\rho(\theta) = \varnothing$, i.e., the empty set
   $u \in \Pi$, $\rho(u) = S \times S$, i.e., the universal program
5. If $\alpha, \beta \in \Pi$ then $\alpha \cup \beta \in \Pi$, $\alpha\beta \in \Pi$ and $\alpha^* \in \Pi$
   $\rho(\alpha \cup \beta) = \rho(\alpha) \cup \rho(\beta)$
   $\rho(\alpha\beta) = \{(s, t) \mid \exists t', (s, t') \in \rho(\alpha), (t', t) \in \rho(\beta)\}$
   $\rho(\alpha^*) = \bigcup_{i \geq 0} \rho(\alpha^i)$ ($\rho(\alpha^0) = \{(s, s) \mid s \in S\}$, $\alpha^{i+1} = \alpha\alpha^i$)
6. If $p \in \Phi$ then $p? \in \Pi$
   $\rho(p?) = \{(s, s) \mid s \models p\}$

Syntax and semantics of BPL:
A model is a triple, $(S, \models, R)$ where:
S, $\models$—as before.
R—the interpretation for atomic programs is an assignment of sets of paths of length one (two states) to atomic programs.

Note that a model for PDL, i.e., a triple $(S, \models, \rho)$ is a priori also a model for BPL. R is taken to be simply ρ itself.

A path in a model is a finite sequence of states, with repetitions allowed. For a path $x = (x_0, \cdots, x_k)$, y is a *proper suffix of* x, denoted $y < x$, if $y = (x_i, \cdots, x_k)$ for some $i \geqq 1$. We extend $\models$ to a satisfiability relation over paths, denoted by $\models_p$, and define R—the interpretation of BPL programs. R assigns a set of paths R_α to each BPL program, α, i.e., the set of all paths corresponding to α computations. Note that while PDL formulas are interpreted over *states*, BPL formulas are interpreted over *paths*. We will use $\models$ to denote $\models_p$ when there is no danger of confusion.

The set of BPL formulas $\bar{\Phi}$, the set of BPL programs $\bar{\Pi}$ and their extended interpretations $\models_p$, R_α are defined by:

1. $\Phi_0 \subseteq \bar{\Phi}$,
   For a path x and atomic formula $P \in \Phi_0$, $x \models_p P$ iff $x_0 \models P$ where x_0 is the first state of x.
   $\mathbf{true} \in \bar{\Phi}$, $\forall x$, $x \models_p \mathbf{true}$
   $\mathbf{false} \in \bar{\Phi}$, $\forall x$, $x \not\models_p \mathbf{false}$
2. If $p, q \in \bar{\Phi}$ then $p \vee q \in \bar{\Phi}$ and $\neg p \in \bar{\Phi}$
   $x \models_p p \vee q$ iff $x \models_p p$ or $x \models_p q$
   $x \models_p \neg p$ iff $x \not\models_p p$
3. If $\alpha \in \bar{\Pi}$, $p \in \bar{\Phi}$ then $\langle \alpha \rangle p \in \bar{\Phi}$
   $x \models_p \langle \alpha \rangle p$ iff $\exists y \in R_\alpha$ such that $xy \models_p p$
   (If $x = (x_0 \cdots x_k)$, $y = (y_0 \cdots y_l)$ and $x_k = y_0$ then $xy = (x_0 \cdots x_k y_1 \cdots y_l)$. If $x_k \neq y_0$ then xy is not defined. All future references to xy implicitly contain the requirement that xy is defined.)

4. If $p, q \in \bar{\Phi}$ then $\mathbf{f}p \in \bar{\Phi}$ and $p\mathbf{suf}q \in \bar{\Phi}$
   $x \models_p \mathbf{f}p$ iff $(x_0) \models_p p$ (x_0 is the first state of x; (x_0) is the path consisting of x_0 alone.)
   $x \models_p p\mathbf{suf}q$ iff there exists a path y such that:
   a. $y < x$ and $y \models_p q$
   b. for every z such that $y < z < x$, $z \models_p p$
5. $\Pi_0 \subseteq \bar{\Pi}$
   $\theta \in \bar{\Pi}$, $R_\theta = \varnothing$
   $u \in \bar{\Pi}$, $R_u = S \times S$
6. If $\alpha, \beta \in \bar{\Pi}$ then $\alpha \cup \beta \in \bar{\Pi}$, $\alpha\beta \in \bar{\Pi}$ and $\alpha^* \in \bar{\Pi}$
   $R_{\alpha \cup \beta} = R_\alpha \cup R_\beta$
   $R_{\alpha\beta} = \{x | \exists y \in R_\alpha, \exists z \in R_\beta \text{ s.t. } x = yz\}$
   $R_{\alpha^*} = \bigcup_{i \geqq 0} R_{\alpha^i}$ $(R_{\alpha^0} = \{(s) | s \in S\})$
7. If $p \in \bar{\Phi}$ then $p? \in \bar{\Pi}$
   $R_{p?} = \{x | x \models_p p\}$

Programs of the form $p?$, in both PDL and BPL, are called *tests.*

3. Results.

DEFINITION.

1. For a BPL program α and a path x, denote $x \in R_\alpha$ simply by $x \in \alpha$.
2. For a PDL program α and a path $x = (x_0, \cdots, x_l)$, $x \in \alpha$ is defined by induction on the structure of α:

If $\alpha \in \Pi_0$ then $x \in \alpha$ iff $l = 1$ and $(x_0, x_1) \in \rho(\alpha)$; $x \notin \theta$ for all x.
$x \in u$ iff $x = (x_0, x_1)$ for some states x_0, x_1;
$x \in \alpha \cup \beta$ iff $x \in \alpha$ or $x \in \beta$;
$x \in \alpha\beta$ iff $\exists j, 0 \leqq j \leqq l$ such that $(x_0, \cdots, x_j) \in \alpha$ and $(x_j, \cdots, x_l) \in \beta$;
$x \in \alpha^*$ iff $\exists i \geqq 1$ such that $x \in \alpha^i$ or $l = 0$ (i.e., $x = (x_0)$);
If $p \in \Phi$ then $x \in p?$ iff $l = 0$ and $(x_0, x_0) \in \rho(p?)$.
Note that for every x, $x \in u^*$.

Denote by $T \subseteq \Pi$ the set of all tests of PDL, i.e., programs of the form $p?$ for $p \in \Phi$. We define Π_u to be the set of all programs over the alphabet $T \cup \{u\}$. Thus, the only atomic program used in Π_u is u. Note, however, that the tests appearing in Π_u may themselves contain programs which are not in Π_u.

THEOREM. *For every* BPL *formula p there exists a* PDL *program $\zeta(p)$ in Π_u such that, for every path x in every model,* $x \models_p p$ iff $x \in \zeta(p)$.

The proof of the theorem will proceed by induction on the structure of BPL formulas. Accordingly we will present a sequence of propositions and lemmas corresponding to the rules for constructing well formed BPL formulas.

PROPOSITION 1. *To atomic* BPL *formulas and to the special formulas* **true** *and* **false** *there correspond* PDL *programs in the sense of the theorem.*

Proof. For atomic $P \in \bar{\Phi}$ take $\zeta(P) = P?u^*$. Obviously, a path $x = (x_0, \cdots, x_l)$ satisfies P iff $x_0 \models P$ iff $(x_0, \cdots, x_l) \in P?u^*$. Similarly take $\zeta(\mathbf{true}) = u^*$ and $\zeta(\mathbf{false}) = \theta$. □

PROPOSITION 2. *If $p, q \in \bar{\Phi}$, two* BPL *formulas, already have corresponding translations $\zeta(p)$, $\zeta(q) \in \Pi_u$ in the sense of the theorem, then so does the formula $p \vee q$.*

Proof. Take $\zeta(p \vee q) = \zeta(p) \cup \zeta(q)$. Obviously $x \models_p p \vee q$ iff $x \models_p p$ or $x \models_p q$ iff $x \in \zeta(p)$ or $x \in \zeta(q)$ iff $x \in \zeta(p) \cup \zeta(q)$. □

Proposition 2 can be interpreted as a closure property, namely, that the class of sets of paths definable by Π_u programs is closed under union. For our next step we will need the closure of this class under complementation. Thus we shall show that to every Π_u program α, there corresponds a program $\tilde{\alpha}$ such that $x \notin \alpha \Leftrightarrow x \in \tilde{\alpha}$. Needless

to say, this fact is not (and does not follow directly from) the fact that regular sets are closed under complementation.

For a PDL program α, denote by T_α the set of tests appearing in α. Then α may be regarded as a regular expression over the alphabet $\Pi_0 \cup T_\alpha$.

DEFINITION. Two PDL programs α and β are *equivalent*, denoted by $\alpha \approx \beta$, if for every path x in every model, $x \in \alpha$ iff $x \in \beta$.

We start by defining two special sets of PDL programs. The set of programs whose execution sequences consist of alternations of atomic programs and tests is denoted by Γ:

$\Gamma = \{\alpha \mid \alpha \in \Pi$, for every word w in the language defined by the regular expression α, w is either λ or of the form $p_0?a_1p_1? \cdots a_kp_k?$ for some $k \geqq 0$ and $p_i \in \Phi$, $a_j \in \Pi_0 \cup \{u\}\}$.

Γ_u is defined similarly, except that u is the only atomic program allowed. Thus Γ_u consists of alternations of tests with the universal program u.

The following lemmas and their corollaries establish that Γ and Γ_u are normal forms (in the "path sense") of the set Π of PDL programs, and, respectively, the set Π_u of PDL programs involving no atomic programs but u.

LEMMA 1.

1. *Let α, $\beta \in \Gamma$; then $\alpha \cup \beta \in \Gamma$.*
2. *Let α, $\beta \in \Gamma$; then there exist programs γ, $\delta \in \Gamma$ such that $\gamma \approx \alpha\beta$, and $\delta \approx \alpha^*$.*

Proof. See Appendix.

COROLLARY 1. *For every* PDL *program $\alpha \in \Pi$ there exists a program $\psi(\alpha) \in \Gamma$ such that $\psi(\alpha) \approx \alpha$.*

Proof. By induction on the structure of α:

For atomic program a, let $\psi(a) = \mathbf{true}?a\mathbf{true}?$.

For $\alpha = p?$, let $\psi(\alpha) = \alpha$.

For $\alpha = \beta \cup \gamma$, let $\psi(\alpha) = \psi(\beta) \cup \psi(\gamma)$. By Lemma 1, $\psi(\alpha) \in \Gamma$.

For $\alpha = \beta\gamma$ consider $\psi(\beta)$ and $\psi(\gamma)$, which exists by the inductive hypothesis. Let $\psi(\alpha)$ be the program in Γ equivalent to $\psi(\beta)\psi(\gamma)$ and existing by Lemma 1.

For $\alpha = \beta^*$, let $\psi(\beta)$ be the program existing for β by the inductive hypothesis. Let $\psi(\alpha)$ be the program in Γ equivalent to $\psi(\beta)^*$ and existing by Lemma 1. □

LEMMA 2.

1. *Let α, $\beta \in \Gamma_u$; then $\alpha \cup \beta \in \Gamma_u$.*
2. *Let α, $\beta \in \Gamma_u$; then there exist programs γ, $\delta \in \Gamma_u$ such that $\gamma \approx \alpha\beta$ and $\delta \approx \alpha^*$.*

Proof. See Appendix.

COROLLARY 2. *For every* PDL *program $\alpha \in \Pi_u$ there exists a program $\psi(\alpha) \in \Gamma_u$ such that $\psi(\alpha) \approx \alpha$.*

Proof. Similar to the proof of Corollary 1. □

DEFINITION. Given a set of formulas $T = \{p_1, \cdots, p_n\}$, an *atom* of T is a conjunction $\bigwedge_{i=1}^n q_i$ where each q_i is either p_i or $\neg p_i$. T^A is defined to be the set of all atoms of T. Thus, in particular, T_α^A is the set of atoms constructed from the tests appearing in the program α.

LEMMA 3. *Let α be a program in Γ_u with tests T_α. Then there exists a program $\tilde{\alpha} \in \Gamma_u$ over the alphabet $\{u\} \cup T_\alpha^A$ such that for every path x in any model, $x \in \tilde{\alpha}$ iff $x \notin \alpha$.*

Proof. See Appendix.

We may thus summarize this sequence of lemmas by the following proposition.

PROPOSITION 3. *If a* BPL *formula $p \in \bar{\Phi}$ is translatable into a* PDL *program $\zeta(p)$ in Π_u in the sense of the theorem, then so is $\neg p$.*

Proof. By Corollary 2 there is a program $\gamma = \psi(\zeta(p))$ in Γ_u, equivalent to $\zeta(p)$. Now let $\zeta(\neg p)$ be $\tilde{\gamma}$, guaranteed to exist and to satisfy the properties by Lemma 3. □

PROPOSITION 4. *If a* BPL *formula* p *is translatable into a* PDL *program* $\zeta(p)$ *in* Π_u *in the sense of the theorem, then so is* $\mathbf{f}p$.

Proof. Let $\zeta(\mathbf{f}p)$ be Yu^*, where Y denotes the union of the set of all words of the form $q?$ in $\zeta(p)$. Since Y is finite, $\zeta(\mathbf{f}p)$ is regular. □

Even though taking care of union and complementation (cf. Lemmas 2, 3) automatically ensures closure under intersection, even in the "path sense", we do need this property in a more general setting. We therefore consider next closure under intersection.

LEMMA 4.

1. Let $\alpha \in \Pi$ and $\beta \in \Gamma_u$. Then there exists a program $\alpha \cap \beta \in \Gamma$ such that for every path x, $x \in \alpha \cap \beta$ iff $x \in \alpha$ and $x \in \beta$.
2. Let $\alpha, \beta \in \Gamma_u$. Then there exists a program $\alpha \cap \beta \in \Gamma_u$ such that for every path x, $x \in \alpha \cap \beta$ iff $x \in a$ and $x \in \beta$.

Proof. See Appendix.

We now have to deal with formulas in BPL of the form $p\mathbf{suf}q$.

DEFINITION. Ω_α, the *partition set* for a program $\alpha \in \Gamma_u$, is defined as a subset of $\Pi_u \times \Pi_u$ by induction on α:

1. $\Omega_u = \{(\mathbf{true}?, u), (u, \mathbf{true}?)\}$.
2. $\Omega_{p?} = \{(\mathbf{true}?, p?), (p?, \mathbf{true}?)\}$.
3. $\Omega_{\beta \cup \gamma} = \Omega_\beta \cup \Omega_\gamma$.
4. $\Omega_{\beta\gamma} = \{(\beta_1, \beta_2\gamma) | (\beta_1, \beta_2) \in \Omega_\beta\} \cup \{(\beta\gamma_1, \gamma_2) | (\gamma_1, \gamma_2) \in \Omega_\gamma\}$.
5. $\Omega_{\beta^*} = \{(\mathbf{true}?, \mathbf{true}?)\} \cup \{(\beta^*\beta_1, \beta_2\beta^*) | (\beta_1, \beta_2) \in \Omega_\beta\}$.

The intuitive meaning is that Ω_α contains the pairs of regular expressions which, when concatenated, yield α.

LEMMA 5. *Let* $\alpha \in \Gamma_u$. *Then for every path* x *and every* x_1, x_2 *such that* $x = x_1x_2$, $x \in \alpha$ *iff there is a pair* $(\alpha_1, \alpha_2) \in \Omega_\alpha$ *such that* $x_1 \in \alpha_1$ *and* $x_2 \in \alpha_2$.

Proof. Immediate. □

LEMMA 6. *Let* $\alpha \in \Gamma_u$. *Then there exists a program* $\sigma(\alpha) \in \Gamma_u$ *such that* $x \in \sigma(\alpha)$ *iff for every suffix (not necessarily proper)* x' *of* x, $x' \in \alpha$.

Proof. Let $\beta = ((u\mathbf{true}?)^*\tilde{\alpha})$, and take $\sigma(\alpha)$ to be $\tilde{\beta}$. One then observes that $x \in \sigma(\alpha)$ iff $x \notin (u\mathbf{true}?)^*\tilde{\alpha}$, iff for every suffix x' of x, $x' \notin \tilde{\alpha}$, iff for every suffix x' of x, $x' \in \alpha$. □

LEMMA 7. *Let* $\alpha, \beta \in \Gamma_u$. *Then there exists a program* $\mu(\alpha, \beta) \in \Gamma_u$ *such that for every path* x, $x \in \mu(\alpha, \beta)$ *iff there exists* $y \in \beta$, *with* $y < x$, *such that*

(*) *for every* z, *if* $y < z < x$ *then* $z \in \alpha$.

Proof. Let Ω be a nonempty subset of the partition set Ω_α. Denote:

$$h(\Omega) = \bigcup_{\{\alpha_1 | \exists \alpha_2, (\alpha_1, \alpha_2) \in \Omega\}} \alpha_1,$$

$$t(\Omega) = \bigcap_{\{\alpha_2 | \exists \alpha_1, (\alpha_1, \alpha_2) \in \Omega\}} \alpha_2.$$

Then let

$$\mu_1(\alpha, \beta) = \bigcup_{\Omega \subseteq \Omega_\alpha} (\sigma(h(\Omega))((u\beta) \cap t(\Omega))),$$

$$\mu_0(\alpha, \beta) = (u\beta) \cup (u\mu_1(\alpha, \beta)).$$

We first prove the following claim:

For every path $x = (x_0, \cdots, x_k)$, $x' = (x_1, \cdots, x_k) \in \mu_1(\alpha, \beta)$ iff there exists $y \in \beta$, $y < x'$ and y satisfies (*) above. Indeed, $x' \in \mu_1(\alpha, \beta)$, iff (for some $\Omega \subseteq \Omega_\alpha$, fixed in what follows) $x' \in \sigma(h(\Omega))((u\beta) \cap t(\Omega))$, iff $\exists j$, $1 \leqq j \leqq k$ such that $(x_1, \cdots, x_j) \in$

$\sigma(h(\Omega))$, $(x_j, \cdots, x_k) \in u\beta$ and $(x_j, \cdots, x_k) \in t(\Omega)$, iff $\exists j$, $1 \leqq j \leqq k$ such that $(x_{j+1}, \cdots, x_k) \in \beta$, $\forall l$, $1 \leqq l \leqq j$, $(x_l, \cdots, x_j) \in h(\Omega)$ and $(x_j, \cdots, x_k) \in \alpha_2$, $\forall \alpha_2$ s.t. $\exists \alpha_1$, $(\alpha_1, \alpha_2) \in \Omega$ (by Lemma 4, Part 2), iff $\exists j$, $1 \leqq j \leqq k$ such that $(x_{j+1}, \cdots, x_k) \in \beta$ and $\forall l$, $1 \leqq l \leqq j$, $\exists(\alpha_1, \alpha_2) \in \Omega_\alpha$ such that $(x_l, \cdots, x_j) \in \alpha_1$ and $(x_j, \cdots, x_k) \in \alpha_2$, iff $\exists j$, $1 \leqq j \leqq k$ such that $(x_{j+1}, \cdots, x_k) \in \beta$ and (by Lemma 5) $\forall l, 1 \leqq l \leqq j, \exists(\alpha_1, \alpha_2) \in \Omega_\alpha$ $(x_l, \cdots, x_k) \in \alpha_1\alpha_2 = \alpha$, iff $\exists y \in \beta$, $y < x'$ such that y satisfies $(*)$ above. This completes the proof of the claim.

Back to the main proof: Clearly, $\mu_0(\alpha, \beta) \in \Pi_u$ and by Corollary 2 there exists a program $\mu(\alpha, \beta) \in \Gamma_u$ such that $\mu_0(\alpha, \beta) \approx \mu(\alpha, \beta)$. Let x be a path, $x = (x_0, x_k)$. Then $x \in \mu(\alpha, \beta)$ iff $x \in \mu_0(\alpha, \beta)$ iff $(x_1, \cdots, x_k) \in \beta$ or $(x_1, \cdots, x_k) \in \mu_1(\alpha, \beta)$ iff $(x_1, \cdots, x_k) \in \beta$ or (by the claim) there exists $y \in \beta$, $y < (x_1, \cdots, x_k)$, such that y satisfies $(*)$ above, iff there exists $y \in \beta$, $y < x$, which satisfies $(*)$. □

PROPOSITION 5. *If* BPL *formulas p and q are translatable into* PDL *programs* $\zeta(p)$ *and* $\zeta(q)$ *in* Π_u *in the sense of the theorem then so is* p**suf**q.

Proof. Let $\zeta(p\textbf{suf}q) = \mu(\zeta(p), \zeta(q))$, existing by Lemma 7. □

Proof of the theorem. By induction on the structure of p: by Propositions 1-5 all we need to prove is the case $p = \langle\beta\rangle q$. We prove first that for every BPL program β there exists a PDL program $\beta' \in \Gamma$ such that for each path, $z \in \beta$ iff $z \in \beta'$. Define β' by induction on β as follows:

If $\beta \in \Pi_0$ then $\beta' = \textbf{true}?\beta\textbf{true}?$.

If $\beta = p?$ then by the main inductive hypothesis and Corollary 2 there exists $\zeta(p) \in \Gamma_u$ such that $z \in \zeta(p)$ iff $z \vDash_p p$. Thus $(p?)'$ can be taken to be $\zeta(p)$.

Now define:

$(\beta_1 \cup \beta_2)' = \beta_1' \cup \beta_2'$.

$(\beta_1\beta_2)' = \beta_1'\beta_2'$.

$(\beta_1^*)' = (\beta_1')^*$.

To continue the proof, note that $x \vDash_p p$ iff there is $y \in \beta$ such that $xy \vDash_p q$, iff (by the inductive hypothesis and the construction of β') $\exists y \in \beta'$ such that $xy \in \zeta(q)$, iff $\exists y \in \beta'$, $\exists(\alpha_1, \alpha_2) \in \Omega_{\zeta(q)}$ such that $x \in \alpha_1$, $y \in \alpha_2$ and xy is defined, iff $\exists(\alpha_1, \alpha_2) \in \Omega_{\zeta(q)}$ such that $x \in \alpha_1$, y_0 is the last state of x and $y_0 \vDash \langle\beta' \cap \alpha_2\rangle\textbf{true}$ (the existence of $\beta' \cap \alpha_2$ is guaranteed by Lemma 4, Part 1), iff $x \in \zeta'(p)$ where $\zeta'(p) \in \Pi_u$ is defined as $\bigcup_{(\alpha_1,\alpha_2)\in\Omega_{\zeta(q)}} \alpha_1(\langle\beta' \cap \alpha_2\rangle\textbf{true})?$.

By Corollary 2 there exists $\zeta(p) \in \Gamma_u$ with $\zeta(p) \approx \zeta'(p)$, and the proof is complete. □

Examples. Let P be an atomic formula, and a an atomic program. The following are simplified forms of $\zeta(q)$ for some sample formulas q.

1. A simplified form of $\zeta(\langle a\rangle P)$: $P?u^*(\langle a\rangle\textbf{true})?$.
2. A simplified form of $\zeta([a^*]\textbf{all}P)$: $P?(uP?)^*([a^*(\neg P)?a^*]\textbf{false})?$.
3. A simplified form of $\zeta(\langle a^*\rangle\textbf{some}P)$: $u^*P?u^* \cup u^*(\langle a^*\rangle P)?$.

COROLLARY 3. *For every* BPL *formula p there exists a* PDL *formula p' such that p is valid, i.e., true of all finite paths in all models, iff p' is valid, i.e., true of all states in all models.*

Proof. We will show that a BPL formula q is satisfiable iff $\langle\zeta(q)\rangle\textbf{true}$ is satisfiable in PDL. Accordingly, p' is taken to be $[\zeta(\neg p)]\textbf{false}$. Indeed, assume $x \vDash_p q$ in some model M. Let M' be the extension of M in which u is interpreted as the universal program connecting any two states. By the theorem $x \in \zeta(q)$ in M', and hence $x_0 \vDash \langle\zeta(q)\rangle\textbf{true}$ in M'. Conversely, if $s \vDash \langle\zeta(q)\rangle\textbf{true}$ in some model M, then there is some path x in M, with $x_0 = s$, such that $x \in \zeta(q)$. By the theorem $x \vDash_p q$ in M. □

DEFINITION. A *program model* over a set $\bar{a} = \{a_1, \cdots, a_n\}$ of atomic programs, is a model in which any two states are connected by some sequence of elements of $\bar{a}$.

COROLLARY 4. *For every* BPL *formula p there exists a* PDL *formula p' such that for every state s in any program model M, M,* $(s)\models_p \mathbf{f}p$ *iff M,* $s\models p'$.

Proof. Let s be a state in a program model M over $\bar{a}=\{a_1,\cdots,a_n\}$. By the theorem $(s)\models_p \mathbf{f}p$ in M iff $(s)\in\zeta(\mathbf{f}p)$ in the model extending M by interpreting u as the universal program. Define $\zeta'(\mathbf{f}p)$ to be the program $\zeta(\mathbf{f}p)$ where every appearance of the universal program u is replaced by $(\bigcup_{i=1}^n a_i)^*$, then by the definition of a program model $x\in\zeta(\mathbf{f}_p)$ in the extended model iff $x\in\zeta'(\mathbf{f}p)$ in M for any path x. Since (s) is of length zero, it follows that M, $(s)\models_p p$ iff M, $(s)\models_p \mathbf{f}p$ iff M, $s\models\langle\zeta'(\mathbf{f}p)\rangle\mathbf{true}$. □

As pointed out by one of the referees, the converse of Corollary 4 is also true. Namely, PDL formulas are translatable in BPL formulas such that satisfaction in a state is transferred into satisfaction in the appropriate zero-length path. To see this, the formula $\neg$(**false suf true**) (true only in paths of length zero) is added to each test, and the PDL formula is otherwise left untouched. Thus, in the sense of Corollary 4, BPL and PDL are actually equivalent in expressive power.

4. Complexity. The operator "**chop**" was defined in [HKP] by $x\models_p p\,\mathbf{chop}\,q$ iff $\exists y, z$ such that $x=yz$ and $y\models_p p$, $z\models_p q$. A formula containing **chop** can be easily translated to a PDL program by $\zeta(p\,\mathbf{chop}\,q)=\zeta(p)\zeta(q)$. Thus, in particular our result gives a decision procedure for validity in BPL + **chop**. A simple analysis of the translation algorithm presented herein shows a nonelementary complexity. This is actually not accidental since (as observed in [HP]) PL + **chop** is nonelementary even for program-free formulas. Thus BPL + **chop** is also nonelementary.

Appendix. The closure properties of Γ and Γ_u claimed in Lemmas 1-4 are proven here by considering special forms of finite automata accepting the programs in these classes. Although regular sets are closed under composition, Kleene-star, complementation and intersection, we are interested in these operations in the "path sense".

For example, the required complement $\tilde{\alpha}$ of α is not a program which defines a set of words over atomic programs and tests which is the complement of that defined by α, but rather is a program whose set of paths in the model is the complement of that of α. The existence of such a complement $\tilde{\alpha}$ is indeed established below by considering a certain deterministic automaton for α and appropriately complementing the set of final states, but this construction has to be done with care, and does not follow from the classical one.

First, for $\alpha\in\Gamma$, let $\mathcal{M}_\alpha$ be a nondeterministic finite automaton defining the language of α, which alternates in a predetermined way between two disjoint subsets of its set of states when encountering tests and atomic programs.

$$\mathcal{M}_\alpha=\langle K_1\cup K_2\cup\{d\}, \Pi_0\cup T_\alpha, k_0, \eta, F\rangle$$

where:

$K_1\cap K_2=\varnothing$, $k_0\in K_1$, $F\subseteq K_2$.

For $k\in K_1$, if $p\in T_\alpha$ then $\eta(k,p)\subseteq K_2$, and if $a\in\Pi_0$ then $\eta(k,a)=\{d\}$,

For $k\in K_2$, if $p\in T_\alpha$ then $\eta(k,p)=\{d\}$, and if $a\in\Pi_0$ then $\eta(k,a)\subseteq K_1$,

If $p\in T_\alpha$ then $\eta(d,p)=\{d\}$, if $a\in\Pi_0$ then $\eta(d,a)=\{d\}$.

Let $\bar{K}_1$ be the set of states leading to a final state:

$\bar{K}_1=\{k\mid k\in K_1, \exists p\in T_\alpha \text{ such that } \eta(k,p)\cap F\neq\varnothing\}$,

For any $k\in\bar{K}_1$ define: $T_\alpha(k)=\{p\mid p\in T_\alpha, \eta(k,p)\cap F\neq\varnothing\}$.

Note. For a program $\alpha\in\Gamma$, and a path $x=(x_0,\cdots,x_l)$, $x\in\alpha$ iff there is a word w defined by α, $w=p_0?a_0\cdots a_{l-1}p_l?$ such that $x\in w$; that is, $x_i\in p_i?$ i.e. $x_i\models p_i$, $0\leqq i\leqq l$, and $(x_i,x_{i+1})\in\rho(a_i)$, $0\leqq i<l$.

Proof of Lemma 1.

1. Obvious.

2. Let $\mathcal{M}_\alpha$, $\mathcal{M}_\beta$ be the automata defining α, β respectively:
$\mathcal{M}_\alpha = \langle K_1 \cup K_2 \cup \{d\}, \Pi_0 \cup T_\alpha, k_0, \eta, F\rangle$,
$\mathcal{M}_\beta = \langle K_1' \cup K_2' \cup \{d'\}, \Pi_0 \cup T_\beta, k_0', \eta', F'\rangle$,
Define: $T_\alpha \wedge T_\beta = \{p \wedge q \mid p \in T_\alpha, q \in T_\beta, p \neq q\} \cup \{T_\alpha \cap T_\beta\}$.
Define the automaton $\mathcal{M}_\gamma$ by:

$$\mathcal{M}_\gamma = \langle \hat{K}_1 \cup \hat{K}_2 \cup \{\hat{d}\}, \Pi_0 \cup T_\alpha \cup T_\beta \cup \{T_\alpha \wedge T_\beta\}, k_0, \hat{\eta}, \hat{F}\rangle$$

where

$$\hat{F} = \begin{cases} F \cup F' & \text{if } \lambda \in \beta \text{ (for example, if } \beta \text{ is of the form } \beta_1^* \text{ or } \beta_1^* \beta_2^*\text{)}, \\ F' & \text{otherwise,} \end{cases}$$

$\hat{K}_1 = K_1 \cup K_1' \cup \{e_1\}$, $\hat{K}_2 = K_2 \cup K_2' \cup \{e_2\}$.

The transition function $\hat{\eta}$ is given by the following cases:

A) For $k \in \hat{K}_1$

if $a \in \Pi_0$ then $\hat{\eta}(k, a) = \{\hat{d}\}$

if $p \in T_\alpha - T_\beta$ then

$$\hat{\eta}(k,p) = \begin{cases} \eta(k,p), & k \in K_1, \\ \{e_2\}, & k \in K_1' \cup \{e_1\} \end{cases}$$

if $p \in T_\beta - T_\alpha$ then

$$\hat{\eta}(k,p) = \begin{cases} \eta'(k,p), & k \in K_1', \\ \{e_2\}, & k \in K_1 \cup \{e_1\} \end{cases}$$

if $p \in T_\alpha \cap T_\beta$ then

$$\hat{\eta}(k,p) = \begin{cases} \eta'(k,p), & k \in K_1', \\ \eta(k,p), & k \in K_1 - \bar{K}_1, \\ \{e_2\}, & k = e_1, \\ \eta(k,p) \cup \eta'(k_0', p), & p \in T_\alpha(k) \cap T_\beta, k \in \bar{K}_1, \\ \eta(k,p), & p \notin T_\alpha(k) \cap T_\beta, k \in \bar{K}_1 \end{cases}$$

if $p \in T_\alpha \wedge T_\beta - \{T_\alpha \cap T_\beta\}$ then

$$\hat{\eta}(k,p) = \begin{cases} \eta'(k_0', q) \cup \eta(k, r), & p = q \wedge r, r \in T_\alpha(k), q \in T_\beta, k \in \bar{K}_1, \\ \{e_2\}, & p \notin T_\alpha(k) \wedge T_\beta, k \in K_1 \text{ or } k \notin \bar{K}_1 \end{cases}$$

B) For $k \in \hat{K}_2$

if $p \in T_\alpha \cup T_\beta \cup \{T_\alpha \wedge T_\beta\}$ then $\hat{\eta}(k,p) = \{\hat{d}\}$

if $a \in \Pi_0$ then

$$\hat{\eta}(k,a) = \begin{cases} \eta(k,a) & k \in K_2, \\ \eta'(k,a) & k \in K_2', \\ \{e_1\} & k = e_2 \end{cases}$$

C) For $\hat{d}$

if $p \in T_\alpha \cup T_\beta \cup \{T_\alpha \wedge T_\beta\} \cup \Pi_0$ then $\hat{\eta}(\hat{d}, p) = \{\hat{d}\}$

The program γ is then taken to be some regular expression corresponding to $\mathcal{M}_\gamma$.

For the α^* case define $\mathcal{M}_\delta$ by:

$$\mathcal{M}_\delta = \langle \hat{K}_1 \cup \hat{K}_2 \cup \{\hat{d}\}, \Pi_0 \cup T_\alpha \cup \{T_\alpha \wedge T_\alpha\} \cup \{\mathbf{true}\}, k_0, \hat{\eta}, \hat{F}\rangle$$

where: $\hat{K}_1 = K_1 \cup \{e_1\}$, $\hat{K}_2 = K_2 \cup \{k_t, e_2\}$, $\hat{F} = F \cup \{k_t\}$. The transition function $\hat{\eta}$ is given by the following cases:

A) For $k \in \hat{K}_1$

if $a \in \Pi_0$ then $\hat{\eta}(k, a) = \{\hat{d}\}$

if $p \in T_\alpha$ then

$$\hat{\eta}(k, p) = \begin{cases} \eta(k, p), & k \in K_1, \\ \{e_2\}, & k = e_1 \end{cases}$$

if $p \in T_\alpha \wedge T_\alpha$ then

$$\hat{\eta}(k, p) = \begin{cases} \eta(k_0, q) \cup \eta(k, r), & p = q \wedge r, r \in T_\alpha(k), q \neq r, q \in T_\alpha, k \in \bar{K}_1, \\ \eta(k_0, p) \cup \eta(k, p), & p \in T_\alpha(k), k \in \bar{K}_1, \\ \{e_2\}, & k \notin \bar{K}_1 \text{ or } p \notin T_\alpha(k) \wedge T_\alpha \end{cases}$$

$$\hat{\eta}(k, \mathbf{true}) = \begin{cases} \{k_t\}, & k = k_0, \\ \{e_2\}, & \text{otherwise} \end{cases}$$

B) For $k \in \hat{K}_2$

if $p \in T_\alpha \cup \{T_\alpha \wedge T_\alpha\}$, $\hat{\eta}(k, p) = \{\hat{d}\}$

if $a \in \Pi_0$

$$\hat{\eta}(k, a) = \begin{cases} \eta(k, a), & k \in K_2, \\ \{e_1\}, & k \in \{k_t, e_2\} \end{cases}$$

C) For $\hat{d}$

if $p \in T_\alpha \cup \{T_\alpha \wedge T_\alpha\} \cup \Pi_0$ then $\hat{\eta}(\hat{d}, p) = \{\hat{d}\}$

The program δ is then taken to be some regular expression corresponding to $\mathcal{M}_\delta$. □

Proof of Lemma 2. Similar to the proof of Lemma 1. □

Proof of Lemma 3. To prove the lemma we first define a *path deterministic* automaton (and later the complement automaton) in the sense that a *path* is "accepted" by at most a single sequence of states of the automaton.

Consider, for example, the automaton given in Fig. 1. While this automaton is deterministic in the usual sense, it is not path deterministic, because, for example, the path $x = (x_0, x_1)$ where $x_0 \models P \wedge Q$, and $x_1 \models R \wedge U$, is "accepted" by both sequences (s_0, s_1, s_2, s_3) and (s_0, s_4, s_5, s_6) of the automaton. Thus, while the labels on the edges originating in s_0 are disjoint, there exist paths satisfying $P \wedge Q$ which are compatible with both.

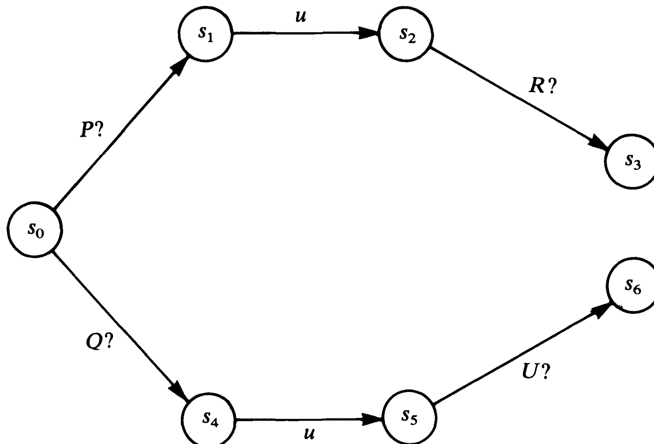


FIG. 1

We will be using the set of atoms T_α^A.

DEFINITION. Let $\alpha \in \Gamma_u$ and let $\mathcal{M}_\alpha = \langle K_1 \cup K_2 \cup \{d\}, \{u\} \cup T_\alpha, k_0, \eta, F\rangle$ be the automaton defining α. Then $\mathcal{M}_\alpha^E$, the (*nondeterministic*) *extended automaton* for α, is defined by:

$$\mathcal{M}_\alpha^E = \langle \hat{K}_1 \cup \hat{K}_2 \cup \{\hat{d}\}, \{u\} \cup T_\alpha^A, k_0, \eta^E, F\rangle$$

where: $\hat{K}_1 = K_1 \cup \{e_1\}$, $\hat{K}_2 = K_2 \cup \{e_2\}$, and T_α^A is the set of atoms for α. The transition function η^E is given by the following cases:

A) For $k \in \hat{K}_1$

$$\eta^E(k, u) = \{\hat{d}\}$$

for $k \in K_1$

$$\eta^E(k, \textstyle\bigwedge_{i=1}^n q_i) = \begin{cases} \{e_2\}, & \{q_i\}_{i=1}^n \cap T_\alpha = \varnothing, \\ \bigcup_j \{\eta(k, q_j) | q_j \in T_\alpha\}, & \text{otherwise} \end{cases}$$

$$\eta^E(e_1, \textstyle\bigwedge_{i=1}^n q_i) = \{e_2\}$$

B) For $k \in \hat{K}_2$

$$\eta^E(k, \textstyle\bigwedge_{i=1}^n q_i) = \{\hat{d}\}$$

$$\eta^E(k, u) = \begin{cases} \eta(k, u), & k \in K_2, \\ \{e_1\}, & k = e_2 \end{cases}$$

C) If $p \in T_\alpha^A \cup \{u\}$, $\eta^E(\hat{d}, p) = \{\hat{d}\}$

Claim 1. Let $\alpha \in \Gamma_u$, and let α^E be any program defining the set of words over $\{u\} \cup T_\alpha^A$ accepted by $\mathcal{M}_\alpha^E$. Then:

1. $\alpha^E \in \Gamma_u$.
2. $\alpha \approx \alpha^E$.

Proof.

1. Obvious from the form of $\mathcal{M}_\alpha^E$.

2. Let $x = (x_0, \cdots, x_k)$ be a path, $x \in \alpha$. Then there is a word $w \in \alpha$, $w = r_0?u \cdots ur_k?$, and $(x_i) \in r_i?$, $0 \leqq i \leqq k$. For every $p \in T_\alpha$ either $(x_i) \in p?$ or $(x_i) \in (\neg p)?$. It follows that for every i, $0 \leqq i \leqq k$ there exist $q_i^1, \cdots, q_i^n \in T_\alpha \cup \neg T_\alpha$ such that $q_i^i = r_i$ and $(x_i) \in (\bigwedge_{j=1}^n q_i^j)?$ for $0 \leqq i \leqq K$. By the definition of $\mathcal{M}_\alpha^E$ the word w^E given by: $w^E = (\bigwedge_{j=1}^n q_0^j)?u \cdots u(\bigwedge_{j=1}^n q_k^j)?$ is accepted by $\mathcal{M}_\alpha^E$ by the sequence of states in $\mathcal{M}_\alpha$ which accepts w, and $(x_i) \in (\bigwedge_{j=1}^n q_i^j)?$ for every $0 \leqq i \leqq k$; that is, $x \in \alpha^E$.

Let $x = (x_0, \cdots, x_k)$, $x \in \alpha^E$. Then there is a word $w^E \in \alpha^E$, $w^E = r_0^E?u \cdots ur_k^E?$, $x \in w^E$, $r_i^E \in T_\alpha^A$; that is, $r_i^E = \bigwedge_{j=1}^n q_i^j$, $q_i^j \in T_\alpha \cup \neg T_\alpha$. Since w^E is accepted by $\mathcal{M}_\alpha^E$, it follows that for every i there is a j_i such that $q_i^{j_i} \in T_\alpha$, otherwise $\eta^E(k, \bigwedge_{k=1}^n q_j) = \{e_2\}$. Since $(x_i) \in r_i^E?$, $(x_i) \in q_i^{j_i}?$, $0 \leqq i \leqq k$, and the word $w = q_0^{j_0}?u \cdots uq_k^{j_k}?$ is accepted by $\mathcal{M}_\alpha$. Thus we have $x \in \alpha$. □

DEFINITION. Let $\alpha \in \Gamma_u$ and let $\mathcal{M}_\alpha^E$ be the extended automaton for α.

$$\mathcal{M}_\alpha^E = \langle K_1 \cup K_2 \cup \{d\}, \{u\} \cup T_\alpha^A, k_0, \eta^E, F\rangle.$$

The *deterministic automaton for* α, $\mathcal{M}_\alpha^D$, is defined from the nondeterministic one in the usual way by:

$$\mathcal{M}_\alpha^D = \langle K_1^D \cup K_2^D \cup \{d\}, \{u\} \cup T_\alpha^A, \{k_0\}, \eta^D, F^D\rangle$$

where: $K_1^D = 2^{K_1}$, $K_2^D = 2^{K_2}$, $F^D = \{\bar{k} | \bar{k} \in K_2^D \text{ and } \bar{k} \cap F \neq \varnothing\}$.

For $\bar{k} \in K_1^D$
  $\eta^D(\bar{k}, u) = \{d\}$
  if $p \in T_\alpha^A$ then $\eta^D(\bar{k}, p) = \bigcup\{\eta^E(k, p) | k \in \bar{k}\}$
For $\bar{k} \in K_2^D$
  $\eta^D(\bar{k}, u) = \bigcup\{\eta^E(k, u) | k \in \bar{k}\}$
  if $p \in T_\alpha^A$ then $\eta^D(\bar{k}, p) = \{d\}$
For every $p \in T_\alpha^A \cup \{u\}$, $\eta^D(d, p) = \{d\}$.

Claim 2. Let $\alpha \in \Gamma_u$ and let α^D be any program defining the set of words accepted by $\mathcal{M}_\alpha^D$, the deterministic automaton for α. Then:

1. $\alpha^D \in \Gamma_u$.
2. For every word w over $\{u\} \cup T_\alpha^A$, $w \in \alpha^E$ iff $w \in \alpha^D$.

Proof.

1. Obvious by the form of $\mathcal{M}_\alpha^D$.
2. If $w \in \alpha^D$ then obviously $w \in \alpha^E$.

Suppose $w \in \alpha^E$, $w = r_0?u \cdots ur_k?$. By the form of $\mathcal{M}_\alpha^E$
  $\{\eta^E(k, u)\} \subseteq K_1$, for each $k \in K_2$
  $\{\eta^E(k, p)\} \subseteq K_2$, for each $k \in K_1$, and $p \in T_\alpha^A$
from which it follows that w is accepted by $\mathcal{M}_\alpha^D$, and we have $w \in \alpha^D$. □

To complete the proof of the lemma let $\mathcal{M}_\alpha^D$ be the deterministic automaton for α

$$\mathcal{M}_\alpha^D = \langle K_1 \cup K_2 \cup \{d\}, \{u\} \cup T_\alpha^A, k_0, \eta^D, F\rangle.$$

Define $\tilde{\mathcal{M}}_\alpha$ by

$$\tilde{\mathcal{M}}_\alpha = \langle K_1 \cup K_2 \cup \{d\}, \{u\} \cup T_\alpha^A, k_0, \eta^D, K_2 - F\rangle$$

and let $\tilde{\alpha}$ be a program defined by $\tilde{\mathcal{M}}_\alpha$. Obviously $\tilde{\alpha} \in \Gamma_u$.

Consider a path $x = (x_0, \cdots, x_k)$ such that $x \notin \alpha$. Let $T_i \in T_\alpha^A$ $(0 \leqq i \leqq k)$ be the T_α atom which is true in x_i (obviously there exists a unique element of T_α^A which is true in x_i). Then the word $T_0?u \cdots uT_k?$ is not accepted by $\mathcal{M}_\alpha^D$ $(x \notin \alpha)$. Neither does it lead $\mathcal{M}_\alpha^D$ to the error state d. Hence it leads $\mathcal{M}_\alpha^D$ to a state in $K_2 - F$ and is acceptable by $\tilde{\mathcal{M}}_\alpha$. Therefore $x \in \tilde{\alpha}$.

Suppose $x \in \tilde{\alpha}$ and $x \in \alpha$, then also $x \in \alpha^D$. Let $x = (x_0, \cdots, x_k)$; then:

there is a word $\tilde{w} \in \tilde{\alpha}$,	$\tilde{w} = \tilde{r}_0?u \cdots u\tilde{r}_k?$ and $x \in \tilde{w}$,
there is a word $w \in \alpha^D$,	$w = r_0?u \cdots ur_k?$ and $x \in w$,
$r_i \in T_\alpha^A \Rightarrow r_i = \bigwedge_{j=1}^n q_i^j$,	$q_i^j \in \{p_j, \neg p_j\}$ for every $p_j \in T_\alpha$,
$\tilde{r}_i \in T_\alpha^A \Rightarrow \tilde{r}_i = \bigwedge_{j=1}^n \tilde{q}_i^j$,	$\tilde{q}_i^j \in \{p_j, \neg p_j\}$ for every $p_j \in T_\alpha$.

We know that for every state x_i and every formula $p \in T_\alpha$, either $(x_i) \in p?$ or $(x_i) \in (\neg p)?$ but not both. It follows that $q_i^j = \tilde{q}_i^j$, $1 \leqq j \leqq n$, $0 \leqq i \leqq k$. But then $w = \tilde{w}$ is accepted by both $\mathcal{M}_\alpha^D$ and $\tilde{\mathcal{M}}_\alpha^D$, contradicting the definition of $\tilde{\mathcal{M}}_\alpha^D$. This completes the proof of Lemma 3. □

Proof of Lemma 4.

1. Without loss of generality we can assume $\alpha \in \Gamma$ by Corollary 1. Let $\mathcal{M}_\alpha$ and $\mathcal{M}_\beta$ be the automata defining α, β respectively.

$$\mathcal{M}_\alpha = \langle K_1 \cup K_2 \cup \{d\}, \Pi_0 \cup T_\alpha, k_0, \eta, F\rangle,$$

$$\mathcal{M}_\beta = \langle K_1' \cup K_2' \cup \{d'\}, \{u\} \cup T_\beta, k_0', \eta', F'\rangle.$$

The automaton $\mathcal{M}_{\alpha\cap\beta}$ is defined by:

$$\mathcal{M}_{\alpha\cap\beta} = \langle \hat{K}_1 \cup \hat{K}_2 \cup \{\hat{d}\}, \Pi_0 \cup \{T_\alpha \wedge T_\beta\}, \hat{k}, \hat{\eta}_0, \hat{F}\rangle$$

where

$$\hat{K}_1 = K_1 \times K_1', \qquad \hat{K}_2 = K_2 \times K_2',$$
$$\hat{F} = F \times F', \qquad \hat{k}_0 = (k_0, k_0').$$

A) Let $(k, k') \in \hat{K}_1$
for $p \in \{T_\alpha \wedge T_\beta\}$

$$\hat{\eta}((k, k'), p) = \begin{cases} \eta(k, q) \times \eta'(k', r), & p = q \wedge r, q \neq r, \\ \eta(k, p) \times \eta'(k', p) & p \in T_\alpha \cap T_\beta \end{cases}$$

for $a \in \Pi_0$,

$$\hat{\eta}((k, k'), a) = \{\hat{d}\}$$

B) Let $(k, k') \in \hat{K}_2$

for $p \in \{T_\alpha \wedge T_\beta\}$, $\quad \hat{\eta}((k, k'), p) = \{\hat{d}\}$
for $a \in \Pi_0$, $\quad \hat{\eta}((k, k'), a) = \eta(k, a) \times \eta'(k', u)$
for $p \in \{T_\alpha \wedge T_\beta\} \cup \Pi_0$, $\quad \hat{\eta}(\hat{d}, p) = \{\hat{d}\}$

Let $\alpha \cap \beta$ be a program defined by $\mathcal{M}_{\alpha\cap\beta}$, then $\alpha \cap \beta \in \Gamma$. Let $x = (x_0, \cdots, x_l)$ be a path. Then: $x \in \alpha \cap \beta$ iff there exists a word w in the language defined by $\alpha \cap \beta$ such that $w = r_0?a_0 \cdots a_{l-1}r_l?$, where for each $i, 0 \leqq i \leqq l, (x_i) \in r_i?, (x_i, x_{i+1}) \in a_i$ and $w_i \in T_\alpha \wedge T_\beta$; that is, $\exists p_i \in T_\alpha, q_i \in T_\beta$ such that $r_i = p_i \wedge q_i$ iff $w_\alpha = p_0?a_0 \cdots a_{l-1}p_l?$ is accepted by $\mathcal{M}_\alpha$, $w_\beta = q_0?u \cdots uq_l?$ is accepted by $\mathcal{M}_\beta$ and $x \in w_\alpha, x \in w_\beta \Leftrightarrow x \in \alpha$ and $x \in \beta$.

2. The proof for $\alpha, \beta \in \Gamma_u$ is similar to the proof of part (1) except that the alphabet for $\mathcal{M}_{\alpha\cap\beta}$ is $\{u\} \cup \{T_\alpha \wedge T_\beta\}$ and consequently Π_0 is replaced by $\{u\}$ in the proof. □

Acknowledgment. Two detailed and thoughtful reports by anonymous referees prompted us to prepare a considerably improved revision replacing a rather unreadable first draft.

REFERENCES

[CHMP] A. CHANDRA, J. HALPERN, A. MEYER AND R. PARIKH, *Equations between regular terms and an application to process logic*, 13th ACM Symposium on Theory of Computing, 1981, pp. 384–390.

[D] E. W. DIJKSTRA, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[FL] M. J. FISCHER AND R. E. LADNER, *Propositional dynamic logic of regular programs*, J. Comp. Syst. Sci., 18 (1979), pp. 194–211.

[H] D. HAREL, *Two results on process logic*, Inform. Proc. Letters, 8 (1979), pp. 195–198.

[HKP] D. HAREL, D. KOZEN AND R. PARIKH, *Process logic: expressiveness, decidability, completeness*, J. Comp. Syst. Sci., 25 (1982), pp. 144–170.

[HP] D. HAREL AND D. PELEG, *Process logic with regular formulas*, submitted.

[Ho] C. A. R. HOARE, *An axiomatic basis for computer programming*, Comm. Assoc. Comput. Mach., 12 (1969), pp. 576–580.

[MP] Z. MANNA AND A. PNUELI, *The modal logic of programs*, 6th International Colloquium on Automata, Languages and Programming, Graz, Austria, 1979, Lecture Notes in Computer Science 71, Springer-Verlag, New York, pp. 385–409.

[N] H. NISHIMURA, *Descriptively complete process logic*, Acta Inform., 14 (1980), pp. 359–369.

[Pa] R. PARIKH, *A decidability result for second order process logic*, 19th IEEE Symposium on Foundations of Computer Science, 1978, pp. 177–183.

[Pr1] V. R. PRATT, *Semantical considerations on Floyd-Hoare logic*, 17th IEEE Symposium on Foundations of Computer Science, 1976.

[Pr2] V. R. PRATT, *Process logic*, 6th ACM Symposium on Principles of Programming Languages, 1979, pp. 93–100.

[R] M. O. RABIN, *Decidability of second order theories and automata on infinite trees*, Trans. Amer. Math. Soc., 141 (1969), pp. 1–35.

[S] R. S. STREETT, *Global process logic is undecidable*, Proc. 2nd Conference on Foundations of Software Technology and Theoretical Computer Science, 1982, Bangalore, India.

SIAM J. COMPUT.
Vol. 13, No. 4, November 1984
© 1984 Society for Industrial and Applied Mathematics
012

INFORMATION TRANSFER UNDER DIFFERENT SETS OF PROTOCOLS*

J. JA'JA'†, V. K. PRASANNA KUMAR‡ AND J. SIMON§

Abstract. We study four kinds of protocols in distributed computing. Besides the case of deterministic protocols, we consider *random, nondeterministic* and *probabilistic* models. We show a strict containment relationship with exponential gaps. We also explore in some depth the relationship between *one-way* and *two-way* communications. It is shown, for example, that the complexity classes of *random one-way* protocols and of *deterministic two-way* protocols are incomparable: exponential gaps exist in either direction. On the other hand there is no difference between the complexity of *one-way* and *two-way* communications in the nondeterministic case.

Key words. distributed computation, VLSI, lower bounds, information transfer, communication complexity, probabilistic methods, complexity classes

1. Introduction. The minimum amount of information flow in distributed computations is a fundamental measure of complexity. It is a lower bound on certain practical measures of cost of VLSI circuits, and it is interesting on its own. There seem to be two limitations to our understanding of parallel algorithms, and to writing good parallel programs (assuming that we understand the sequential case at least moderately well):

a) It is difficult to understand which sequentiality constraints (if any) are inherent to a given problem, and which partial results can be computed simultaneously.

b) It is difficult to determine how much information must be exchanged among the separate parallel subcomputations in order to obtain the final result.

The two problems are to a large extent unrelated. In many cases, it is the second that limits the efficiency of parallel algorithms—information flow considerations determine the optimal layout of the processors, the routing of messages, the size of buffers, and, very often, the speed of the algorithm. A quantitative measure of the minimal information flow has been defined and studied by Yao [Y3], [Y4], Ja'Ja' and Kumar [JK], Mehlhorn and Schmidt [MS] and by Papadimitriou and Sipser [PS] and others. We continue this line of research, studying different nonstandard information transfer protocols. In particular, using different models of probabilistic computation we examine the problem of whether probabilistic computations can decrease the amount of communication necessary to solve a problem. The question is specially interesting, since it has been shown that random choices can be very powerful in distributed algorithms [R], [RS1], [RS2] etc. The intuition behind such probabilistic algorithms is that, in distributed computations, a lot of resources must be used to coordinate and synchronize independent subcomputations, that in general, behave in an unknown and unpredictable fashion. Much less coordination is necessary if processes behave similarly, and random choices ensure such a "generic" behavior. It is then to be expected that information

* Received by the editors October 28, 1982, and in revised form July 25, 1983. This research was supported in part by the U.S. Army Research Office, Department of the Army, under contract DAAG29-82-K-0110.

† Department of Computer Science, Pennsylvania State University, University Park, Pennsylvania 16802. Current address: Electrical Engineering Department, University of Maryland, College Park, Maryland 20742.

‡ Department of Electrical Engineering Systems, University of Southern California, Los Angeles, California 90089-0781. The work of this author was also supported by an IBM graduate fellowship.

§ Department of Computer Science, Pennsylvania State University, University Park, Pennsylvania 16802. The work of this author was also supported by the National Science Foundation under grant MCS 81-04876.

transfer will be decreased in some probabilistic computations. Our results bear out this intuition.

Our model is similar to that of [Y3] and [PS]: we consider two processors computing a boolean function of $2n$ boolean variables. Each processor receives n bits of input. The two processors are connected by a communication link. We measure the number of bits through this link. We consider one-way and two-way links, as well as fixed or arbitrary partitions of the $2n$ input bits into two groups of n bits.

We study four kinds of computers (and protocols). Besides the case of deterministic machines we study *probabilistic*, *nondeterministic*, and *random* models. For probabilistic protocols, we allow *each* processor to toss an independent, unbiased coin to decide what message to send, and to use in its computation. We say that the set of two processors computes f *probabilistically* if the probability of the event "the result output by the computation is the value of f for the input given" is greater than $\frac{1}{2}$. This is a very weak notion of probabilistic computation, analogous to the probabilistic (unbounded error) computations discussed in [G], [S].

A more restricted, and more realistic model is that of random protocols, analogous to the sequential random algorithms, studied by Adleman [Ad] and others. Again, the processors can toss coins, but we say that the two processors compute f if the computation outputs a 0 if the value of f is 0, and if the value of f is 1, then a 1 will be output with probability at least $\frac{1}{2}$. In both cases, we count the number of bits exchanged in the worst case—i.e. the number of bits interchanged for any outcome of the coin tosses.

The nondeterministic model was studied in [PS] and [MS]. In this case we allow each processor to make a guess such that, if the value of f is 1, then there exists a guess for which the output produced is equal to 1; otherwise, no such guess exists. Note that, for the *nondeterministic* model there is no loss of generality by assuming that the accept/reject decision is *deterministic*.

One of the reasons for studying these problems is that they provide a parallel communication analogue to the fundamental and very difficult problems in sequential computations, namely proving the strict containment relations among probabilistic, nondeterministic, random and deterministic computations. For the polynomial time complexity classes, it is widely conjectured that $PP \supsetneq NP \supsetneq R \supsetneq P$, but a proof is nowhere in sight. In contrast, it is possible to prove exponential gaps among the corresponding communication complexity classes. These are some of the results of the paper.

Another issue which we explore in some depth is the relationship between *one-way* and *two-way* communications. In *one-way* communication we allow only one processor to send messages to the other; when both processors can exchange information, we call it *two-way* communication. We strengthen the known gap for deterministic algorithms, and obtain some possibly surprising results for other classes of algorithms. It is shown, for example, that the complexity classes of random one-way protocols and of deterministic two-way protocols are incomparable: exponential gaps exist in either direction. On the other hand, there is no difference between the complexity of one-way and two-way communications in the nondeterministic case.

Before closing this section we introduce some notation. We recall that all functions considered in this paper are single output boolean functions. We define the following eight classes.

$D_1(g(n))$: the set of all functions with input of length $2n$ which can be computed with $\leqq g(n)$ bits of information transfer (in the worst case) by a *one-way deterministic* algorithm in the case when the input bits are evenly divided between two processors $P1$ and $P2$.

$D_2(g(n))$: same as $D_1(g(n))$ but by a *two-way deterministic* algorithm.
$ND_1(g(n))$: same as $D_1(g(n))$ but by a *one-way nondeterministic* algorithm.
$ND_2(g(n))$: same as $D_1(g(n))$ but by a *two-way nondeterministic* algorithm.
$R_1(g(n))$: same as $D_1(g(n))$ but by a *one-way random* algorithm.
$R_2(g(n))$: same as $D_1(g(n))$ but by a *two-way random* algorithm.
$P_1(g(n))$: same as $D_1(g(n))$ but by a *one-way probabilistic* algorithm.
$P_2(g(n))$: same as $D_1(g(n))$ but by a *two-way probabilistic* algorithm.

2. Basic characterizations. In this section we give combinatorial characterizations of the amount of communication required by *one-way nondeterministic, random* and *probabilistic* algorithms. In § 4 we explore the relationship of these results to *two-way* communication. Our characterizations lead to optimal bounds.

Let f be a boolean function whose input is specified by $2n$ bits partitioned evenly in an arbitrary fashion between two processors $P1$ and $P2$. Let $M(N)$ be the set of all possible values taken by input bits known to $P1(P2)$. The rectangle R corresponding to this partition is defined to be $M\times N$. A subrectangle $S\times T$, $S\subseteq M$, $T\subseteq N$ is said to be *constant* (monochromatic in [Y3]) if the function remains constant over $S\times T$. It is known [Y3] that, if $\operatorname{nrow}(f)$ is the number of distinct rows of R, then $\log \operatorname{nrow}(f)$ is precisely the amount of information transfer required by *one-way deterministic* algorithms. On the other hand, if we let $d(f)$ be the minimum number of *disjoint constant* rectangles which cover R, then $\log d(f)$ is a lower bound on the information transfer required by *two-way deterministic* algorithms. It is not known whether $O(\log d(f))$ is also an upper bound. It turns out that the nondeterministic case is easier to characterize.

A *1-cover* of R is a collection of constant rectangles $\{S_i\times T_i\}$ which cover all the 1's in R. Note that the rectangles need not be disjoint. We can similarly define the 0-cover for the complement of f. Let $\operatorname{nd}(f)$ be the minimum number of rectangles in any 1-cover of R relative to f. We are ready to prove our first theorem.

THEOREM 2.1. *Let f, R and $\operatorname{nd}(f)$ be as defined above. The amount of information transfer required to compute f by a one-way nondeterministic algorithm is $\Theta(\log \operatorname{nd}(f))$.*

Proof. Let I be the amount of communication required. Let $\{m_1, m_2, \cdots, m_k\}$ be the set of possible strings $P1$ sends to $P2$. It is clear that $k\leqq 2^I$. Define, for $i\in M$ and $j\in N$,

$U(i)$ = the set of all possible messages sent by $P1$ on input i,

$V(j)$ = the set of all messages which can yield 1 as output when $P2$ has j as input.

For $1\leqq l\leqq k$, let

$$S_l=\{i\mid i\in M \text{ and } m_l\in U(i)\},\qquad T_l=\{j\mid j\in N \text{ and } m_l\in V(j)\}.$$

It is clear that $S_l\times T_l$ is a constant rectangle containing 1's, for otherwise there exist $(i,j)\in S_l\times T_l$ such that $f(i,j)=0$ and there is a nondeterministic computation leading to $f(i,j)=1$. On the other hand, whenever $f(i,j)=1$, there exists l such that $(i,j)\in S_l\times T_l$. Hence $k\geqq \operatorname{nd}(f)$ and therefore $I\geqq \log \operatorname{nd}(f)$.

We now derive a matching upper bound. Let $\{S_l\times T_l\}_{l=1}^{k}$ be a 1-cover of R. Given i known to $P1$, i may belong to several $S_l\times T_l$. $P1$ makes a nondeterministic guess and sends the corresponding index to $P2$. If i does not belong to any $S_l\times T_l$ then $P1$ sends the index $k+1$ to $P2$. In either case $P2$ has enough information to complete the computation. Hence the amount of communication required is $\leqq\log(k+1)$. □

We will see in § 4 that the above argument can be modified to prove that the classes $ND_1(g(n))$ and $ND_2(g(n))$ are identical. From the above proof it follows that,

given a 1-cover of size k, there exists a *nondeterministic* protocol computing f using $\log(k+1)$ bits such that,

i) $P1$ is a *nondeterministic* processor while $P2$ is *deterministic*,

ii) at every step the number of nondeterministic choices at $P1$ is at most 2.

This observation will be helpful in simulating a nondeterministic protocol using a probabilistic protocol. The lemma below, the easy proof of which is omitted, is often helpful in establishing lower bounds on the size of 1 and 0-covers.

LEMMA 2.2. *If f has a 1-cover of size k, then*

i) *f has a decomposition into disjoint constant rectangles of size at most 2^{k+1}.*

ii) *f has a 0-cover of size at most 2^k.*

We now turn our attention to *random* and *probabilistic* algorithms. Recall that we are only considering one-way communications in this section. For *random* algorithms, we expand each row i of R by a set of rows $r_1^i, r_2^i, \cdots, r_p^i$ such that the following conditions hold.

i) If $f(i,j)=0$, then the jth entry in r_l^i is 0, for $1 \leqq l \leqq p$.

ii) If $f(i,j)=1$, then at least half the entries in the jth column contain a 1 in the rows $r_1^i, r_2^i, \cdots, r_p^i$.

Let R' be the rectangle that has the vectors r_l^i for its rows and let d_r be the minimum number of distinct rows in any such R'. Then,

THEOREM 2.3. *The number of bits exchanged by a random algorithm computing f is precisely* $\log d_r$.

Proof. It is easy to check that we can assume that $P2$ is deterministic. Given i known to $P1$, $P1$ flips a set of coins, and based on the outcome, sends a message to $P2$. Suppose that there are p possible random outcomes $\{s_t\}$, $1 \leqq t \leqq p$. Then for a given $\{i, s_t\}$, $P1$ behaves just like in the deterministic case. Hence the entire computation can be viewed as a deterministic process with corresponding rectangle R'. □

A similar characterization can be easily given for *probabilistic* algorithms. Expand each row i of the rectangle R into a collection r_l^i of rows and each column j into a collection c_s^j of columns such that if $f(i,j)=1$ then at least half the entries in the corresponding subrectangle are 1's. Similarly for $f(i,j)=0$. Again we let R' be the expanded rectangle. Let d_p be the minimum number of distinct rows in *any* R' which satisfies the above conditions. Then $\log d_p$ is both an upper and lower bound on the information transfer necessary to compute f in a *one-way probabilistic* computation.

3. Relative power of different classes. We study in this section the amount of information transfer reduction introduced by allowing randomization or nondeterminism. Our results will imply a strict hierarchy $D_1(g(n)) \subsetneq R_1(g(n)) \subsetneq ND_1(g(n)) \subsetneq P_1(g(n))$, with exponential gaps. Before establishing the main result of this section, we discuss the case of fixed partition which will illustrate the intriguing relationship between the above classes.

Let $x = x_0x_1 \cdots x_{n-1}$ and $y = y_0y_1 \cdots y_{n-1}$ be two n-bit strings, such that $P1$ has x and $P2$ has y. Define the *equality* function $\delta(x, y)$ and the *boolean inner product* $\alpha(x, y)$ as follows:

$$\delta(x,y) = \begin{cases} 1 & \text{if } x = y, \\ 0 & \text{otherwise,} \end{cases}$$

$$\alpha(x,y) = \begin{cases} 1 & \text{if } \exists i\ 0 \leqq i \leqq n-1 \text{ such that } x_i = y_i = 1, \\ 0 & \text{otherwise.} \end{cases}$$

THEOREM 3.1. *With respect to the fixed partition given above the following hold:*

i) $\delta(x, y) \in P_1(\log n)$ *while* $\delta(x, y)$ *requires* $\Omega(n)$ *bits of communication by any nondeterministic or one-way random algorithm.*

ii) $\alpha(x, y) \in ND_1(\log n)$ *while it requires* $\Omega(n)$ *bits of communication by any one-way random algorithm.*

iii) $\bar{\delta}(x, y) \in ND_1(\log n) \cap R_1(\log n)$, *while* $\bar{\delta}$ *requires* $\Omega(n)$ *bits of communication by any two-way deterministic algorithm.*

Proof. (i) we show that $O(\log n)$ bits of communication are enough to compute $\delta(x, y)$ probabilistically. We use a technique similar to that used in [G]. We use $\log 2n$ unbiased coins. let S be the set of all possible outcomes. Clearly, $|S| = 2n$. Let $S_1 \subseteq S$ be such that $|S_1| = n-1$ and $S_2 = S - S_1$. The algorithm goes as follows:

1) $P1$ tosses $\log 2n$ coins. If the outcome is in S_1, it outputs the value of δ as 0. Otherwise, $P1$ tosses $\log n$ coins whose outcome is used to specify a position i in x. $P1$ then sends i and x_i to $P2$.

2) When $P2$ receives i and x_i, he checks to see if $x_i = y_i$. If this is the case, $P2$ declares $\delta = 1$; otherwise, $\delta = 0$.

We now show that the above algorithm works correctly. Suppose $x = y$, then the correct result is output whenever the coin tossing results in an outcome in S_2. Hence,

$$\Pr(\text{output} = 1) = \frac{n+1}{2n} = \frac{1}{2} + \frac{1}{2n} > \frac{1}{2}.$$

Suppose that $x \neq y$ then we obtain the correct result if the outcome is in S_1 or the outcome is in S_2 and i is the bit position in which x and y differ. Hence,

$$\Pr(\text{output} = 0) \geqq \frac{n-1}{2n} + \frac{n+1}{2n}\left(\frac{1}{n}\right) = \frac{1}{2} + \frac{1}{2n^2} > \frac{1}{2}.$$

Thus $\delta(x, y) \in P_1(\log n)$. On the other hand, the corresponding rectangle consists of 1's along the diagonal and 0's everywhere else. Hence the number of *constant* rectangles must be at least 2^n and thus $\delta(x, y)$ requires $\Omega(n)$ bits of communication by any *nondeterministic* or *random* algorithm.

(ii) $P1$ just guesses i such that $x_i = 1$ and sends i to $P2$. $P2$ checks to see if $y_i = 1$ and, if this is the case, outputs a 1; otherwise a 0. Hence $\alpha(x, y) \in ND_1(\log n)$.

We use Theorem 2.3 to show that $\Omega(n)$ bits of communication are required by any random algorithm. Let R be the corresponding rectangle and let R_1 be the subrectangle obtained by restricting y to have exactly one 1. It is clear that R_1 consists of n columns and 2^n rows and that the rows correspond to the set of all strings of length n. Let $R_1^{n/2}$ be the set of rows of R_1 such that each contains exactly $n/2$ 1's. We show that any expansion R_1' of R_1, as outlined before the statement of Theorem 2.3, has at least $2^{n/4}$ distinct rows.

Suppose a row in $R_1^{n/2}$ is replaced by l rows in R_1'. Then one of these rows must have at least $k \geqq n/4$ 1's. This row can only appear in the blocks corresponding to rows with 1's in these k positions. The number of such rows is

$$\binom{n-k}{\frac{n}{2}-k} = \binom{n-k}{\frac{n}{2}} \leqq \binom{n-\frac{n}{4}}{\frac{n}{2}} = \binom{\frac{3}{4}n}{\frac{n}{2}}.$$

Hence the number of distinct rows in R_1' is at least

$$\binom{n}{\frac{n}{2}} \Bigg/ \binom{\frac{3n}{4}}{\frac{n}{2}}$$

which can be shown by simple algebraic manipulations to be $\geqq 2^{n/4}$.

(iii) It is obvious that $\bar{\delta}(x, y) \in ND_1(\log n)$. To show that $\bar{\delta}(x, y) \in R_1(\log n)$, we use a well-known technique ([P], [Y2], [MS]). $P1$ randomly selects a prime p_i, $0 < p_i \leqq n$, and sends the residue $x \bmod p_i$ to $P2$ together with the index i of p_i. The desired result follows from the following fact: Let $p_1, p_2, \cdots, p_l$ be the primes $\leqq n$; if $x, y \in \{0, 1, \cdots, 2^n - 1\}$, $x \neq y$, then $|\{j \mid x \not\equiv y \bmod p_j\}| \geqq l/2$. As for the last part, it is easy to check that any decomposition of the corresponding rectangle (as shown in Fig. 1) has 2^n *disjoint constant* rectangles. □

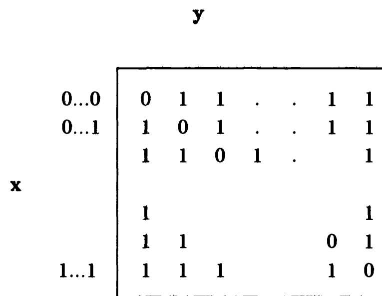


FIG. 1

There is a weak point about the above examples. The results heavily depend on the partition of the input and as a matter of fact, if the input is properly partitioned, then δ and $\bar{\delta}$ could be computed with constant amount of communication by a deterministic algorithm. In the remainder of this section, we show the strict hierarchy by exhibiting examples for which the corresponding results hold regardless of the partition of the input bits.

To this end we introduce *shifted equality* function γ and *shifted inner product* function β whose input consists of an m-bit string x and a $\log m$-bit selection input i. Without loss of generality we assume that m and $\log m$ are even. Let $2n = m + \log m$. Then γ and β are defined as follows:

$$\gamma(x, i) = \begin{cases} 1 & \text{if } x_0 x_1, \cdots, x_{m/4-1} = x_i x_{(i+1) \bmod m}, \cdots, x_{(i+m/4-1) \bmod m}, \\ 0 & \text{otherwise,} \end{cases}$$

$$\beta(x, i) = \begin{cases} 1 & \text{if } \exists j,\ 0 \leqq j \leqq m/4 - 1, \text{ s.t. } x_j = x_{(i+j) \bmod m} = 1, \\ 0 & \text{otherwise.} \end{cases}$$

THEOREM 3.2. *Regardless of how the input (x, i) is partitioned equally between two processors $P1$ and $P2$, we have the following*:

(i) $\bar{\gamma}(x, i) \in R_1(\log n) \cap ND_1(\log n)$, *while it requires $\Omega(n)$ bits of communication by any two-way deterministic algorithm.*

(ii) $\beta(x, i) \in ND_1(\log n)$ *but $\Omega(n)$ bits of communication are required by any random algorithm.*

(iii) $\gamma(x, i) \in P_1(\log n)$ *but it requires* $\Omega(n)$ *bits of communication by any nondeterministic or random algorithm.*

Proof. (i) We prove the upper bound for all partitions of (x, i) such that $P1$ has i. In the next paragraph, we prove the lower bounds hold for all partitions. The problem now involves checking if $y \neq z$, where y and z are $m/4$ bit integers obtained from x. Suppose $P1$ has y_1 and z_1 and $P2$ has y_2 and z_2, such that $y = y_1 + y_2$ and $z = z_1 + z_2$. $P1$ randomly chooses a prime p_j, $0 < p_j \leqq m/4$ and sends the residues $y_1 \bmod p_j$ and $z_1 \bmod p_j$ to $P2$ together with the indices j and i. As in the proof of Theorem 3.1(iii), one can show that the random algorithm above works correctly and involves the transmission of $O(\log n)$ bits from $P1$ to $P2$. On the other hand, it is easy to check that $\bar{\gamma}(x, i) \in ND_1(\log n)$.

We now prove the following claim.

CLAIM. *Regardless of how the bits of* (x, i) *are partitioned between* $P1$ *and* $P2$, $\bar{\gamma}(x, i)$ *requires* $\Omega(n)$ *bits of communication by any two-way deterministic algorithm.*

Proof of the claim. Consider the set of indices $\{0, 1, \cdots, m/4-1\}$ of x. Without loss of generality assume that $P1$ has at least $m/8$ of these indices, say $i_1, i_2, \cdots, i_p$, $p \geqq m/8$. Construct the table shown in Fig. 2 as follows. The (i_k, l)th entry is 1 if $(i_k + l) \bmod m$ corresponds to an index in $P1$ and 0 otherwise.

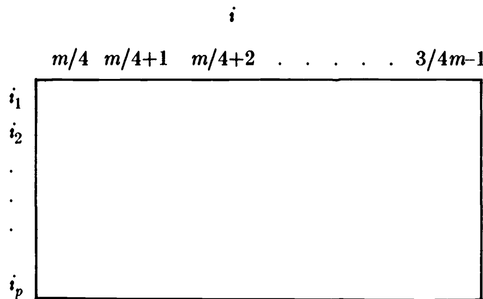


FIG. 2

Since $P1$ has p indices in $\{0, 1, \cdots, m/4-1\}$, given any i_k, the row i_k has at least $m/2-((m+\log m)/2-p)$ 0's. Therefore the table contains at least $s = p(m/8-\log m/2)$ 0's. But $s = p(2n-5\log m)/8 \geqq pn/8 \geqq mn/64$, for n sufficiently large. Hence there exists a column say l with at least $n/32$ 0's. Considering the subfunction $\bar{\gamma}(x, l)$, it is clear that $P1$ and $P2$ have to compare two $n/32$ bit integers, which requires $\Omega(n)$ bits of communication by any *two-way deterministic* algorithm.

(ii) Without loss of generality, we assume that $P1$ has all the bits of i. If $P1$ has indices j and $(i+j) \bmod m$ such that $x_j = x_{(i+j) \bmod m} = 1$, then $P1$ outputs 1. Otherwise it guesses such a j and sends i, j to $P2$. Hence $\beta(x, i) \in ND_1(\log n)$.

Using the same technique as in the proof of the claim in part (i) and Theorem 3.1(ii), it is easy to see that $\beta(x, i)$ requires $\Omega(n)$ bits of communication by any *one-way random* algorithm.

(iii) It is easy to generalize the proof of Theorem 3.1(i) so that it works for arbitrary partition of (x, i). $\square$

Before stating the main theorem of this section, we show the following lemma.

LEMMA 3.3. *The following inclusions hold*:

$$D_1(g(n)) \subseteq R_1(g(n)) \subseteq ND_1(g(n)) \subseteq P_1(g(n))$$

for any function $g(n) \leqq n$.

Proof. The first two inclusions are obvious. The inclusion $ND_1(g(n)) \subseteq P_1(g(n))$ can be shown as follows: $P1$ tosses $g(n)+1$ coins and outputs a 1 for $2^{g(n)}-1$ outcomes, while for the remaining outcomes, it simulates the nondeterministic algorithm by substituting coin tossings for guesses. The rest of the proof is similar to that of Theorem 3.1(i) (cf. [G]). □

THEOREM 3.4. *Let $g(n)$ be any function such that $\log n \leqq g(n) < n$. Then we have the following strict hierarchy*:

$$D_1(g(n)) \subsetneq R_1(g(n)) \subsetneq ND_1(g(n)) \subsetneq P_1(g(n))$$

where exponential gaps exist.

Proof. Immediate from Theorem 3.2 and Lemma 3.3. □

4. One-way versus two-way communications. We have considered one-way communications in the previous sections and we turn our attention to two-way communications. A result is known in the literature [PS] which shows that *two-way deterministic* communication is more powerful than *one-way deterministic* communication. We strengthen this result and show several such results for random and nondeterministic algorithms.

Fig. 3 defines a function $\lambda(x, y)$ which will be used repeatedly in this section. Note that the length of the input is $2(k+1)$.

Suppose $P1$ has value x_0 of x and $P2$ has a value y_0 of y. Then, with two-way communication, $P1$ and $P2$ can in $O(1)$ bits of communication determine one of the six subrectangles which contains (x_0, y_0). Then in at most $\log k$ bits, they can compute the value of $\lambda(x_0, y_0)$. However since the rectangle has 2^k+k+1 distinct rows, $\Omega(k)$ bits of communication are required by any *one-way deterministic* algorithm.

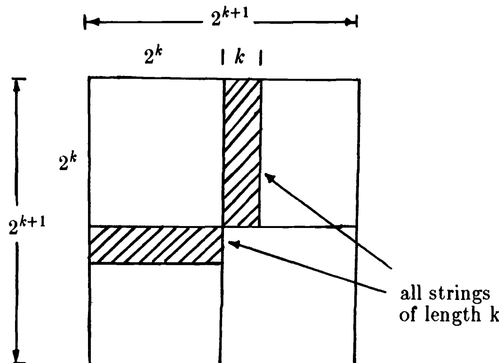


FIG. 3

We now generalize the function λ to the following function $f(x, i)$:

$$f(x, i) = \lambda(x_0 x_1, \cdots, x_{m/4-1}, x_i x_{(i+1) \bmod m}, \cdots, x_{(i+m/4-1) \bmod m})$$

where x is an m bit integer and i is a $\log m$ bit integer.

THEOREM 4.1. *Let $g(n)$ be any function such that $\log n \leqq g(n) < n$. Then*:

(i) *$D_1(g(n)) \subsetneq D_2(g(n))$ with exponential gap.*

(ii) *$D_2(g(n))$ and $R_1(g(n))$ are incomparable and exponential gaps exist in each direction.*

(iii) *$R_1(g(n)) \subsetneq R_2(g(n))$ with exponential gap.*

(iv) *$ND_1(g(n)) = ND_2(g(n))$.*

(v) *$D_2(g(n)) \subsetneq R_2(g(n))$ with exponential gap.*

(vi) *$ND_2(g(n)) \subsetneq P_2(g(n))$ with exponential gap.*

Proof. (i) It is not hard to generalize the argument before statement of the theorem to show that regardless of how (x, i) is partitioned between $P1$ and $P2$, $O(\log n)$ bits of communication are enough to compute $f(x, i)$ by a *two-way deterministic* algorithm. On the other hand using the claim shown in Theorem 3.2(i) it is easy to see that the corresponding rectangle has $\Omega(2^{cn})$ distinct rows, for some constant $c>0$ (regardless of the partition of (x, i)) and hence $\Omega(n)$ bits of information transfer are required by any *one-way deterministic* algorithm.

(ii) Using the claim established in Theorem 3.2(i) and the technique outlined in Theorem 3.1(ii), one can show that $f(x, i)$ requires $\Omega(n)$ bits of communication by any *one-way random* algorithm. Theorem 3.2(i) finishes the proof of this part.

(iii) It is clear that $D_2(g(n)) \subseteq R_2(g(n))$. Hence $f(x, i) \in R_2(\log n)$.

(iv) We show that the lower bound established in Theorem 2.1 for ND_1 will also hold for ND_2. Define for $i \in M$ and $j \in N$,

$U(i)$ = the set of all possible messages exchanged between $P1$ and $P2$ on input i,

$V(j)$ = the set of all possible messages which can yield 1 as output when $P2$ has j as input.

For $q \leqq l \leqq k$, let

$$S_l = \{i \mid i \in M \text{ and } m_l \in U(i)\}, \qquad T_l = \{j \mid j \in N \text{ and } m_l \in V(j)\}.$$

The rest of the proof follows as in the proof of Theorem 2.1. (v) follows from (ii). (vi) follows from (iv) and Theorem 3.2(iii). □

We conclude this section by sketching a diagram representing most of our results.

$$\begin{array}{ccccccc} D_2 & \subsetneq & R_2 & \overset{?}{\subseteq} & ND_2 & \subsetneq & P_2 \\ \rotatebox{90}{$\subsetneq$} & & \rotatebox{90}{$\subsetneq$} & & \| & & \rotatebox{90}{$\subseteq$}? \\ D_1 & \subsetneq & R_1 & \subsetneq & ND_1 & \subsetneq & P_1 \end{array}$$

Exponential gaps exist between all strict inclusions. We conjecture that exponential gaps exist in the two inclusions $R_2 \subseteq ND_2$ and $P_1 \subseteq P_2$.

5. The complement classes. The main result of this section is stated in the following theorem.

THEOREM 5.1.

i) *For any function $g(n) \leqq n$, $P(g(n))$ and $D(g(n))$ are closed under complement.*[1]

ii) *For any function $g(n)$, $\log n \leqq g(n) < n$, $ND(g(n))$ and $R(g(n))$ are not closed under complement.*

Proof. i) is trivial and ii) follows from Theorem 3.2. □

6. Acknowledgment. We would like to thank the referees for their constructive comments.

REFERENCES

[Ab] H. ABELSON, *Lower bounds on information transfer in distributed computations*, Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, Ann Arbor, MI, 1978, pp. 151–158.

[Ad] L. ADLEMAN, *Two theorems on random polynomial time*, Proc. 11th Annual ACM Symposium on Theory of Computing, Atlanta, GA, 1979, pp. 75–83.

[1] The omission of the subscripts means that the statement is true for both one-way and two-way communications.

[BK] R. P. BRENT AND H. T. KUNG, *The chip complexity of binary arithmetic*, Proc. 12th Annual ACM Symposium on Theory of Computing, Los Angeles, 1980, pp. 190–200.

[CM] B. M. CHAZELLE AND L. M. MONIER, *A model of computation for* VLSI *with related complexity results*, Proc. 13th Annual ACM Symposium on Theory of Computing, Milwaukee, WI, 1981, pp. 318–325.

[G] J. GILL, *Computational complexity of probabilistic Turing machines*, this Journal, 6 (1977), pp. 675–695.

[JK] JOSEPH JA'JA' AND V. K. PRASANNA KUMAR, *Information transfer in distributed computing with applications to* VLSI, Tech. Report CS-81-14, Pennsylvania State Univ., University Park, PA, July 1981; J. Assoc. Comput. Mach., to appear.

[LS] R. J. LIPTON AND R. SEDGEWICK, *Lower bounds for* VLSI, Proc. 13th Annual ACM Symposium on Theory of Computing, Milwaukee, WI, 1981, pp. 300–307.

[MS] K. MEHLHORN AND E. SCHMIDT, *Las Vegas is better than determinism in* VLSI *and distributed computing*, Proc. 14th Annual ACM Symposium on Theory of Computing, San Francisco, 1982, pp. 330–337.

[PS] C. H. PAPADIMITRIOU AND M. SIPSER, *Communication complexity*, Proc. 14th Annual ACM Symposium on Theory of Computing, San Francisco, 1982, pp. 196–200.

[P] N. PIPPENGER, *The power of randomization and nondeterminism for one-tape machines*, IBM research report, Yorktown Heights, NY, 1978.

[R] M. O. RABIN, *Probabilistic algorithms*, in Algorithms and Complexity: New Directions and Recent Results, Traub ed., Academic Press, New York, 1976, pp. 21–39.

[RS1] J. H. REIF AND P. SPIRAKIS, *Distributed algorithms for synchronizing interprocess communication within real time*, Proc. 13th Annual ACM Symposium on Theory of Computing, Milwaukee, WI, 1981, pp. 133–145.

[RS2] ———, *Unbounded speed variability in distributed communication systems*, Proc. 9th ACM Symposium on Principles of Programming Languages, Albuquerque, NM, 1982, pp. 46–56.

[S] J. SIMON, *On some central problems in computational complexity*, Ph.D. thesis, Cornell University, Ithaca, NY, 1975.

[T] C. D. THOMPSON, *Area time complexity for* VLSI, Proc. 11th Annual ACM Symposium on Theory of Computing, Atlanta, GA, 1979, pp. 81–88.

[V] J. VUILLEMIN, *A combinatorial limit to the computing power of* VLSI *circuits*, Proc. 21st Annual IEEE Symposium on Foundations of Computer Science, Syracuse, NY, 1980, pp. 296–300.

[Y1] A. C. YAO, *Probabilistic computations: toward a unified measure of complexity*, Proc. 18th Annual IEEE Symposium on Foundations of Computer Science, Providence, RI, 1977, pp. 222–227.

[Y2] ———, *A lower bound to palindrome recognition by probabilistic Turing machines*, Tech. Report CS-77-647, Stanford Univ., Stanford, CA, 1977.

[Y3] ———, *Some complexity questions related to distributive computing*, Proc. 11th Annual ACM Symposium on Theory of Computing, Atlanta, GA, 1979, pp. 209–213.

[Y4] ———, *The entropic limitations on* VLSI *computations*, Proc. 13th Annual ACM Symposium on Theory of Computing, Milwaukee, WI, 1981, pp. 308–311.

SIAM J. COMPUT.
Vol. 13, No. 4, November 1984

© 1984 Society for Industrial and Applied Mathematics
013

HOW TO GENERATE CRYPTOGRAPHICALLY STRONG SEQUENCES OF PSEUDO-RANDOM BITS*

MANUEL BLUM† AND SILVIO MICALI‡

Abstract. We give a set of conditions that allow one to generate 50–50 unpredictable bits.

Based on those conditions, we present a general algorithmic scheme for constructing polynomial-time deterministic algorithms that stretch a short secret random input into a long sequence of unpredictable pseudo-random bits.

We give an implementation of our scheme and exhibit a pseudo-random bit generator for which any efficient strategy for predicting the next output bit with better than 50–50 chance is easily transformable to an "equally efficient" algorithm for solving the discrete logarithm problem. In particular: if the discrete logarithm problem cannot be solved in probabilistic polynomial time, no probabilistic polynomial-time algorithm can guess the next output bit better than by flipping a coin: if "head" guess "0", if "tail" guess "1".

Key words. randomness, pseudo-random number generation, unpredictability, random self-reducibility

1. Introduction.

1.1. Randomness and complexity theory.

We introduce a new method of generating sequences of pseudo-random bits. Any such method implies, directly or indirectly, a definition of randomness.

Much effort has been devoted in the second half of this century to make precise the notion of randomness. Let us informally recall Kolmogorov's influential definition [18]:

> A sequence of bits $A = a_1, a_2, \cdots, a_k$ is random if the length of the minimal program outputting A is at least k.

We remark that the length of a program, from a computational complexity point of view, is a rather unnatural measure. We want to investigate a more operative definition of randomness in the light of complexity theory.

A mental experiment. A and B want to play head and tail in four different ways. In all of them A "fairly" flips a "fair" coin. In the first way, A asks B to bet and then flips the coin. In such a case we expect B to win with a 50% frequency. In the second way, A flips the coin and, while it is spinning in the air, she asks B to bet. We are still expecting B to win with a 50% frequency. However, in the second case the outcome of the toss is determined when B bets: in principle, he could solve the equation of the motion and win! The third way is similar to the second one: B is allowed to bet when the coin is spinning in the air, but he is also given a pocket calculator. Nobody will doubt that B is still going to win with 50% frequency: before he can initialize any computation the coin will have come up head or tail. The fourth way is similar to the third, except that now B is given a very powerful computer, able to take pictures of the spinning coin, and quickly compute its speed, momentum, etc. We will not say that B will always win, but we may suspect he may win 51% of the time!

The purpose of the above example is to suggest that

> *The randomness of an event is relative to a specific model of computation with a specified amount of computing resources.*

* Received by the editors April 11, 1983, and in final form January 15, 1984. Supported in part by National Science Foundation grant MCS 82-04506 and by a fellowship of Consiglio Nazionale delle Ricerche-Italy. A version of this paper was presented at the AMS Conference on Probabilistic Computational Complexity, June 1982, Durham, New Hampshire and in the 23rd FOCS, November 1982, Chicago, Illinois.

† Computer Science Department, University of California, Berkeley, California 94720.

‡ Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

The links between randomness and the computation model were pointed out by Michael Sipser in [31], where he defines randomness with respect to finite state automata (see also [24]). In his very nice paper [29], Shamir considers also the factor of the computing resources, presents significant progress in this direction and points out some open problems as well.

In this paper we investigate the randomness of k-bit long sequences with respect to the computation model of *Boolean circuits with only* Poly (k) *gates.*

1.2. CSPRB generators. We introduce the notion of a cryptographically strong pseudo-random bit generator (CSPRB generator) and show under which conditions it can be constructed. A CSPRB generator is a program G that, upon receiving as input a random number i (hereafter referred to as the *seed*), outputs a sequence of pseudo-random bits $b_1, b_2, b_3, \cdots$. G possesses the following properties:

1) *The bits b_k's are easy to generate.* Each b_k is output in time polynomial in the length of the seed.

2) *The bits b_k's are unpredictable.* Given the generator G and $b_1, \cdots, b_s$, the first s output bits, *but not the seed i*, it is computationally infeasible to predict the $s+1$st bit in the sequence with better than 50–50 chance. Here s is polynomial in the length of the seed.

Our generators are an improvement of Shamir's pseudo-random number generators. In [29], Shamir presents programs that from a short secret random seed, output a sequence of numbers x_i's such that the ability of predicting the next output is equivalent to inverting the RSA function [27]. The main difference between ours and Shamir's generator is:

Shamir's generator outputs numbers and not bits. Such numbers could be unpredictable and yet of very special form. In particular every bit of (information about) the next number in the sequence could be heavily biased or predictable with high probability. As a consequence, if the numbers so generated are 100 bits long, they might not be uniformly spread in the interval $[1, 2^{100}]$.

1.3. Pseudo-random sequences and statistical tests. Passing given statistical tests is the key point for evaluating pseudo-random sequences. The classical sequence $x_{i+1} = ax_i + b \bmod n$, provides a fast way of generating pseudo-random numbers. Such a sequence is known (for a clever choice of the parameters a, b, and n) to generate "well mixed numbers" (see Knuth [17]). However it is not cryptographically strong. Plumstead [25] shows that the sequence can be inferred even when a, b and n are all unknown.

In contrast, our bit-sequences cannot be generated as fast, but cannot be inferred either. This is so because they have "embedded" some hard problem.

An analysis of a particularly simple pseudo-random number generator appears in Blum, Blum and Shub [9]. They point out that *well mixed sequences* in which *hard problems are embedded* can nevertheless be poor pseudo-random sequences. Something more is needed to construct good generators of pseudo-random sequences; what that is is pointed out in § 2.

Unpredictability of the next output bit is the key test studied in this paper. In an earlier version of this paper [10], we presented a deterministic polynomial-time algorithm that stretched a random k-bit long seed into a polynomially (in k) long bit-sequence. Any probabilistic polynomial-time algorithm, correctly predicting the next bit with probability greater than $\frac{1}{2}+\varepsilon$ in a so produced pseudo-random sequence, could be easily transformed to a probabilistic algorithm, running in expected

poly (k, ε^{-1}) time, for solving the discrete logarithm problem for a fraction ε of the primes of length k.

Though we were aware of the polynomial dependency on ε^{-1} (in fact in [10] Lemma 2 explicitly states it), our main theorem summarized our results stating that the next bit in our pseudo-random sequences could not be predicted in polynomial (in k) time with probability greater than $\frac{1}{2}+\varepsilon$, for $0<\varepsilon<1$.

Yao [33] was the first one to realize the importance of emphasizing the polynomial functional dependency on ε^{-1} and made excellent use of it (see § 1.3.2).

Indeed, without any changes in our algorithm, ε could be replaced with the smallest value that will keep the running time polynomial. Since in our case the running time is polynomial in k and ε^{-1}, we henceforth use $1/\text{poly}(k)$ for ε in this paper.

We now proceed to a formal treatment.

1.3.1. The next-bit-test. Let P be a polynomial and $S=\{S_k\}$ be a collection of multisets such that S_k contains $P(k)$-bit long sequences (the same sequence s may belong more than once to S_k). Let P_1 be a polynomial. A *predicting collection* $C=\{C_k^i\}$ is a collection of circuits such that each circuit C_k^i has less than $P_1(k)$ gates, i Boolean inputs, $i<P(k)$, and one Boolean output. On input the first i bits of a sequence s randomly selected in S_k, C_k^i will output a bit b. Let $p_{k,i}^C$ denote the probability that $b=$ the $i+1$st bit of s. We say that the collection S *passes the next-bit-test* if for all predicting collections C, all polynomials Q, all sufficiently large k and all $i<P(K)$,

$$p_{k,i}^C<\frac{1}{2}+\frac{1}{Q(k)}.$$

Ability to predict the next bit from the preceding ones is indeed a statistical test for a bit-sequence. In fact, if a bit-sequence were generated by independent coin flips, no strategy would predict the next bit with a success rate even slightly better than 50–50. This particular test is passed by the sequences produced by a CSPRB generator. Therefore CSPRB *generators produce evenly distributed numbers*: just divide the output sequences into k-bit long segments.

Subsequently, Yao [33] showed the following very interesting result.

1.3.2. Yao's statistical test. The following definition is derived from Yao [33]. As before, the collection $S=\{S_k\}$ is such that the multiset S_k contains $P(k)$-bit long sequences.

Let P_1 be a polynomial. A *polynomial-size statistical test* is a collection $C=\{C_k\}$ of circuits. Each circuit C_k has less than $P_1(k)$ gates, $P(k)$ Boolean inputs and one Boolean output. Let $p_{k,S}^C$ denote the probability that C_k outputs 1 on input a randomly selected sequence in S_k, and $p_{k,R}^C$ the probability that C_k outputs 1 on a randomly selected $P(k)$-bit long sequence. The collection S *passes all polynomial-size statistical tests* if for all polynomial-size statistical test C, for any polynomial Q, for all sufficiently large k,

$$|p_{k,S}^C-p_{k,R}^C|<\frac{1}{Q(k)}.$$

THEOREM (Yao). *A collection $S=\{S_k\}$ passes the next-bit-test if and only if it passes all polynomial-size statistical tests.*

1.3.3. Related tests for strings. Earlier definitions and theorems about tests for distinguishing strings belonging to two different sets can be found in Goldwasser and Micali [13]. They presented a probabilistic encryption scheme in which a single bit b

is, with the help of a coin, encoded by a k-bit long string β, called a *probabilistic encryption of b*. Here k is a security parameter. Both 0 and 1 will have many possible probabilistic encodings, but all of them are uniquely decodable. They defined a probabilistic encryption scheme to be *bit-secure* if for all polynomials P and Q, for all sufficiently large k, no circuit with less than $P(k)$ gates can correctly guess whether β is the encryption of 0 or 1 with probability greater than $\frac{1}{2}+1/Q(k)$. Under an intractability assumption for the quadratic residuosity problem, they present a probabilistic encryption scheme that is bit-secure.

A probabilistic encryption of an n-bit long ($n < P_1(k)$ for some polynomial P_1) string $b_1, \cdots, b_n$ is a sequence $\beta_1, \cdots, \beta_n$ where each β_i is a probabilistic encryption of b_i. Let P be a polynomial. A *separator* is defined to be a collection of circuits $C = \{C_k\}$. Each circuit C_k has less than $P(k)$ gates, kn Boolean inputs and one Boolean output. For a string $s = s_1, \cdots, s_n$ let $p^C_{s,k}$ denote the probability that C_k outputs 1 on input a probabilistic encryption of s. The encryptions of the string $x = x_1, \cdots, x_n$ are *unseparable* from the encryptions of a string $y = y_1, \cdots, y_n$ if for all separators C and for all polynomials Q, for sufficiently large k,

$$|p^C_{x,k} - p^C_{y,k}| < \frac{1}{Q(k)}.$$

The computational difficulty of separating the encryptions of polynomially long bit-sequences reduces to the one of correctly guessing the decoding of an encrypted single bit.

THEOREM (Goldwasser and Micali). *For any pair of n-bit long strings x and y, the encryptions of x and y are unseparable if and only if the encryption scheme is bit-secure.*

1.4. Instances of the CSPRB generator model. A general algorithmic scheme for constructing CSPRB generators is presented in § 2. The first instance of this scheme is based on the intractability assumption for the discrete logarithm problem and is described in § 4. Other interesting instances of the general model have subsequently been found based on the intractability assumption of various one-way functions. Yao [33] and Blum, Blum and Shub [9] found instances based on the intractability of deciding quadratic residuosity modulo composite numbers whose factorization is unknown. Yao [33] and Goldwasser, Micali and Tong [14] implemented CSPRB generators based on the intractability of factoring. Yao [33] also proves that one can obtain instances of the CSPRB generator scheme if one-way functions with a particular property exist.

1.5. Applications. Recently, a large number of cryptographic protocols for protecting private communication and business transactions have been developed [7], [8], [13], [14], [15], [20]. The security of these new protocols is based both on the security of some encryption scheme and the ability of the participants to generate large random numbers unknown to an adversary. Security vanishes if an adversary, though not able to break the encryption scheme, can successfully predict the output of the pseudo-random number generator. This is not an abstract worry as shown by Plumstead. The problem calls for the use of CSPRB generators.

In private key cryptography, one-time pads constitute the best type of cryptosystem. In practice, one-time pads are approximated by pseudo-random number generators [4]. Shamir [29] points out that "unpredictable" pseudo-random number generators may be a valid substitute for one-time pads. Therefore, CSPRB generators are particularly good substitutes for one-time pads: two partners who both possess

the same CSPRB generator and have *secretly* exchanged a random seed, are actually sharing a long bit-sequence that can be successfully used as a one-time pad.

The fact that the cryptographic strength of CSPRB generators depends only on the secrecy of the seed and not on the secrecy of the program, makes them an available tool to mathematically nonsophisticated users. In fact, it is unreasonable to assume that a business person should be able to design a cryptographically strong generator. However, anyone can buy such a program from the public market and secretly select a short random seed.

2. A general algorithmic scheme for constructing CSPRB generators. In this section we present the formal definition of a CSPRB generator and a theorem showing a set of conditions that allow one to construct such generators. In § 3 we show some results about the discrete logarithm problem; namely that there exists a Boolean predicate whose computational difficulty is equivalent to that of the discrete logarithm problem. In § 4 we show that these results make possible, under the intractability assumption for the discrete logarithm problem, to concretely implement CSPRB generators. In § 5 we explicitly define the notion of random self-reducibility that is the basis of our results.

DEFINITION. Let Q be a polynomial, I a set of inputs and $I_k \subseteq I$ the set of inputs of length k. Let A be a deterministic algorithm that, on input a seed $x \in I_k$, outputs a $Q(k)$-bit sequence s_x. Let $S_k = \{s_x | x \in I_k\}$. The algorithm A is a Q-CSPRB *generator* if the collection $S = \{S_k\}$ passes the next-bit-test.

The sequences output by a CSPRB generator will be called the CSPRB sequences. CSPRB sequences are ultimately periodic. However, for the great majority of the seeds, the corresponding CSPRB sequences do not quickly become periodic with a short period.

Let α and β be integers. We say that a bit-sequence is (α, β)-periodic if it becomes periodic, with period length less than β, after at most α bits.

THEOREM 1. *Let P_1, P_2 and P_3 be polynomials. Set $Q = P_1 + P_2 + 2P_3 + 1$ and let G be a Q-CSPRB generator. Let δ_k denote the fraction of the seeds of length k for which G generates a $(P_1(k), P_2(k))$-periodic pseudo-random sequence. Then $\delta_k < 1/P_3(k)$ for all sufficiently large k.*

Proof. Assume, for contradiction, that $\delta_k \geqq 1/P_3(k)$ for each $k \in F$ where F is infinite. Let $k \in F$. Denote by ε_i the fraction of seeds of length k for which the first $P_1(k) + P_2(k) + i$ bits in the corresponding CSPRB sequence form a $(P_1(k), P_2(k))$-periodic sequence. Then we have

$$1 = \varepsilon_0 \geqq \varepsilon_1 \geqq \cdots \geqq \varepsilon_{2 \cdot P_3(k)} \geqq \delta_k \geqq \frac{1}{P_3(k)}.$$

Let the integer $i \in [0, 2 \cdot P_3(k))$ be such that $\varepsilon_i - \varepsilon_{i+1} \leqq \frac{1}{2}(1/P_3(k))$. (Such an i exists, otherwise $\varepsilon_0 > 1$.) Consider the following algorithm A_k that predicts the $(i+1)$st bit in a CSPRB sequence $b_1, b_2, \cdots, b_{Q(k)}$ produced with a seed of length k:

> Look at $S = b_1, \cdots, b_i$. If S is not a $(P_1(k), P_2(k))$-periodic sequence, predict b_{i+1} by flipping a coin. Else, predict b_{i+1} so to preserve the $(P_1(k), P_2(k))$-periodicity.

Because of $\varepsilon_i \geqq 1/P_3(k)$ and our choice of i, A_k will predict b_{i+1} correctly with probability greater than $\frac{1}{2} + 1/(2 \cdot P_3(k))$. We have reached a contradiction as, for some polynomial P, for each $k \in F$, A_k can be transformed to a circuit C_k, with less than $P(k)$ gates, that accomplishes the same task. Q.E.D.

We just mention that we could replace the above algorithms A_k by a uniform probabilistic polynomial time algorithm that makes use of sampling.

2.1. Problems in building generators. Let B be a predicate defined in a domain D with 2^k elements, such that $B(x)=1$ for half of the x's in D. Then we could generate random bits $b_0, b_1, \cdots$ by picking x_i at random in D and outputting $b_i = B(x_i)$. The drawback of this method is that we need to use k random bits to pick each x_i, but we generate a single bit b_i. Instead, we would like to pick only the first x_0 at random in D and to select the other x_i's in a deterministic way, namely, by setting $x_{i+1}=f(x_i)$ where f is a deterministic function. The problems of this approach are illustrated by the following example. Let D consist of the integers in the interval $[0, n]$, $B(x)=1$ if $x<n/2$ and $B(x)=0$ otherwise, and let $f(x)=x+1$. With such a choice for f, we would essentially output always 0 or always 1, a not too random-looking bit-sequence! This simple example shows that the deterministic function f *may interact badly with the predicate* B spoiling the "randomness" of the output. Theorem 2 essentially shows simultaneous conditions on f and B that prevent such bad interaction. We first describe what predicates B should be used.

2.2. Unapproximable predicates. $N=\{0, 1, 2, \cdots\}$. B is said to be a *set of predicates* if $B=\{B_i: D_i \to \{0, 1\}/i \in S_n, n \in N\}$, where S_n is a subset of the n-bit integers and D_i is a subset of the integers with at most n bits. An element of D_i is always represented by n bits, the leading ones may be 0's.

Set $I_n = \{(i, x) | i \in S_n \text{ and } x \in D_i\}$. An element of I_n is called an input of size n. B is *accessible* if there are two constants c_1 and c_2 and a probabilistic algorithm A such that, on input n, A halts after n^{c_1} steps; A outputs "?" with probability $1/2^{c_2 k}$; and whenever A does not output "?" it outputs an element $(i, x) \in I_n$ with uniform probability.

Let B be a set of predicates and P be a polynomial. Let c_n^P denote the size of a minimum size circuit $C=C[\cdot, \cdot]$ that computes $B_i(x)$ correctly (i.e. $C[i, x]=B_i(x)$) for at least a fraction $\frac{1}{2}+1/P(n)$ of the inputs (i, x) of size n. Such a circuit C will be said to $1/P(n)$*-approximate B*. B is *unapproximable* if for any polynomial P, c_n^P grows faster than any polynomial in n.

Example (Goldwasser and Micali [13]). Let S_n = set of all n-bit composite integers that are products of two distinct equal-length primes. Let D_i denote the set of all integers $x \in [1, i]$ relatively prime to i whose Jacobi symbol $(x/i)=+1$; and let $B_i(x)=1$ if x is a quadratic residue mod i, $B_i(x)=0$ otherwise. Then it is easy to show that B is accessible. Furthermore, under the quadratic residuosity assumption [13], B is also unapproximable. Another example can be found in § 4.

Remark. Note that for an unapproximable predicate B, as n goes to infinity, the fraction of I_n such that $B_i(x)=1$ $(B_i(x)=0)$ goes to $\frac{1}{2}$.

2.3. Sufficient conditions for constructing CSPRB generators.

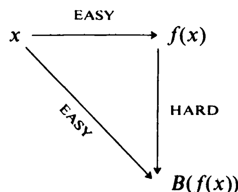


FIG. 1

THEOREM 2. *Let Q be a polynomial, $B=\{B_i: D_i \to \{0, 1\} | i \in S_n, n \in N\}$, an unapproximable and accessible set of predicates and $I=\{(i, x) \in I_n | n \in N\}$ be the set of all*

inputs relative to B. Let

1) $f:(i, x) \in I \to D_i$ *be a polynomial time computable function*;

2) $f_i \equiv f(i, \cdot): D_i \to D_i$ *be a permutation for all* $i \in S_n$,

3) $h:(i, x) \in I \to B_i(f_i(x))$, *be a polynomial time computable predicate* (*see Fig.* 1).

*Then it is possible to construct a Q-*CSPRB *generator G.*

Proof. Let $n \in N$. As B is an accessible set of predicates, let c_1, c_2 and A be its relative constants and probabilistic algorithm. Set $c = Q(n)$, the desired length of the sequence and $n' = \lfloor n^{1/c_1} \rfloor$. The following constitutes the Q-CSPRB generator G that stretches the random n-bit seed r to a $Q(n)$-bit pseudo-random sequence.

> Run A on input n' using the bits of r as coin tosses. If A's output is "?" then generate the sequence consisting of c 0's. Else, if A has randomly selected an input $(i, x) \in I_{n'}$,
> generate the sequence $T_{i,x} = x, f_i(x), f_i^2(x), \cdots, f_i^c(x)$ and
> from right to left (!), extract one bit from each element in $T_{i,x}$ as follows:
> for $j = c$ to 1, output the bit $B_i(f_i^j(x))$.

For simplicity, let us assume that A never outputs "?" and $n = n'$. Then G takes the random input (i, x) and stretches it into the sequence $S_{i,x} = (s_j)_{j=1,\cdots,c}$ where $s_j = B_i(f_i^{c-j+1}(x))$.

G operates in polynomial time. The sequence $T_{i,x}$ can be constructed in Poly (n) time as the function f (and thus each function f_i) is polynomial time computable (hypothesis (1)).

Once the sequence $T_{i,x}$ is computed and stored, each bit $s_j \in S_{i,x}$ can also be computed in polynomial time: by hypothesis (3), $s_j = B_i(f_i^{c-j+1}(x))$ is easy to compute as $f_i^{c-j}(x)$ is given.

G is cryptographically strong. Let P_1 and P_2 be polynomials. We want to prove that, when n is large enough, for any k between 1 and $c-1 = Q(n)$, a circuit C with less than $P_1(n)$ gates, cannot "predict" s_{k+1} with probability greater than $\frac{1}{2} + 1/P_2(n)$. The proof is by contradiction. Assume that there is an infinite family of integers, F, such that for each $n \in F$ there is a circuit C_n, with less than $P_1(n)$ gates, predicting s_{k+1} correctly with probability (taken over all the possible seeds of length n) at least $\frac{1}{2} + 1/P_2(n)$. Then the following poly(n) time algorithm A, making calls to the circuits C_n, $1/P_2(n)$-approximates B; i.e. $A(i, x) = B_i(x)$ for a fraction $\frac{1}{2} + 1/P_2(n)$ of the $(i, x) \in I_n$. This will contradict the assumption that B is unapproximable. In fact, as all C_n have size less than $P_1(n)$, for some polynomial P_3, for each $n \in F$, A can be transformed to a circuit, with less than $P_3(n)$ gates, that $\frac{1}{2} + 1/P_2(n)$ – approximates B for inputs of size n.

ALGORITHM A.

> For input $(i, x) \in I_n$, $n \in F$, generate the sequence of bits $(b_1, \cdots, b_{k-1}, b_k) = (B_i(f_i^k(x)), \cdots, B_i(f_i^2(x)), B_i(f_i(x)))$. Input these k bits to the circuit C_n to compute a bit y. Set $A(i, x) = y$.

We now prove that A $1/P_2(n)$-approximates B for inputs of size n, $n \in F$. Notice that the bits $b_1, \cdots, b_k$ are the first k bits of the Q-CSPRB sequence $S_{i, f_i^{k-c}(x)}$. Thus $A(i, x) = B_i(x)$ if and only if C_n correctly predicts the $k+1$st bit of $S_{i, f_i^{k-c}(x)}$. But this will happen for a fraction at least $\frac{1}{2} + 1/P_2(n)$ of the $(i, x) \in I_n$. In fact we are assuming that C_n correctly predicts the $k+1$st bit of the $S_{i,x}$ sequences for a fraction $\frac{1}{2} + 1/P_2(n)$ of the $(i, x) \in I_n$ and we know that the function f_i^{k-c} is a permutation as, by hypothesis (2), f_i is. Q.E.D.

3. The discrete logarithm problem. Let p be a prime. The set of integers $[1, p-1]$ forms a cyclic group under multiplication mod p. Such a group is denoted by Z_p^*. Let

g be a generator for Z_p^*. The function $f_{p,g}: x \in Z_p^* \to g^x \bmod p$ defines a permutation on Z_p^* computable in Poly ($|p|$) time. The *discrete logarithm problem* (DLP) with inputs p, g and y consists in finding the $x \in Z_p^*$ such that $g^x \bmod p = y$. A circuit $C[\cdot,\cdot,\cdot]$ *solves the* DLP mod *a prime p* if, for any g generator for Z_p^* and any $y \in Z_p^*$, $C[p, g, y] = x$ such that $x \in Z_p^*$ and $g^x \bmod p = y$. Such an x will be simply denoted by $\text{index}_g(y)$ whenever no ambiguity may arise about p.

3.1. Actual knowledge about the DLP. $g^x \bmod p$ seems to be a one-way function. The fastest algorithm known for the DLP is due to Adleman [1] and runs in time $O(2^{c\sqrt{\log p \log\log p}})$. It is easy to see that the difficulty of the DLP does not depend on the generator g or y. By this we mean that if for a nonnegligible fraction (1/Poly ($|p|$)) of pairs (g, y), g a generator and $y \in Z_p^*$, the DLP with inputs p, g and y could be efficiently solved, then it could be solved in probabilistic poly ($|p|$) time for any g and any y. Thus our intractability assumption for the DLP will depend only on the prime p.

Pohlig and Hellman [26] show that the DLP mod a prime p such that $p-1$ contains only small prime factors can be efficiently solved. However such primes constitute a negligible fraction of all primes [34]. No one knows how to construct "small" circuits that solve the DLP mod *even a single* prime p that is not of the above type.

3.2. The intractability assumption for the DLP. Let P be a polynomial and let c_n^P denote the size of a smallest size circuit C that solves the DLP for at least a fraction $1/P(n)$ of the n-bit primes p. Then c_n^P grows faster than any polynomial in n.

Why circuit complexity? The above intractability assumption is certainly a strong one. It implies that the CSPRB sequences, implemented using the DLP as described in § 4, resist prediction by polynomial size circuits.

For the same implementation, if we assume that the DLP cannot be solved in probabilistic polynomial time, we could prove (in essentially the same way!) that the CSPRB sequences would resist prediction by any *fixed* probabilistic polynomial time Turing machine M, i.e., for all sufficiently large seed length k, M cannot predict the next bit in a CSPRB sequence generated with a seed of length k better than at random.

This, however, is not satisfactory for the cryptographic applications mentioned in § 1.5. We would like first to choose a seed length, and then allow our adversary to choose *any* probabilistic polynomial time Turing machine for predicting our sequence! The problem calls for nonuniform complexity.

3.3. The DLP and the principal square root problem. We recall some known results about Z_p^*.

An element T of Z_p^* is called a quadratic residue mod p if and only if $T = x^2 \bmod p$ for some $x \in Z_p^*$; such an x is called a square root mod p of T.

FACT 1. *Given any generator g for* Z_p^*, *an element T of* Z_p^* *is a quadratic residue* mod p *if and only if* $T = g^{2s} \bmod p$ *for some integer* $s \in [1, (p-1)/2]$. *We recall that such a representation of T is unique. Moreover T has two square roots* mod p: $g^2 \bmod p$ *and* $g^{s+(p-1)/2} \bmod p$ (*e.g. see Shanks* [30]).

FACT 2. *There exists a polynomial time algorithm for testing whether an element T of* Z_p^* *is a quadratic residue* mod p (*e.g. see* [30]).

FACT 3 (Adleman, Manders and Miller [2], Berlekamp [5]). *Given any T, a quadratic residue* mod p, *there exists a probabilistic polynomial-time algorithm to compute both square roots of T* mod p.

We introduce the following basic definition.

DEFINITION. Let g be a generator for Z_p^*, T a quadratic residue mod p and $2s$ the unique index of T such that $2s \in [1, p-1]$. Then $g^s \bmod p$ will be called the

g-principal square root of T, and $g^{s+(p-1)/2} \bmod p$ the *g-nonprincipal square root* of T. We will simply say principal square root and nonprincipal square root when no ambiguity about the generator g may arise.

Let g be a generator for Z_p^*. Notice that given T, a quadratic residue mod p, but not the index of T base g, one can still test efficiently that T is indeed a quadratic residue and can efficiently extract its two square roots mod p, say X and Y. However Theorem 3 shows that deciding which square root of T is the g-principal one is a much harder problem. In fact, even allowing a *weak oracle* for the principal square root problem, the DLP becomes easy.

DEFINITION. Let g be a generator for Z_p^* and $x \in Z_p^*$. The predicate $B_{p,g}(x)$ is defined to be equal to 1 if x is the principal square root of $x^2 \bmod p$ and 0 otherwise.

Remark 1. Notice that, given $s \in Z_p^*$ such that $x = g^s \bmod p$, it is easy to evaluate $B_{p,g}(x)$: just check whether or not $s \leqq (p-1)/2$ and output 1 or 0 respectively.

THEOREM 3. *Let Q be a polynomial. Let $MB_Q[\cdot, \cdot, \cdot]$ (magic box) be an oracle such that, for all primes p and for all generators for Z_p^*, $MB_Q[p, g, x] = B_{p,g}(x)$ for a fraction at least $\frac{1}{2} + 1/Q(|p|)$ of the $x \in Z_p^*$. Then there is a probabilistic algorithm with oracle MB_Q that, for all primes p, solves the* DLP mod p *in expected* poly $(|p|)$ *time.*

We first establish some intermediate results. The following lemma shows that with an oracle for the principal square root problem, the DLP is solvable in polynomial time.

LEMMA 1. *Let $MB[\cdot, \cdot, \cdot]$ be an oracle such that, for all primes p, for all generators g for Z_p^* and all $x \in Z_p^*$, $MB[p, g, x] = B_{p,g}(x)$. Then there is a* poly $(|p|)$ *time algorithm with oracle MB that solves the* DLP mod p *for all primes p.*

Proof. We actually prove a stronger result: we exhibit a poly $(|p|)$ time algorithm that finds indices base $g \bmod p$ by only making use of the more restricted oracle $MB[p, g, \cdot]$.

The algorithm, given by $y \in Z_p^*$, finds $x = \text{index}_g(y)$ bit-by-bit from right to left. In the middle of the execution, the variable *index* will contain the right half of the bits of x and the variable *element* is such that $\text{index}_g(element)$ = the left half of x. Think of $\text{index}_g(element)$ and *index* as lists of 0's and 1's. The algorithm, abstractly, transfers the last bit of $\text{index}_g(element)$ in front of *index* until $\text{index}_g(element)$ vanishes (i.e. $element = g^0 = 1$) and thus all of x has been reconstructed in *index*. "$\sim$" denotes the concatenation operator.

Step 0 (Initialization)
element := y; *index* := empty word.
Step 1 (check for termination condition)
If *element* = 1 HALT. *index* = x.
Step 2 (find one more bit of x)
Test whether *element* is a quadratic residue mod p. If yes *index* := $0 \sim$ *index* and go to step 4 else *index* := $1 \sim$ *index* and go to step 3.
Step 3 (*element* is a quadratic nonresidue, i.e. $\text{index}_g(element)$ is odd. Change the last bit of $\text{index}_g(element)$ from 1 to 0)
element := $g^{-1} \cdot$ *element* mod p
Step 4 (erase 0 from the tail of $\text{index}_g(element)$)
element is a quadratic residue. Compute both square roots of *element* mod p. Have MB select the principal one. *element* := principal square root of *element* and go to Step 1.
Q.E.D.

The algorithm in Lemma 1 needs, for $|p|$ times, to select the principal square root of a quadratic residue mod p. It does so by making $|p|$ calls to the oracle MB that computes $B_{p,g}$ correctly 100% of the time.

We should ask what happens to the algorithm if it is allowed to make calls to an oracle that evaluates $B_{p,g}$ only slightly better than guessing at random.

The following lemma, making use of the algebraic structure of Z_p^*, shows how to "concentrate a stochastic advantage", i.e. how to turn an oracle that answers *most of the instances* of a decision problem correctly, *even if we do not know which ones!*, into an oracle answering *any specific instance* correctly with arbitrarily high probability. Let us recall one version of the weak law of large numbers:

> *If* $y_1, \cdots, y_k$ *are k independent* 0–1 *variables such that* $y_i = 1$ *with probability* α, *and* $S_k = y_1 + \cdots + y_k$, *then for real numbers* ε *and* $\delta > 0, k > 1/(4\varepsilon^2\delta)$ *implies that* $\text{Prob}(|S_k/k - \alpha| > \varepsilon) < \delta$.

Let us define trials (ε, δ). trials $(\varepsilon, \delta) = 1/(4\varepsilon^2\delta)$. Notice that trials (ε, δ) depends polynomially on ε^{-1} and δ^{-1}.

Let p be a prime, g a generator for Z_p^* and $t \in [1, p-1]$. Then $IS(p, g, t)$, the *t-initial segment of* Z_p^* with respect to g, is defined by $IS(p, g, t) = \{g^x \bmod p \mid 0 \leqq x \leqq (p-1)/t\}$.

LEMMA 2. *Let* $\varepsilon \in (0, \frac{1}{2})$ *and* $\delta \in (0, 1)$. *Set* $t = \text{trials}(\varepsilon/2, \delta)$. *Let* $MB_\varepsilon[\cdot\,. \cdot, \cdot]$ *be an oracle such that for p prime, g generator for* Z_p^* *and* $x \in Z_p^*$, $MB_\varepsilon[p, g, x] = B_{p,g}(x)$ *for a fraction at least* $\frac{1}{2} + \varepsilon$ *of the* $x \in Z_p^*$. *Then, there is a probabilistic* poly $(|p|, \varepsilon^{-1}, \delta^{-1})$ *algorithm with oracle* MB_ε *that on input p (prime) and e (quadratic residue* mod *p belonging to* $IS(p, g, t)$) *selects the g-principal square root of e correctly with probability greater than* $1 - \delta$.

Proof. Let p prime and g generator for Z_p^*. Again, to find indices base g mod p we will only make use of the more restricted oracle $MB_\varepsilon[p, g, \cdot]$. As in the rest of the proof p and g will remain fixed, we write $MB_\varepsilon[x]$ instead of $MB_\varepsilon[p, g, x]$. On input $e \in IS(p, g, t)$, e quadratic residue mod p, select $r_1, \cdots, r_t$ at random in $[1, (p-1)/2]$. Compute $2r_1, \cdots, 2r_i$. Compute $e_1 = e \cdot g^{2r_1} \bmod p, \cdots, e_n = e \cdot g^{2r_i} \bmod p$. All the e_i's are quadratic residues mod p as $\text{index}_g(e_i)$ is even for all i's. In fact $\text{index}_g(e_i) = (\text{index}_g(e) + 2r_i) \bmod p-1$ and both $\text{index}_g(e)$ and $p-1$ are even. Compute the two square roots X_i and Y_i of each e_i. (Note that while both square roots can be computed, it is not (yet) clear which of X_i and Y_i is principal.) For each e_i select $PSQR_i$, your guess for the principal square root of e_i, in the following way: if $MB_\varepsilon[X_i] = MB_\varepsilon[Y_i]$, randomly select, with probability $\frac{1}{2}$, one of the two square roots X_i and Y_i; call Z_i your selection and set $PSQR_i = Z_i$. Otherwise, if $MB_\varepsilon[X_i] = 1$, set $PSQR_i = X_i$; else set $PSQR_i = Y_i$. Notice that the e_i's have been drawn at random with uniform probability among the quadratic residues mod p: in fact every even index between 1 and $p-1$ can be uniquely written in the form $(\text{index}_g(e) + 2r) \bmod p-1$, for $1 \leqq 2r \leqq p-1$. Thus, even if an *adversary* has chosen the x's for which $MB_\varepsilon(x) = B_g(x)$, setting $\alpha' = \text{Prob}$ ($PSQR_i$ is the principal square root of e_i), we have $\alpha' = \frac{1}{2} + \varepsilon$.

Notice the following fact:

> *Let* $2s$ *be the index of e, i.e.* $e = g^{2s} \bmod p$ *and* $2s \in [1, p-1]$, *and let X and Y be its square roots* mod *p. Let* $2s + 2r < p - 1$. *Then* $X \cdot g^r \bmod p$ *is the principal square root of* $e \cdot g^{2r} \bmod p$ *if and only if X is the principal square root of e.*

$2s$ is unknown, but, as $e \in IS(p, g, t)$, we know that $2s \in [1, (p-1)/t]$. Therefore, if $2s + 2r_i > p-1$, $2r_i$ must belong to the interval $[(t-1)(p-1)/t, p-1]$. This will happen with probability $= 1/t$. Assume, without loss of generality, that $PSQR_i = X_i$ and $X_i \cdot g^{r_i} \bmod p = X$. Then,

$$\alpha = \text{Prob}(B_{p,g}(X) = 1 \mid PSQR_i = X_i) \geqq \alpha' - 1/t > \tfrac{1}{2} + \varepsilon/2.$$

(Recall that $t=1/\varepsilon^2\delta$.) We exploit this fact in the following way: initialize to "0" two counters C_X and C_Y. For each r_i if $X\cdot g^{r_i} \bmod p = PSQR_i$ then increment C_X, else increment C_Y. Upon termination, if $C_X > C_Y$ output X as the principal square root of e, or else Y. As $\alpha > \frac{1}{2}+\varepsilon/2$ let X be the principal square root of e, then by the weak law of large numbers, Prob $(|C_X/t-\alpha|>\varepsilon/2)<\delta$. Or equivalently, $C_X > C_Y$ with Probability $>1-\delta$. Q.E.D.

LEMMA 3. *Let Q be a polynomial and*

$$t=\text{trials}\left(\frac{1}{Q(|p|)},\frac{1}{2|p|}\right).$$

Let MB_Q be an oracle such that, for all primes p and all generators g for Z_p^*, $MB_Q[p,g,x]=B_{p,g}(x)$ for at least a fraction $\frac{1}{2}+1/Q(|p|)$ of the $x\in Z_p^*$. Then there is a probabilistic poly $(|p|)$ *algorithm that on input p prime and $y\in IS(p,g,t)$ finds* $\text{index}_g(y) \bmod p$ *in expected* poly $(|p|)$ *time.*

Proof. On inputs p, g, and y we will only call the more restricted oracle $MB_\varepsilon[p,g,\cdot]$. Let y be any element in $IS(p,g,t)$. We apply a modification of the algorithm in Lemma 1 to find the index of y. That algorithm, in Step 4, to select the principal square root of a quadratic residue mod p, and thus also for a quadratic residue in $IS(p,g,t)$, would call the oracle MB. Call instead MB_Q as in the algorithm of Lemma 2 setting $\varepsilon=1/Q(|p|)$ and $\delta=1/2|p|$. By Lemma 2, Step 4 will be performed correctly with *independent probability* equal to $1-1/2|p|$. Notice that if x belongs to $IS(p,g,n)$, so does $x\cdot g^{-1} \bmod p$; and that if x is a quadratic residue mod p belonging to $IS(p,g,t)$, also its principal square root will belong to $IS(p,g,t)$. Therefore, if in Step 4 the algorithm correctly selects the principal square root, the total computation will be done in the initial segment $IS(p,g,t)$. As Step 4 is executed at most $|p|$ times, the probability that the index of y will be found correctly is greater than $(1-1/(2|p|))^{|p|}>\frac{1}{2}$ (consider the Taylor series expansion around $x=0$ of the function $f(x)=(1-1/x)^{|p|}$). It is easy to see that the whole computation is polynomial in $|p|$. Q.E.D.

We are now ready to prove Theorem 3.

Proof of Theorem 3. The following probabilistic poly $(|p|)$ time algorithm finds $\text{index}_g(y) \bmod p$ for *any* $y\in Z_p^*$ by only making calls to the oracle $MB_Q[p,g,\cdot]$. Set

$$t=\text{trials}\left(\frac{1}{Q(|p|)},\frac{1}{2|p|}\right).$$

Recall that

$$IS(p,g,t)=\left\{g^x \bmod p \,\middle|\, x\in\left[0,\frac{p-1}{t}\right]\right\}.$$

The algorithm makes use of the variables *i*, *w*, *index*(w), and *candidate*.

Step 0 (Initialization)

$i:=1$

Step 1 (guess that $\text{index}(y)\in[i(p-1)/t,\ (i+1)(p-1)/t]$ and map y into the t-initial segment)

$w:=y\cdot g^{-i(p-1)/t} \bmod p$

Step 2 (If $w \in IS(p, g, t)$, find the index of w)

Apply the algorithm in Lemma 3 to find the index of w. $index(w) := $ the index of w.

Step 3 (check whether the index of y has been found)

$candidate := index(w) + i(p-1)/t$; if $g^{candidate} \bmod p = y$ then HALT: *candidate* is the index of y in base g. Else continue.

Step 4 (keep on guessing)

$i := i+1$. If $i > t$ then $i := 1$ and go to Step 0; else go to Step 1. Q.E.D.

4. A concrete CSPRB generator based on the discrete logarithm problem. Let us describe an implementation, based on the intractability assumption for the DLP, of our general algorithmic scheme for constructing CSPRB generators.

We first recall a recent and powerful result due to Erich Bach [3].

LEMMA 4 (Bach [3]). *There is a probabilistic algorithm that, on input $n \in N$, selects an integer k, together with its prime factorization, with uniform probability among the n-bits integers. The algorithm runs in expected* poly (n) *time.*

THEOREM 4. *Under the intractability assumption for the* DLP, *we can construct a* CSPRB *generator.*

Proof. Let S_{2n} be the set of the $2n$-bit integers i (leading bit $=1$) such that the first n bits of i constitute a prime p, and the next n bits (leading bit possibly 0) a generator g for Z_p^*. Let "$\sim$" denote concatenation. For $i \in S_{2n}$, $i = p \sim g$, set $D_i = Z_p^*$ and, for $x \in Z_p^*$, set $B_i(x) = B_{p,g}(x)$. We show that the set of predicates $B = \{B_i | i \in S_{2n}\}$ is an accessible, unapproximable set of predicates.

B is accessible. a) With uniform probability, we can select, among all the n-bits primes, a prime p together with the factorization of $p-1$, in probabilistic poly (n) time.

Select, with uniform probability, an n-bit integer k, together with its prime factorization, until $k+1$ is a prime. By Lemma 4, k can be selected in expected poly (n) time. $k+1$ can be tested for primality in random poly(n) time (see Solovay and Strassen [32]) and it will be a prime after expected $O(n)$ random selections of k because of the prime number theorem. If the prime $p = k+1$ has been so selected, it has been selected with uniform probability.

b) To generate a triplet (p, g, x) such that p is an n-bit prime, g a generator for Z_p^* and $x \in Z_p^*$, with uniform probability we follow the following algorithm:

(1): Generate p as in (a).
(2): Flip $2n$ fair coins; if the first n outcomes of the flips constitute a generator for Z_p^* and the second n outcomes constitute an $x \in Z_p^*$ then halt, the desired triplet has been selected, else go to (1).

As all triplets (p, g, x) so generated have the same probability of being selected it remains to show that the above algorithm runs fast. For this, note that, for all n-bit primes p, the probability of generating an $x \in Z_p^*$ is greater than $\frac{1}{2}$. Also, for all the n-bit primes p, the probability of constructing a generator for Z_p^* by flipping n fair coins is greater than $1/(12 \log_e \log_e (p-1))$. In fact, for all p, the generators for Z_p^* are $\varphi(p-1)$ (where φ is Euler totient function) and Rosser and Schoenfield [28] prove that $\varphi(k) > 1/(6 \log_e \log_e k)$ for $k > 3$. Moreover, as we have the factorization of $p-1$ as well, it is easy to check whether g is a generator for Z_p^* (see [30]).

B is unapproximable. By contradiction. Assume that there are polynomials P_1 and P_2 such that for $n \in F$, F infinite, there is a $P_1(n)$-size circuit C_n that evaluates

$B_{p,g}(x)$ correctly for a fraction of at least $\frac{1}{2}+1/P_2(n)$ of the n-bit inputs p, g, and x. Then a counting argument shows that there would be a fraction at least $1/P_2(n)$ of pairs (p, g) for which the circuit C_n evaluates $B_{p,g}(x)$ correctly for at least a fraction $\frac{1}{2}+1/(2 \cdot P_2(n))$ of the $x \in Z_p^*$. A trivial modification of Theorem 3 would then show that there is a probabilistic poly (n) time algorithm A, with oracle C_n, that for each $n \in F$ solves the DLP for at least a fraction $1/P_2(n)$ of the n-bit primes p. This would violate the intractability assumption for the DLP as, for some polynomial P_3, for each $n \in F$, A could be transformed to a circuit with less than $P_3(n)$ gates.

B satisfies the hypothesis of Theorem 2. A seed is a pair $(i = p \sim g, x)$. Define $f_i(x) = g^x \bmod p$. Note that, given $x \in Z_p^*$, it is easy to check whether $g^x \bmod p$ is a principal square root: just check whether $x \leqq (p-1)/2$. The other properties trivially hold. Q.E.D.

Theorems 2, 3 and 4 imply that it is possible to stretch a short random seed into a long pseudo-random bit sequence such that any efficient strategy to predict better than 50–50 the next output bit can be easily transformed to a "small" circuit solving the discrete logarithm problem.

5. Random self-reducibility. The purpose of this section is to single out the notion of random self-reducibility that we hope will be useful to complexity theory.

The notion of reducibility (Cook [11], Karp [16] and Levin [19]) is central to complexity theory. Conjunctive self-reducibility has also played an important role (see Berman [6], Fortune [12], Meyer and Paterson [22] and the article of Mahaney [21] proving the conjecture of Berman and Hartmanis that no NP-complete set can be reduced to a sparse set unless P = NP).

We introduce the notion of random self-reducibility by distilling the properties of the reductions in § 3. Informally, these properties guarantee that, if the majority of the instances of a decision problem (even if we do not know which ones) can be efficiently answered correctly, then every individual instance can be efficiently answered correctly with arbitrarily high probability.

In the two definitions below, $B = \{B_i: D_i \to \{0, 1\} | i \in S_n, \text{ and } n \in N\}$ is an *accessible* set of predicates. ρ is the "reduction function" and σ the "interpretation function": using r, a sequence of coin tosses, ρ randomly maps instance x into instance y; σ, given the answer for y and the sequence of coin tosses r, tells us what the answer for x should be.

DEFINITION (*strong random self-reducibility*). Let $\rho: (i \in S_n, x \in D_i, r \in D_i) \to D_i$ and $\sigma: (i \in S_N, x \in D_i, r \in D_i, b \in \{0, 1\}) \to \{0, 1\}$ be polynomial time computable functions. We say that B is *strongly randomly self-reducible* if for all $i \in S_N$ and all $x \in D_i$:

a) $\rho(i, x, \cdot)$ is a permutation over D_i and

b) for all $r \in D_i$, $B_i(x) = \sigma(i, x, r, B(\rho(i, x, r)))$.

DEFINITION (*weak random self-reducibility*). Let $\rho: (i \in S_n, x \in D_i, r \in D_i) \to D_i$ be, as before, a polynomially computable function and $\sigma: (i \in S_N, x \in D_i, r \in D_i, \varepsilon \in (0, 1), b \in \{0, 1\}) \to \{0, 1\}$ be a function computable in probabilistic poly $(|i|, \varepsilon^{-1})$ time. We say that B is *weakly randomly self-reducible* if for any polynomial Q, for all sufficiently large $i \in S_N$ and all $x \in D_i$:

a) $\rho(i, x, \cdot)$ is a permutation over D_i and

b) letting r be randomly selected in D_i,

$$\text{Prob}\,(B_i(x) = \sigma(i, x, r, B_i(\rho(i, x, r))) > \tfrac{1}{2} + 1/Q(|i|).$$

Acknowledgments. We are proud to thank many friends.

We are grateful to Shafi Goldwasser for having suggested the discrete logarithm

as a suitable one-way function for our purposes and for numerous valuable discussions, to Richard Karp for his precious gift of setting the context and making vague ideas precise, and to Andy Yao for having brought to light hidden potentials.

This work gained a great deal of concision and clarity due to the elegant result of Erich Bach [3].

We benefitted highly from the insightful comments of Lenore Blum, Steve Cook, Faith Fich, Zvi Galil, Donald Johnson, Leonid Levin, David Lichtenstein, Mike Luby, Gary Miller, Andrew Odlizko, Joan Plumstead, Charlie Rackoff, Ron Rivest, Jeff Shallit, Mike Sipser and Po Tong.

REFERENCES

[1] L. ADLEMAN, *A subexponential algorithm for the discrete logarithm problem with applications to cryptography*, Proc. 20th IEEE Symposium on Foundations of Computer Science, 1979, pp. 55–60.

[2] L. ADLEMAN, K. MANDERS AND G. MILLER, *On taking roots in finite fields*, Proc. 18th IEEE Symposium on Foundations of Computer Science, 1977, pp. 175–177.

[3] E. BACH, *How to generate random integers with known factorization*, Proc. 15th ACM Symposium on Theory of Computing, 1983.

[4] H. BEKER AND F. PIPER, *Cipher Systems*, Northwood, 1982.

[5] E. BERLEKAMP, *Factoring polynomials over large finite fields*, Math. Comp., 24 (1970), pp. 713–735.

[6] P. BERMAN, *Relationship between density and deterministic complexity of* NP*-complete languages*, 5th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science, 62, Springer-Verlag, New York, 1978, pp. 63–71.

[7] M. BLUM, *How to exchange (secret) keys*, Proc. 15th ACM Symposium on Theory of Computing, 1983.

[8] ———, *Three applications of the oblivious transfer*, unpublished manuscript, 1981.

[9] L. BLUM, M. BLUM AND M. SHUB, *A simple secure pseudo-random number generator*, Proc. CRYPTO-82, Allen Gersho, ed.; this Journal, to appear.

[10] M. BLUM AND S. MICALI, *How to generate cryptographically strong sequences of pseudo-random bits*, Proc. 23rd IEEE Symposium on Foundations of Computer Science, 1982, pp. 112–117.

[11] S. COOK, *The complexity of theorem proving procedures*, Proc. 3rd ACM Symposium on Theory of Computing, 1971, pp. 151–158.

[12] S. FORTUNE, *A note on sparse complete sets*, this Journal, 8 (1979), pp. 431–433.

[13] S. GOLDWASSER AND S. MICALI, *Probabilistic encryption and how to play mental poker keeping secret all partial information*, Proc. 14th ACM Symposium on Theory of Computing, 1982, pp. 365–377, Probabilistic Encryption, J. Comp. Sys. Sci., to appear.

[14] S. GOLDWASSER, S. MICALI AND P. TONG, *Why and how to establish a private code on a public network*, Proc. 23rd IEEE Symposium on Foundations of Computer Science, 1982, pt. 134–144.

[15] S. GOLDWASSER, S. MICALI AND A. YAO, *Strong signature schemes and authentication*, Proc. 15th ACM Symposium on Theory of Computing, 1983.

[16] R. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. Miller and J. Thatcher, eds., Plenum, New York, 1972, pp. 85–103.

[17] D. KNUTH, *The Art of Computer Programming*: *Vol.* 2 *Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 1981.

[18] A. KOLMOGOROV, *Three approaches to the concept of "the amount of information"*, Probl. of Inf. Transm., 1/1, 1965.

[19] L. A. LEVIN, *Universal sequential search problems*, Probl. of Inf. Transm., 9/3 (1973), pp. 265–266.

[20] M. LUBY, S. MICALI AND C. RACKOFF, *The MiRackoLus exchange of a secret bit*, Proc. 24th IEEE Symposium on Foundations of Computer Science, 1983.

[21] S. MAHANEY, *Sparse complete sets for* NP: *a solution of a conjecture of Berman and Hartmanis*, Proc. 20th IEEE Symposium on Foundations of Computer Science, 1980, pp. 54–59.

[22] A. MEYER AND M. PATERSON, *With what frequency are apparently intractable problems difficult?*, Massachusetts Institute of Technology, Tech. Report, Cambridge, MA, Feb. 1979.

[23] G. MILLER, *Riemann's hypothesis and tests for primality*, J. Comp. Sys. Sci., 13 (1976), pp. 300–317.

[24] G. PETERSON, *Succint representations, random strings and complexity classes*, Proc. 21st IEEE Symposium on Foundations of Computer Science, 1980, pp. 86–95.

[25] J. PLUMSTEAD, *Inferring a sequence generated by a linear congruence*, Proc. 23rd IEEE Symposium on Foundations of Computer Science, 1982, pp. 153–159.

[26] S. POHLIG AND M. HELLMAN, *An improved algorithm for computing logarithms over GF(p) and its cryptographic significance*, IEEE Trans. Information Theory, IT-24 (1978), pp. 106–110.

[27] R. RIVEST, A. SHAMIR AND L. ADLEMAN, *On digital signatures and public key cryptosystems*, Comm. ACM, 21 (1978), pp. 120–126.

[28] J. ROSSER AND L. SCHOENFIELD, *Approximate formulas for some functions of prime numbers*, Illinois J. Math., 6 (1962), pp. 64–94.

[29] A. SHAMIR, *On the generation of cryptographically strong pseudo-random sequences*, 8th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science, 62, Springer-Verlag, New York, 1981.

[30] D. SHANKS, *Solved and Unsolved Problems in Number Theory*, Chelsea, London, 1978.

[31] M. SIPSER, *Three approaches to a definition of finite state randomness*, unpublished manuscript, 1979.

[32] R. SOLOVAY AND V. STRASSEN, *A fast Monte-Carlo test for primality*, this Journal, 6 (1977), pp. 84–85.

[33] A. YAO, *Theory and applications of trapdoor functions*, Proc. 23rd IEEE Symposium on Foundations of Computer Science, 1982.

[34] ANDREW ODLIZKO, private communication, 1984.

SIAM J. COMPUT.
Vol. 13, No. 4, November 1984

© 1984 Society for Industrial and Applied Mathematics
014

CYLINDRICAL ALGEBRAIC DECOMPOSITION I: THE BASIC ALGORITHM*

DENNIS S. ARNON †, GEORGE E. COLLINS ‡, AND SCOTT McCALLUM §

Abstract. Given a set of r-variate integral polynomials, a *cylindrical algebraic decomposition (cad)* of euclidean r-space E^r partitions E^r into connected subsets compatible with the zeros of the polynomials. Each subset is a *cell*. Collins gave a cad construction algorithm in 1975, as part of a quantifier elimination procedure for real closed fields. The cad algorithm has found diverse applications (optimization, curve display); new applications have been proposed (term rewriting systems, motion planning). In the present two-part paper, we give an algorithm which determines the pairs of adjacent cells as it constructs a cad of E^2. Such information is often useful in applications. In Part I we describe the essential features of the r-space cad algorithm, to provide a framework for the adjacency algorithm in Part II.

Key words. polynomial zeros, computer algebra, computational geometry, semi-algebraic geometry, real closed fields, decision procedures, real algebraic geometry

1. Introduction. Given a set of r-variate integral polynomials, a *cylindrical algebraic decomposition (cad)* of euclidean r-space E^r partitions E^r into connected subsets compatible with the zeros of the polynomials. By "compatible with the zeros of the polynomials" we mean that on each subset of E^r, each of the polynomials either vanishes everywhere or nowhere. For example, consider the bivariate polynomial

$$y^4 - 2y^3 + y^2 - 3x^2y + 2x^4.$$

Its zeros comprise the curve shown in Figure 1.

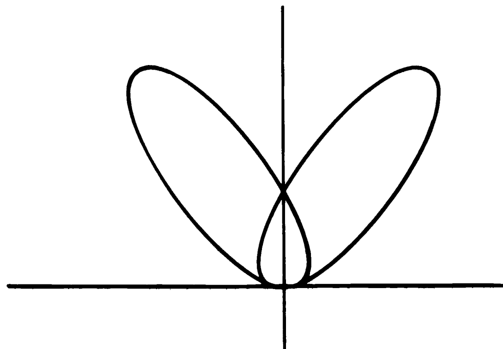


Fig. 1

*Received by the editors January 13, 1983, and in revised form January 2, 1984. This work was partially supported by the National Science Foundation, Grant MCS-8009357 to the University of Wisconsin-Madison, and by the Purdue Research Foundation. This paper was typeset at Xerox PARC using TeX in the Cedar environment. The final copy was produced on September 19, 1984.

† Xerox PARC, 3333 Coyote Hill Road, Palo Alto, California 94304. Formerly with Computer Science Department, Purdue University, West Lafayette, Indiana.

‡ Computer Science Department, University of Wisconsin-Madison, Madison, Wisconsin, 53706.

§ Department of Computer Science, University of Toronto, Toronto, Canada M5S 1A7. Formerly with Computer Science Department, University of Wisconsin-Madison, Madison, Wisconsin.

Figure 2 shows a cad of the plane compatible with its zeros. The cad consists of the distinct "dots", "arcs", and "patches of white space" of the Figure (a rigorous definition of cad is given in Section 2).

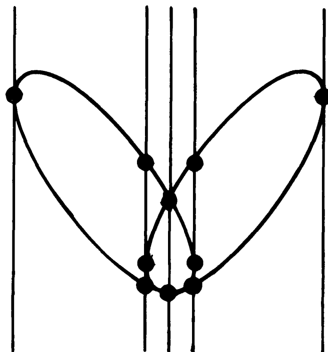


Fig. 2

Cad's were introduced by Collins in 1973 (see [COL75], [COL76]) as part of a new quantifier elimination, and hence decision, method for elementary algebra and geometry. He gave an algorithm for cad construction, and proved that for any fixed number of variables, its computing time is a polynomial function of the remaining parameters of input size. As can be seen in the example above, cad's are closely related to the classical simplicial and CW-complexes of algebraic topology. In fact, the essential strategy of Collins' cad algorithm, induction on dimension, can be found in van der Waerden's 1929 argument ([WAE29], pp. 360-361) that real algebraic varieties are triangulable.

Collins' cad-based decision procedure for elementary algebra and geometry is the best known (see [FER79]; very little besides a cad is needed for the decision procedure). J. Schwartz and M. Sharir used the cad algorithm to solve a motion planning problem ([SCH83a], [SCH83b]). D. Lankford [LAN78] and N. Dershowitz [DER79] pointed out that a decision procedure for elementary algebra and geometry could be used to test the termination of term-rewriting systems. P. Kahn used cad's to solve a problem on rigid frameworks in algebraic topology ([KAH79]). Kahn also observed ([KAH78]) that a cad algorithm provides a basis for a constructive proof that real algebraic varieties are triangulable, and thus for computing the homology groups of a real algebraic variety.

Implementation of Collins' cad algorithm began soon after its introduction, culminating in the first complete program in 1981 [ARN81]. The program has begun to find use; in May, 1982 the termination of the term-rewriting system for group theory in the Appendix of [HUE80] was verified using it. It has also been utilized for display of algebraic curves [ARN83]. In 1977, Müller implemented certain subalgorithms of the cad algorithm and used them to solve algebraic optimization problems [MUE77].

We use a somewhat different (but equivalent) definition of cad than that in [COL75]; we devote Section 2 to it. We then take up the cad algorithm. Its intuitive strategy can be described by means of an example. Consider the curve of Figures 1 and 2. Given the bivariate polynomial which defines it, we will compute univariate polynomials whose roots constitute a "silhouette" of the curve. By this we mean that the roots of the univariate polynomials are the projections, onto the x-axis (E^1), of the "significant points" of the curve. The curve's "significant points"

are its singularities (e.g., self-crossings, cusps, isolated points), and the points at which its tangent is vertical. Suppose that E^1 is decomposed into the points of the silhouette, and their complementary open intervals (this is done by finding the roots of the univariate polynomials). Then the portion of the curve "over" each of these points (intervals) consists of finitely many disjoint "dots" ("arcs"). Our cad of the plane is made by decomposing the line (strip) in the plane "over" each point (interval) in E^1 into the "dots" ("arcs") of the curve, and the "arcs" ("patches") of the complement of the curve, that it contains.

For our sample curve, we compute a single univariate polynomial (its discriminant):

$$2048x^{12} - 4608x^{10} + 37x^8 + 12x^6.$$

This polynomial has five roots, whose approximate values are -1.49, -0.23, 0.0, 0.23, and 1.49. All roots but the third are projections of points with vertical tangent. The third is the projection of the two singularities (self-crossings). Using the roots, we decompose the real line into points and open intervals (Figure 3).

Fig. 3

The Cartesian products of each of the eleven elements of this decomposition with a line, give us eleven vertical lines and strips. As we see in Figure 2, each "significant point" of the curve lies on one of the vertical lines, and within each strip, the curve has finitely many disjoint "arcs". The "dots" and "arcs" which make up each line, and the "arcs" and "patches of white space" which make up each strip, give us the cad of Figure 2.

The general algorithm consists of three phases: projection (computing successive sets of polynomials in $r-1$, $r-2, \ldots, 1$ variables; the zeros of each set contain a "silhouette" of the "significant points" of the zeros in the next higher dimensional space), base (constructing a decomposition of E^1), and extension (successive extension of the decomposition of E^1 to a decomposition of E^2, E^2 to $E^3, \ldots, E^{r-1}$ to E^r). In Sections 3, 4, and 5 we describe each of these phases in turn. In the interests of succintness, we will at various times specify simple but inefficient methods of performing computations (for example, isolating the roots of a product of polynomials, rather than isolating the roots of each of the factors separately). In Section 6, we give a detailed example of the algorithm.

2. Definition of cylindrical algebraic decomposition. Connectivity plays an important role in the theory of cad's. It is convenient to have a term for a nonempty connected subset of E^r; we will call such sets *regions*. For a region R, the *cylinder over* R, written $Z(R)$, is $R \times E^1$. A *section* of $Z(R)$ is a set s of points $< a, f(a) >$, where a ranges over R, and f is a continuous, real-valued function on R. s, in other words, is the graph of f. We say such an s is the *f-section* of $Z(R)$. A *sector* of $Z(R)$ is a set $\hat{s}$ of all points $< \alpha, b >$, where α ranges over R and $f_1(\alpha) < b < f_2(\alpha)$ for (continuous, real-valued) functions $f_1 < f_2$. The constant functions $f_1 = -\infty$, and $f_2 = +\infty$, are allowed. Such an $\hat{s}$ is the *(f_1, f_2)-sector* of $Z(R)$. Clearly sections and sectors of cylinders are regions. Note that if $r = 0$ and $R = E^0 =$ a point, then $Z(R) = E^1$, any point of E^1 is a section of $Z(R)$, and any open interval in E^1 is a

sector of $Z(R)$.

For any subset X of E^r, a *decomposition* of X is a finite collection of disjoint regions whose union is X. Continuous, real-valued functions $f_1 < f_2 < \cdots < f_k, k \geq 0$, defined on R, naturally determine a decomposition of $Z(R)$ consisting of the following regions: (1) the (f_i, f_{i+1})-*sectors* of $Z(R)$ for $0 \leq i \leq k$, where $f_0 = -\infty$ and $f_{k+1} = +\infty$, and (2) the f_i-*sections* of $Z(R)$ for $1 \leq i \leq k$. We call such a decomposition a *stack over* R (determined by $f_1, \ldots, f_k$).

A decomposition D of E^r is *cylindrical* if either (1) $r = 1$ and D is a stack over E^0, or (2) $r > 1$, and there is a cylindrical decomposition D' of E^{r-1} such that for each region R of D', some subset of D is a stack over R. It is clear that D' is unique for D, and thus associated with any cylindrical decomposition D of E^r are unique *induced* cylindrical decompositions of E^i for $i = r-1, r-2, \ldots, 1$. Conversely, given a cad $\hat{D}$ of E^i, $i < r$, a cad D of E^r is an *extension* of $\hat{D}$ if D induces $\hat{D}$.

For $0 \leq i \leq r$, an *i-cell* in E^r is a subset of E^r which is homeomorphic to E^i. It is not difficult to see that if c is an i-cell, then any section of $Z(c)$ is an i-cell, and any sector of $Z(c)$ is an $(i+1)$-cell (these observations are due to P. Kahn [KAH78]). It follows by induction that every element of a cylindrical decomposition is an i-cell for some i. Also, if c is an i-cell, we say that $Z(c)$ is an $(i+1)$-*cylinder*, and that any stack over c is an $(i+1)$-*stack*.

The decomposition of E^2 in Figure 2 is cylindrical. Figure 3 shows the induced decomposition of E^1, consisting of five 0-cells and six 1-cells. The decomposition in Figure 2 consists of eleven stacks. The first, or leftmost, stack consists of a single 2-dimensional sector; the next stack consists of two 1-dimensional sectors and one 0-dimensional section; and so forth.

A subset of E^r is *semi-algebraic* if it can be constructed by finitely many applications of the union, intersection, and complementation operations, starting from sets of the form

$$\{x \in E^r \mid F(x) \geq 0\},$$

where F is an element of $\mathbf{Z}[x_1, \ldots, x_r]$, the ring of integral polynomials in r variables. We write I_r to denote $\mathbf{Z}[x_1, \ldots, x_r]$. As we shall now see, a different (but equivalent) definition of semi-algebraic sets is possible, from which one obtains a useful characterization of them. By a *formula* we will mean a well-formed formula of the first order theory of real closed fields. (The "first order theory of real closed fields" is a precise name for what we referred to above as "elementary algebra and geometry"; see [KRE67]). The formulas of the theory of real closed fields involve elements of I_r. A *definable set* in E^k is a set X such that for some formula $\Psi(x_1, \ldots, x_k)$, X is the set of points in E^k satisfying Ψ. Ψ is a *defining formula* for X. (We follow the convention that $\Psi(x_1, \ldots, x_k)$ denotes a formula Ψ in which all occurrences of $x_1, \ldots, x_k$ are free, each x_i may or may not occur in Ψ, and no variables besides $x_1, \ldots, x_k$ occur free in Ψ.) A definable set is *semi-algebraic* if it has a defining formula which is quantifier-free. The existence of a quantifier elimination method for real closed fields was established by Tarski [TAR48]. Hence a subset of E^r is semi-algebraic if and only if it is definable.

A decomposition is *algebraic* if each of its regions is a semi-algebraic set. A *cylindrical algebraic decomposition* of E^r is a decomposition which is both cylindrical and algebraic.

Let X be a subset of E^r, and let F be an element of I_r. F is *invariant* on X (and X is F-*invariant*), if one of the following three conditions holds:

(1) $F(\alpha) > 0$ for all α in X. ("F has positive sign on X").

(2) $F(\alpha) = 0$ for all α in X. ("F has zero sign on X").

(3) $F(\alpha) < 0$ for all α in X. ("F has negative sign on X").

Let $A = \{A_1, \ldots, A_n\}$, be a subset of I_r ("subset of I_r" will always mean "finite subset"). X is *A-invariant* if each A_i is invariant on X. A collection of subsets of E^r is A-invariant if each element of the collection is.

The decomposition in Figure 2 is an A-invariant cad of E^2 for $A = \{y^4 - 2y^3 + y^2 - 3x^2y + 2x^4\}$. Note that a set $A \subset I_r$ does not uniquely determine an A-invariant cad D of E^r. Since any subset of an A-invariant region is also A-invariant, we can subdivide one or more regions of D to obtain another, "finer", A-invariant cad.

3. The cylindrical algebraic decomposition algorithm: projection phase. Let us begin with a more precise version of the cad algorithm outline at the end of Section 1. Let $A \subset I_r$ denote the set of input polynomials, and suppose $r \geq 2$. The algorithm begins by computing a set $PROJ(A) \subset I_{r-1}$ ("$PROJ$" stands for "projection"), such that for any $PROJ(A)$-invariant cad D' of E^{r-1}, there is an A-invariant cad D of E^r which induces D'. Then the algorithm calls itself recursively on $PROJ(A)$ to get such a D'. Finally D' is extended to D. If $r = 1$, an A-invariant cad of E^1 is constructed directly.

Thus for $r \geq 2$, if we trace the algorithm, we see it compute $PROJ(A)$, then $PROJ(PROJ(A)) = PROJ^2(A)$, and so on, until $PROJ^{r-1}(A)$ has been computed. This is the projection phase. The construction of a $PROJ^{r-1}(A)$-invariant cad of E^1 is the base phase. The successive extensions of the cad of E^1 to a cad of E^2, the cad of E^2 to a cad of E^3, and so on, until an A-invariant cad of E^r is obtained, are the extension phase. For the example of Section 1, where $A = \{y^4 - 2y^3 + y^2 - 3x^2y + 2x^4\}$, $PROJ(A) = \{2048x^{12} - 4608x^{10} + 37x^8 + 12x^6\}$.

The key to the projection phase is to define the map $PROJ$ (which takes a subset of I_r to a subset of I_{r-1}), and to prove that it has the desired property. We stated this property above as: any $PROJ(A)$-invariant cad of E^{r-1} is induced by some A-invariant cad of E^r. To establish this, clearly it suffices to show that over any semi-algebraic, $PROJ(A)$-invariant region in E^{r-1}, there exists an A-invariant algebraic stack. In this section, we define $PROJ$ and outline the proof that it has this latter property.

Central to our definition of $PROJ$ will be the notion of delineability. For $F \in I_r, r \geq 1$, let $V(F)$ denote the real variety of F, i.e., the zero set of F. Let R be a region in E^{r-1}. F is *delineable* on R if the portion of $V(F)$ lying in $Z(R)$ consists of k disjoint sections of $Z(R)$, for some $k \geq 0$. Clearly when F is delineable on R, it gives rise to a stack over R, namely the stack determined by the continuous functions whose graphs make up $V(F) \cap Z(R)$. We write $S(F, R)$ to denote this stack, and speak of the *F-sections* of $Z(R)$. One easily sees that $S(F, R)$ is F-invariant.

For example, consider again $F(x, y) = y^4 - 2y^3 + y^2 - 3x^2y + 2x^4$. F is delineable on each of the eleven cells shown in Figure 3, and in fact the stacks which comprise the cad of Figure 2 are just the stacks determined by F over these eleven cells.

Our tentative strategy for defining $PROJ$ is: insure that for any $PROJ(A)$-invariant region R, the following two conditions hold: (1) each $A_i \in A$ is delineable on R, and (2) the sections of $Z(R)$ belonging to different A_i and A_j are either disjoint or identical. If these conditions are met, then clearly we have an A-invariant stack over R, namely the stack determined by the functions whose graphs are the sections of the A_i's.

The lefthand drawing in Figure 4 illustrates a region R and hypothetical bivariate polynomials A_1, A_2, and A_3 for which these conditions do not hold. A_1 and A_2 are delineable on R, but the A_1-section meets the A_2-section. A_3 is not delineable on R. The righthand drawing in the Figure illustrates a partition of R into five regions, on each of which the conditions are satisfied.

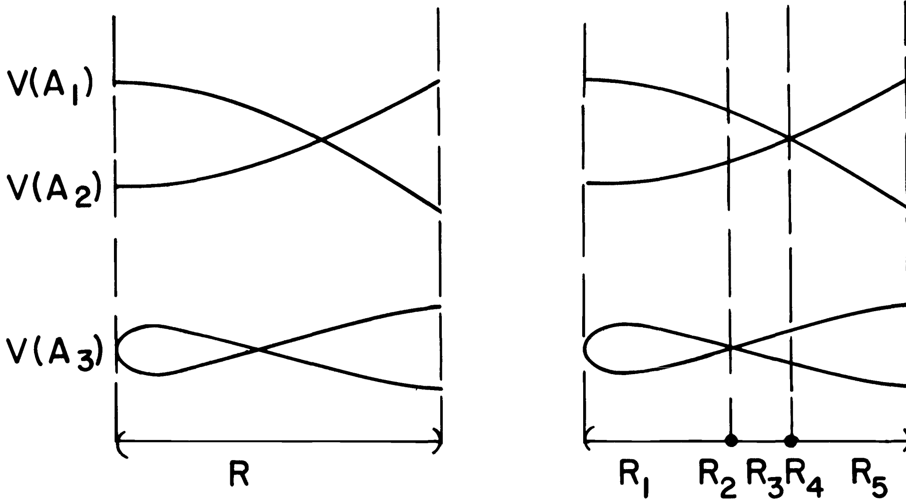


Fig. 4

The following example points out a difficulty with our tentative strategy. Let $A \subset I_2$ be the set $\{A_1(x,y), A_2(x,y)\} = \{x, y^2 + x^2 - 1\}$. $A_1(0,y)$ is the zero polynomial, hence A_1 vanishes everywhere on $Z(\{0\})$, hence A_1 is not delineable on the set $\{0\} \subset E^1$, nor on any superset of it. We resolve this difficulty as follows. We say $F \in I_r$ is *identically zero* on $X \subset E^{r-1}$ if $F(\alpha, x_r)$ is the zero polynomial for every $\alpha \in X$. If F is identically zero on X, then any decomposition of $Z(X)$ will be F-invariant. Hence we may simply ignore F in decomposing $Z(X)$. In particular, in our example, we need only take account of the sections of A_2 in decomposing $Z(\{0\})$. Thus, we modify condition (1) above to read "(1′) each $A_i \in A$ is either delineable or identically zero on R".

$PROJ(A)$ will consist of two kinds of elements: those designed to attend to condition (1′), and those to attend to condition (2). Elements of both kinds are formed from the coefficients of the polynomials of A by addition, subtraction, and multiplication (remark: I_r consists of polynomials in x_r whose coefficients are elements of I_{r-1}). We now specify how this is done.

Let J be any unique factorization domain, and let F and G be nonzero elements of $J[x]$. We write $deg(F)$ to denote the degree of F (the zero polynomial has degree $-\infty$). Let $n = min(deg(F), deg(G))$. For $0 \leq j < n$, let $S_j(F,G)$ denote the jth subresultant of F and G. $S_j(F,G)$ is an element of $J[x]$ of degree $\leq j$. (Each coefficient of $S_j(F,G)$ is the determinant of a certain matrix of F and G coefficients; see [LOO82b], [BRT71], or [COL75] for the exact definition.) For $0 \leq j < n$, the *j^{th} principal subresultant coefficient of F and G*, written $psc_j(F,G)$, is the coefficient of x^j in $S_j(F,G)$. We define $psc_n(F,G)$ to be $1 \in J$. Note that for $0 \leq j < n$, $psc_j(F,G) = 0$ if and only if $deg(S_j(F,G)) < j$.

The following theorem is the basis for definition of the first class of elements of $PROJ(A)$. Some notation: suppose F is an element of I_r. The *derivative* of F, written F', is the partial derivative of F with respect to x_r. $deg(F)$ is the degree of F in x_r. For $\alpha \in E^{r-1}$, we write $F_\alpha(x_r)$ or F_α to denote $F(\alpha, x_r)$.

THEOREM 3.1. *Let $F \in I_r, r \geq 2$, and let R be a region in E^{r-1}. Suppose that $deg(F_\alpha)$ is constant and nonnegative for $\alpha \in R$, and that if positive, then the least k such that $psc_k(F_\alpha, F'_\alpha) \neq 0$ is constant for $\alpha \in R$. Then F is delineable on R.*

A proof is given in [ACM82] (Theorem 3.6). The essential ideas are contained in the proof of Theorem 4 of [COL75]. [COL75] uses a definition of delineability stronger than ours, but a polynomial delineable by that definition is delineable by ours.

Theorem 3.1 suggests that for a $PROJ(A)$-invariant region R, and for each $A_i \in A$, we should have $deg((A_i)_\alpha)$ constant for $\alpha \in R$. This may be a nontrivial requirement. Suppose, for example, that $r = 3$ and A contains

$$F(x, y, z) = (y^2 + x^2 - 1)z^3 + (x - 1)z^2 + (x - 1)^2 + y^2.$$

If R is a region in the plane disjoint from the unit circle, then F_α has degree 3. If R is a subset of the unit circle which does not contain the point $< 1, 0 >$, then F_α has degree 2. If R is the point $< 1, 0 >$, then F_α is the zero polynomial. $PROJ$ must separate these cases. Theorem 3.1 also suggests that for any $PROJ(A)$-invariant region R on which $deg(F_\alpha)$ is constant and positive, we should insure that the least k such that $psc_k(F_\alpha, F'_\alpha) \neq 0$ is constant for $\alpha \in R$.

To achieve these goals, we introduce the notion of reductum of a polynomial. For any nonzero $F \in I_r = I_{r-1}[x_r]$, *ldcf(F)* denotes the leading coefficient of F. The *leading term* of F, written *ldt(F)*, is

$$ldcf(F) \cdot x_r{}^{deg(F)}.$$

The *reductum* of F, written *red(F)*, is $F - ldt(F)$. If $F = 0$, we define $red(F) = 0$. For any $k \geq 0$, the *kth reductum of F*, written $red^k(F)$, is defined by induction on k:

$$red^0(F) = F.$$

$$red^{k+1}(F) = red(red^k(F)).$$

For any $F \in I_r$, the *reducta set of F*, written $RED(F)$, is

$$\{red^k(F) | 0 \leq k \leq deg(F) \ \& \ red^k(F) \neq 0\}.$$

Thus the reducta set of our sample $F(x, y, z)$ above is

$$\{F(x, y, z), (x - 1)z^2 + (x - 1)^2 + y^2, (x - 1)^2 + y^2\}.$$

We now incorporate reducta into a specification of the (first) desired property of a $PROJ(A)$-invariant region R. For each $F \in A$, there should exist an m such that $deg(F_\alpha) = m$ for all $\alpha \in R$. Furthermore, if m is positive, then where i is such that $deg(red^i(F)) = m$, and $Q = red^i(F)$, the least k such that $psc_k(Q_\alpha, Q'_\alpha) \neq 0$ should be constant for $\alpha \in R$.

Let F and G be nonzero elements of $I_r[x]$. Let $n = min(deg(F), deg(G))$. The *psc set of F and G*, written $PSC(F, G)$, is

$$\{psc_j(F, G) \mid 0 \leq j \leq n \ \& \ psc_j(F, G) \neq 0\}.$$

If either $F = 0$ or $G = 0$, then $PSC(F, G)$ is defined to be the empty set. Let $A = \{A_1, \ldots, A_n\}, n \geq 1$, be a set of polynomials in $I_r, r \geq 2$. $PROJ_1(A) \subset I_{r-1}$, the first class of polynomials in $PROJ(A)$, is defined as follows: For each i, $1 \leq i \leq n$, let $R_i = RED(A_i)$. Then

$$PROJ_1(A) = \bigcup_{i=1}^{n} \bigcup_{G_i \in R_i} (\{ldcf(G_i)\} \cup PSC(G_i, G_i')).$$

With the following simple observation, we can prove that $PROJ_1$ behaves as we want. Suppose F and G are nonzero elements of I_r, and suppose that for some $\alpha \in E^{r-1}$, $deg(F) = deg(F_\alpha) \geq 0$, and $deg(G) = deg(G_\alpha) \geq 0$. Let $n = min(deg(F), deg(G))$. Then for every j, $0 \leq j \leq n$, it is the case that $(psc_j(F, G))_\alpha = psc_j(F_\alpha, G_\alpha)$. We see this as follows: For $j < n$, since $deg(F) = deg(F_\alpha)$ and $deg(G) = deg(G_\alpha)$, the matrix obtained by evaluating the entries of the Sylvester matrix of F and G at α is just the Sylvester matrix of F_α and G_α, hence if $j < n$ then $(S_j(F, G))_\alpha$ is equal to $S_j(F_\alpha, G_\alpha)$, and so $(psc_j(F, G))_\alpha = psc_j(F_\alpha, G_\alpha)$. If $j = n$, then $(psc_j(F, G))_\alpha = psc_j(F_\alpha, G_\alpha) = 1$.

THEOREM 3.2. *For $A \subset I_r, r \geq 2$, if R is a $PROJ_1(A)$ -invariant region in E^{r-1}, then every element of A is either delineable or identically zero on R.*

Proof. Consider any $F \in A$. If $F = 0$, then F is identically zero on R. Suppose $F \neq 0$. By definition, $PROJ_1(A)$ includes every nonzero coefficient of F, so each coefficient of F either vanishes everywhere or nowhere on R. Hence $deg(F_\alpha)$ is constant for $\alpha \in R$. Let $deg_R(F)$ denote this constant value. If $deg_R(F) = -\infty$, then F is identically zero on R. If $deg_R(F) = 0$, then obviously F is delineable on R. Suppose $deg_R(F) \geq 1$. Then there is a unique reductum Q of F such that $deg(Q) = deg_R(Q) = deg_R(F)$. Then $F_\alpha = Q_\alpha$ for all $\alpha \in R$, hence if Q is delineable on R, then F is delineable on R. Since $PSC(Q, Q') \subset PROJ(A)$, the least k such that $(psc_k(Q, Q'))_\alpha \neq 0$ is constant for $\alpha \in R$. Hence by our observation above, the least k such that $psc_k(Q_\alpha, Q'_\alpha) \neq 0$ is constant for $\alpha \in R$. Hence by Theorem 3.1, Q is delineable on R, hence F is delineable on R. Thus every element of A is either identically zero or delineable on R. ∎

The following theorem is the basis for definition of the second class of elements of $PROJ(A)$.

THEOREM 3.3. *Let $A \subset I_r$, $r \geq 2$, and let R be a region in E^{r-1}. Suppose that for every $F \in A$, the hypotheses of Theorem 3.1 are satisfied. Suppose also that for every $F, G \in A$, $F \neq G$, the least k such that $psc_k(F_\alpha, G_\alpha) \neq 0$ is constant for $\alpha \in R$. Then every $F \in A$ is delineable on R, and for every $F, G \in A$, any F-section and any G-section of $Z(R)$ are either disjoint or identical.*

A proof is given in [ACM82] (Theorem 3.7). The essential ideas are contained in the proof of Theorem 5 of [COL75].

Let A and R_i be as in the definition of $PROJ_1$. Let

$$PROJ_2(A) = \bigcup_{1 \leq i < j \leq n} \bigcup_{G_i \in R_i \ \& \ G_j \in R_j} PSC(G_i, G_j).$$

We define $PROJ(A)$ to be the union of $PROJ_1(A)$ and $PROJ_2(A)$. The following theorem establishes that $PROJ$ works, i.e., that conditions (1′) and (2) are satisfied for a $PROJ(A)$-invariant region:

THEOREM 3.4. *For $A \subset I_r, r \geq 2$, if R is a $PROJ(A)$-invariant region in E^{r-1}, then every element of A is either delineable or identically zero on R, and for every $F, G \in A$, any F-section and any G-section of $Z(R)$ are either disjoint or identical.*

Proof. By Theorem 3.2, every element of A is either delineable or identically zero on R. By an argument similar to that used in the proof of Theorem 3.2, but with Theorem 3.3 in place of Theorem 3.1, it follows that for every $F, G \in A$, any F-section and any G-section of $Z(R)$ are either disjoint or identical. ■

This completes the proof that if R is a $PROJ(A)$-invariant region in E^{r-1}, then there exists an A-invariant stack over R, namely the stack whose sections are the sections of those A_i's in A which are delineable on R. In general, when every element of $A \subset I_r$ is either delineable or identically zero on some $R \subset E^{r-1}$, we write $S(A, R)$ to denote this stack. Our agenda for this Section will be completed by showing that if also R is semi-algebraic, then $S = S(A, R)$ is algebraic. By our remarks in Section 2, it suffices to show that each region of S is definable. Let x denote $< x_1, \ldots, x_{r-1} >$ and y denote x_r. Any section of S is an F-section of $Z(R)$ for some $F \in A$ which is delineable on R; say that it is the jth section of $S(F, R)$ (where sections are numbered from bottom to top). Then we can define it as the set of $< x, y >$ satisfying a formula "$x \in R$ and y is the jth real root of $F(x, y)$". If ϕ is a defining formula for R, then the following is such a formula:

$$\phi(x) \;\&\; (\exists y_1)(\exists y_2)\cdots(\exists y_{j-1})\;[\; y_1 < y_2 < \cdots < y_{j-1} < y$$

$$\&\; F(x, y_1) = 0 \;\&\; F(x, y_2) = 0 \;\&\; \cdots \;\&\; F(x, y_{j-1}) = 0 \;\&\; F(x, y) = 0$$

$$\&\; (\forall y_{j+1})\;\{\; (y_{j+1} \neq y_1 \;\&\; y_{j+1} \neq y_2 \;\&\; \cdots \;\&\; y_{j+1} \neq y_{j-1} \;\&$$

$$y_{j+1} \neq y \;\&\; F(x, y_{j+1}) = 0) \implies y_{j+1} > y \;\}\;].$$

The sectors of S can now be defined using the defining formulas for the sections: a sector is either the set of $< x, y >$ between two sections of S, or the set of $< x, y >$ above the topmost section of S, or the set of $< x, y >$ below the bottommost section of S. This concludes the proof.

4. The cylindrical algebraic decomposition algorithm: base phase. Let us use the precise definition of cad given in Section 2 to give precise specifications for a cad algorithm. Its input is a set $A \subset I_r$, $r \geq 1$. Its output is a description of an A-invariant cad D of E^r. This description should inform one of the number of cells in the cad, how they are arranged into stacks, and the sign of each element of A on each cell. We define in this section the index of a cell in a cad; our cad algorithm meets the first two of the above requirements by producing a list of indices of the cells of the cad of E^r that it constructs. We also define in this section an exact representation for algebraic points in E^r, that is, points whose coordinates are all real algebraic numbers. Our cad algorithm constructs, for each cell, an exact representation of a particular algebraic point belonging to that cell (we call this a *sample point* for the cell). The sign of $A_i \in A$ on a particular cell can then be determined by evaluating A_i (exactly) at the cell's sample point, and in this way we meet the third requirement above.

Where $A \subset I_r$ is the input to the cad algorithm, in the projection phase we computed $PROJ(A)$, $PROJ^2(A)$, and finally $K = PROJ^{r-1}(A) \subset I_1$. It is the task of the base phase to construct a K-invariant cad D^* of E^1, that is, to construct cell indices and sample points for the cells of such a cad. Let us now define cell indices.

In a cylindrical decomposition of E^1, the index of the leftmost 1-cell (the 1-cell with left endpoint $-\infty$), is (1). The index of the 0-cell (if any) immediately to its right is (2), the index of the 1-cell to the right of that 0-cell (if any) is (3), etc. Suppose that cell indices have been defined for cylindrical decompositions of E^{r-1}, $r \geq 2$. Let D be a cylindrical decomposition of E^r. D induces a cylindrical decomposition D' of

E^{r-1}. Any cell d of D is an element of a stack S over a cell c of D'. Let $(i_1, \ldots, i_{r-1})$ be the index of c. The cells of S may be numbered from bottom to top, with the bottommost sector being called cell 1, the section above it (if any) cell 2, the sector above that (if any) cell 3, etc. If d is the jth cell of the stack by this numbering, then its cell index is $(i_1, \ldots, i_{r-1}, j)$.

The sum of the parities of the components of a cell index is the dimension of the cell (even parity $= 0$, odd parity $= 1$). In a cylindrical decomposition of E^2, for example, cell (2,4) would be a 0-cell, (2,5) would be a 1-cell.

We begin the base phase by constructing the set of all distinct (i.e., relatively prime) irreducible factors of nonzero elements of K (see [KAL82] for polynomial factorization algorithms). Let $M = \{M_1, \ldots, M_k\} \subset I_1$ be the set of these factors. The real roots $\alpha_1 < \cdots < \alpha_n$, $n \geq 0$, of $\prod M$ will be the 0-cells of D^* (if $n = 0$ then D^* consists of the single 1-cell E^1). We determine the α_j's by isolating the real roots of the individual M_i's [CLO82]. By their relative primeness, no two elements of M have a common root. Hence by refining the isolating intervals for the α_j's, we obtain a collection of disjoint left-open and right-closed intervals $(r_1, s_1]$, $(r_2, s_2], \ldots, (r_n, s_n]$ with rational endpoints, each containing exactly one α_j, and with $r_1 < s_1 \leq r_2 < \cdots$

As soon as we know n, we can trivially write down the indices of the $2n+1$ cells of D^*. Clearly each cell is definable, hence semi-algebraic. To describe sample point construction, we first define a representation for an algebraic point in E^i, $i \geq 1$. Loos ([LOO82a], Section 1) describes the representation of a real algebraic number γ by its minimal polynomial $M(x)$, and an isolating interval for a particular root of $M(x)$. With γ so represented, and letting $m = deg(M)$, one can represent any element of $Q(\gamma)$ as an element of $Q[x]$ of degree $\leq m-1$ (as Loos describes). For an algebraic point in E^i, there exists a real algebraic γ such that each coordinate of the point is in $Q(\gamma)$; γ is a *primitive element* for the point. Our representation for the point is: a primitive element γ and an i-tuple of elements of $Q(\gamma)$, all represented as described by Loos.

For the 1-cells of D^* we primarily use appropriately chosen (rational) endpoints from the isolating intervals above as sample points. However, if $s_i = r_{i+1}$ is a 0-cell, we find (by bisection) a positive rational ϵ, such that $(r_{i+1} + \epsilon, s_{i+1}]$ isolates α_{i+1}, and use $r_{i+1} + \epsilon$ as sample point for cell $(2i+1)$. Also, we use $s_n + 1$ as a sample point for cell $(2n+1)$. If $D^* = \{E^1\}$, we use an arbitrary rational number. Obviously the only point in a 0-cell is the cell itself. Its value is an algebraic number. Thus all our sample points for D^* are algebraic numbers, and hence can be trivially expressed in our just-defined algebraic point representation. Examples of sample points for a cad of E^1 are given in Section 6.

5. The cylindrical algebraic decomposition algorithm: extension phase. First, consider the extension of the cad D^* of E^1 to a cad of E^2. In the projection phase, we computed a set $J = PROJ^{r-2}(A) \subset I_2$. Let c be a cell of D^*. We want to construct the stack $S(J, c)$ (as defined following Theorem 3.4). Let α be the sample point for c, and let $J_c(y)$ be the product of all nonzero $G(\alpha, y)$, $G \in J$ (we construct $J_c \in Q(\alpha)[y]$ using algorithms for exact arithmetic in $Q(\alpha)$ [LOO82a]). We isolate the real roots of $J_c(y)$ ([LOO82a], Section 2). This determines $S(J, c)$: β is a root of $J_c(y)$ if and only if $< \alpha, \beta >$ lies on a section of $S(J, c)$. For each such β, we use the representation for α, the isolating interval for β, and the algorithms NORMAL and SIMPLE of [LOO82a] to construct a primitive element γ for $Q(\alpha, \beta)$; we use γ to construct a representation of the form we require for $< \alpha, \beta >$. We get sector sample points for $S(J, c)$ from α and the (rational) endpoints of the isolating intervals for the roots of J_c, much as

was done in Section 4 for E^1. Thus sector sample points are of the form $< \alpha, r >$, r rational, so we can take $\gamma = \alpha$ for them. Given the cell index for c, and the isolated roots of J_c, we can trivially write down the indices for the cells of $S(J, c)$ (as for E^1 in Section 4).

After processing each cell c of D^* in this fashion, we have determined a cad of E^2 and constructed a sample point for each cell.

Extension from E^{i-1} to E^i for $3 \leq i \leq r$ is essentially the same as from E^1 to E^2. The only difference is that a sample point in E^{i-1} has $i-1$, instead of just one, coordinates. But where α is the primitive element of an E^{i-1} sample point, and $F = F(x_1, \ldots, x_i)$ an element of I_i, arithmetic in $Q(\alpha)$ still suffices for constructing the univariate polynomial over $Q(\alpha)$ that results from substituting the coordinates $< \alpha_1, \ldots, \alpha_{i-1} >$ of the sample point for $< x_1, \ldots, x_{i-1} >$ in F.

The following abstract algorithm summarizes our discussion of the cad algorithm.

CAD(r,A;I,S)

Inputs: r is a positive integer. A is a list of $n \geq 0$ integral polynomials in r variables.

Outputs: I is a list of the indices of the cells comprising an A-invariant cad D of E^r. S is a list of sample points for D.

(1) [$r = 1$]. If $r > 1$ then go to 2. Set $I \leftarrow$ the empty list. Set $S \leftarrow$ the empty list. Isolate the real roots of the irreducible factors of the nonzero elements of A. Construct the indices of the cells of D and add them to I. Construct sample points for the cells of D and add them to S. Exit.

(2) [$r > 1$]. Set $P \leftarrow PROJ(A)$. Call CAD recursively with inputs $r-1$ and P to obtain outputs I' and S' that specify a cad D' of E^{r-1}. Set $I \leftarrow$ the empty list. Set $S \leftarrow$ the empty list. For each cell c of D', let i denote the index of c, let α denote the sample point for c, and carry out the following four steps: first, set $A_c(x_r) \leftarrow \prod\{A_j(\alpha, x_r) | A_j \in A \;\&\; A_j(\alpha, x_r) \neq 0\}$; second, isolate the real roots of $A_c(x_r)$; third, use i, α, and the isolating intervals for the roots of A_c to construct cell indices and sample points for the sections and sectors of $S(A, c)$; fourth, add the new indices to I and the new sample points to S. Exit. ■

6. An example. We now show what algorithm CAD does for a particular example in E^2. Let

$$A_1(x, y) = 144y^2 + 96x^2y + 9x^4 + 105x^2 + 70x - 98,$$
$$A_2(x, y) = xy^2 + 6xy + x^3 + 9x,$$

and $A = \{A_1, A_2\}$. CAD is called with input A. We compute $PROJ(A)$:

$ldcf(A_1) = 144,$

$psc_0(A_1, A_1') = -580608(x^4 - 15x^2 - 10x + 14),$

$psc_1(A_1, A_1') = 1,$

$ldcf(red(A_1)) = 96x^2,$

$psc_0(red(A_1), [red(A_1)]') = 1,$

$ldcf(red^2(A_1)) = 9x^4 + 105x^2 + 70x - 98,$

$ldcf(A_2) = x,$

$psc_0(A_2, A_2') = 4x^5,$

$psc_1(A_2, A_2') = 1,$

$ldcf(red(A_2)) = 6x,$

$psc_0(red(A_2), [red(A_2)]') = 1,$

$ldcf(red^2(A_2)) = x(x^2 + 9),$

$psc_0(A_1, A_2) = x^2(81x^8 + 3330x^6 + 1260x^5 - 37395x^4 - 45780x^3 - 32096x^2 + 167720x + 1435204)$,
$psc_1(A_1, A_2) = 96x(x^2 - 9)$,
$psc_2(A_1, A_2) = 1$,
$psc_0(red(A_1), A_2) = x(81x^8 + 5922x^6 + 1260x^5 + 31725x^4 - 25620x^3 + 40768x^2 - 13720x + 9604)$,
$psc_1(red(A_1), A_2) = 1$,
$psc_0(A_1, red(A_2)) = -36x(3x^4 - 33x^2 - 70x - 226)$,
$psc_1(A_1, red(A_2)) = 1$,
$psc_0(red(A_1), red(A_2)) = 1$.

It turns out that the roots of $p_1(x) = x^4 - 15x^2 - 10x + 14$ and $p_2(x) = x$ give us a "silhouette" of $V(A_1) \cup V(A_2)$, hence for simplicity in this example, let us set $PROJ(A) = \{p_1(x), p_2(x)\}$ (in general, $PROJ(A)$ may contain superfluous elements; [COL75] and [ARN81] describe techniques for detecting and eliminating such elements).

p_1 and p_2 are both irreducible, so we have $M_1 = p_1$ and $M_2 = p_2$ in the notation of Section 5. M_1 has four real roots with approximate values -3.26, -1.51, 0.7, and 4.08; M_2 has the unique root $x = 0$. The following collection of isolating intervals for these roots satisfies the conditions set out in Section 5:

$$(-4, -3], (-2, -1], (-1, 0], (\frac{1}{2}, 1], (4, 8].$$

Since there are five 0-cells, the cell indices for the cad are (1), (2), ..., (11).

We now construct representations for the sample points of the induced cad of E^1. Each 1-cell will have a rational sample point, hence any rational γ will be a primitive element. We arbitrarily choose $\gamma = 0$. $(-1, 0]$ is an isolating interval for γ as a root of its mimimal polynomial x. We may take the 1-cell sample points to be -4, -2, -1, $\frac{1}{2}$, 4, and 9. The four irrational 0-cells have as their primitive elements the four roots of $M_1(x)$. The representation for the leftmost 0-cell, for example, consists of $M_1(x)$, the isolating interval $(-4, 3]$ for the leftmost root γ of M_1, and the 1-tuple $< x >$, where x corresponds to the element γ of $Q(\gamma)$. The 0-cell $x = 0$ is represented in the same fashion as the rational 1-cell sample points.

We now come to the extension phase of the algorithm. Let c be the leftmost 1-cell of the cad D' of E^1. $A_1(-4, y) \neq 0$ and $A_2(-4, y) \neq 0$, hence

$$A_c(y) = A_1A_2(-4, y) = 24(y^2 + 6y + 25)(24y^2 + 256y + 601).$$

$y^2 + 6y + 24$ has no real roots, but $24y^2 + 256y + 601$ has two real roots, which can be isolated by the intervals $(-8, -7]$ and $(-4, -2]$. Thus the stack $S(A, c)$ has two sections and three sectors; the indices for these cells are (1,1), (1,2), ..., (1,5). From the endpoints of the isolating intervals we obtain sector sample points of $< -4, -8 >$, $< -4, -4 >$, and $< -4, -1 >$ (which will be represented in the customary fashion). The two roots γ_1 and γ_2 of $24y^2 + 256y + 601$ are not only y-coordinates for the section sample points, but also primitive elements for these sample points. Thus the (representations for the) section sample points are

$$\{24y^2 + 256y + 601, (-8, -7], < -4, y >\}$$

and

$$\{24y^2 + 256y + 601, (-4, -2], < -4, y >\}.$$

Now let c be the leftmost 0-cell of D'; let α also denote this point. $A_1(\alpha, y) \neq 0$ and $A_2(\alpha, y) \neq 0$; we have

$$A_c(y) = A_1 A_2(\alpha, y) = (y^2 + 6y + \alpha^2 + 9)(y + \frac{1}{3}\alpha^2)^2.$$

$y^2 + 6y + \alpha^2 + 9 \in Q(\alpha)[y]$ has no real roots, but obviously $y + \frac{1}{3}\alpha^2$ has exactly one; $(-8, 8]$ is an isolating interval for it. Hence $S(A, c)$ has one section and two sectors; the indices of these cells are (2,1), (2,2), and (2,3). The appropriate representations for $< -\alpha, -8 >$ and $< -\alpha, 9 >$ are the sector sample points. Since $y + \frac{1}{3}\alpha^2$ is linear in y, its root is an element of $Q(\alpha)$. Hence

$$\{M_1(x), (-4, 3], < x, -\frac{1}{3}x^2 >\}$$

is the representation of the section sample point. Thus in this particular case it was not necessary to apply the NORMAL and SIMPLE algorithms of [LOO82a] to find primitive elements for the sections of $S(A, c)$. In general, however, for a 0-cell $c = \alpha$ of D', $A_c(y)$ will have nonlinear factors with real roots, and it will be necessary to apply NORMAL and SIMPLE. Saying this another way, where α is a 0-cell of D' and $< \alpha, \beta >$ is a section sample point of D, we had in our example above $Q(\alpha, \beta) = Q(\alpha)$, but in general, $Q(\alpha)$ will be a proper subfield of $Q(\alpha, \beta)$.

The steps we have gone through above for a 1-cell and a 0-cell are carried out for the remaining cells of D' to complete the determination of the A-invariant cad D of E^2.

Although information of the sort we have described is all that would actually be produced by CAD, it may be useful to show a picture of the decomposition of the plane to which the information corresponds. The curve defined by $A_1(x, y) = 0$ has three connected components which are easily identified in Figure 5 below. The curve defined by $A_2(x, y) = 0$ is just the y-axis, i.e., the same curve as defined by $x = 0$, which cuts through the middle of the second component of $V(A_1)$. Figure 5 shows the A-invariant cad that CAD constructs.

We remark that the curve $A_1(x, y)$ is from ([HIL32], p. 329).

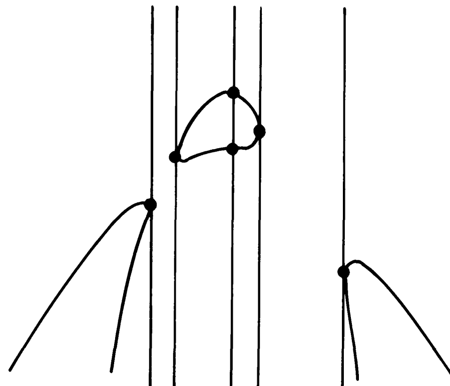


Fig. 5

REFERENCES

References for both Parts I and II are collected at the end of Part II.

SIAM J. COMPUT.
Vol. 13, No. 4, November 1984
© 1984 Society for Industrial and Applied Mathematics
015

CYLINDRICAL ALGEBRAIC DECOMPOSITION II: AN ADJACENCY ALGORITHM FOR THE PLANE*

DENNIS S. ARNON †, GEORGE E. COLLINS ‡, AND SCOTT McCALLUM §

Abstract. Given a set of r-variate integral polynomials, a *cylindrical algebraic decomposition (cad)* of euclidean r-space E^r partitions E^r into connected subsets compatible with the zeros of the polynomials. Each subset is a *cell*. Informally, two cells of a cad are *adjacent* if they touch each other; formally, they are adjacent if their union is connected. In applications of cad's one often wishes to know the adjacent pairs of cells. Previous algorithms for cad construction (such as that given in Part I of this paper) have not actually determined them. We give here in Part II an algorithm which determines the pairs of adjacent cells as it constructs a cad of E^2.

Key words. polynomial zeros, computer algebra, computational geometry, semi-algebraic geometry, real closed fields, decision procedures, real algebraic geometry

1. Introduction. In Part I of the present paper we defined cylindrical algebraic decompositions (cad's), and described an algorithm for cad construction. In Part II we give an algorithm that provides information about the topological structure of a cad of the plane. Informally, two disjoint cells in E^r, $r \geq 1$, are *adjacent* if they touch each other; formally, they are adjacent if their union is connected. In a picture of a cad, e.g., Figure 2 of Part I, it is obvious to the eye which pairs of cells are adjacent. However, the cad algorithm of Part I does not actually produce this information, nor did the original version of that algorithm in [COL75].

Adjacency information has already been an essential part of certain applications of cad's. For example, for $r = 2$ and $r = 3$, a cad construction algorithm has been developed which uses adjacency information to circumvent certain time-consuming steps of the original algorithm [ARN81]. Schwartz and Sharir use cell adjacency in their solution to the mover's problem [SCH83b], as do Arnon and McCallum in their recently developed algorithm to determine the topological type of a real algebraic curve [ARM84]. We remark that adjacency of cells is a slight generalization of the notion of incidence of cells in algebraic topology (for which one may consult [MAS78]).

We present here an algorithm which, given a set of polynomials $A \subset I_2$, constructs an A-invariant cad of E^2, and determines all pairs of adjacent cells in that cad. For certain inputs A, the cad constructed by this algorithm (which we call a "proper" cad) is different from the cad constructed by algorithm CAD of Part I. In a proper cad, knowing only some of the pairs of adjacent cells (those in which both cells are

*Received by the editors January 13, 1983, and in revised form January 2, 1984. This work was partially supported by the National Science Foundation, Grant MCS-8009357 to the University of Wisconsin-Madison, and by the Purdue Research Foundation. This paper was typeset at Xerox PARC using TEX in the Cedar environment. The final copy was produced on September 19, 1984.

† Xerox PARC, 3333 Coyote Hill Road, Palo Alto, California 94304. Formerly with Computer Science Department, Purdue University, West Lafayette, Indiana.

‡ Computer Science Department, University of Wisconsin-Madison, Madison, Wisconsin, 53706.

§ Department of Computer Science, University of Toronto, Toronto, Canada M5S 1A7. Formerly with Computer Science Department, University of Wisconsin-Madison, Madison, Wisconsin.

sections) suffices for determining the rest, and the algorithm for constructing the set of sufficient pairs is attractively simple.

The following definitions and observations are our starting point for adjacency determination. If R_1 and R_2 are adjacent regions, we call the set $\{R_1, R_2\}$ an *adjacency*. If both R_1 and R_2 are sections, it is a *section-section* adjacency. Recall that a cad of E^r, $r \geq 2$, is the union of certain stacks over the cells of some induced cad of E^{r-1}. Clearly the adjacencies of any cad can be divided into those in which both cells are in the same stack, and those in which the two cells involved are in different stacks. The first kind may be called *intrastack* adjacencies, and the second kind *interstack* adjacencies. To determine the intrastack adjacencies of a cad it suffices to know the number of sections in each of its stacks. This is because in any stack, each section is adjacent to the sector immediately below it, and to the sector immediately above it, and every adjacency involving two cells of that stack is of one of these two forms. Any algorithm for cad construction must determine how many sections are in each stack (cf. the specifications for cad algorithms in Section 4 of Part I). Hence, the only hard part of determining the adjacencies of a cad is determining the interstack adjacencies.

The contents of the paper is as follows. Section 2 defines the notion of a proper cad, and shows that in a proper cad of the plane, if one knows the section-section interstack adjacencies, then one can infer from them all other interstack adjacencies. Section 3 presents an algorithm (called SSADJ2) to determine the section-section interstack adjacencies between a pair of stacks in E^2 satisfying certain hypotheses (which will be satisfied by the stacks in a proper cad). In Section 4 we develop an algorithm (called CADA2) which, given $A \subset I_2$, constructs a proper A-invariant cad of E^2, and its adjacencies. Section 5 traces CADA2 on the example of Section 6 of Part I.

Algorithm CADA2 has been implemented [ARN81]. The basic idea of algorithm SSADJ2 is due to McCallum [MCC79]. We remark that Schwartz and Sharir [SCH83b] propose adjacency algorithms for cad's which are quite different from ours, and whose feasibilities have not yet been determined. The notion of proper cad (called "basis-determined cad") was first used in [ARN81]. A similar notion ("well-based cad") was used in [SCH83b].

2. Adjacencies in proper cylindrical algebraic decompositions. We say that a cad D of E^r is *proper* if (1) there exists some $F \in I_r$ such that $V(F)$ is equal to the union of the sections of D, and (2) if $r > 1$, then the induced cad D' of E^{r-1} is proper. When such an F exists for a cad D, we say it is a *defining polynomial* for D. Such polynomials are not unique. For example, given a defining polynomial $F(x, y)$ for a cad D of E^2, $(y^2 + 1)F$ is also one.

The following theorem provides a useful characterization of proper cad's. Recall that if D is a cad of E^r, then $D = \bigcup_{c \in D'} S(c)$, where for each $c \in D'$, $S(c)$ denotes the unique stack over c which is a subset of D.

THEOREM 2.1. *If D is a proper cad of E^r, $r \geq 2$, then (1) D' is proper, (2) any defining polynomial $F \in I_r$ for D is delineable on every $c \in D'$, and (3) for any defining polynomial F for D, $D = \bigcup_{c \in D'} S(F, c)$. Conversely, if D' is proper, if there exists $F \in I_r$ which is delineable on every $c \in D'$, and if $D = \bigcup_{c \in D'} S(F, c)$, then D is proper.*

Proof. Suppose D is proper, and suppose F is a defining polynomial for it. By definition of proper cad, D' is proper, and for any $c \in D'$, every section of $S(c)$ is in $V(F)$, and no sector of $S(c)$ meets $V(F)$. Hence F is delineable on c, and $S(c) = S(F, c)$. Thus $D = \bigcup_{c \in D'} S(F, c)$. Conversely, if D' is proper, if there exists

$F \in I_r$ which is delineable on every $c \in D'$, and if $D = \bigcup_{c \in D'} S(F, c)$, then clearly $V(F)$ is equal to the union of the sections of D, and so D is proper. ■

Let us illustrate how a cad constructed by algorithm CAD of Part I may not be proper. Given input $\{xy\}$, CAD would construct the $\{xy\}$-invariant cad of E^2 shown in Figure 1. A superscript attached to the name of a cell will denote its dimension, e.g., c^0 is a 0-cell, c^1 is a 1-cell.

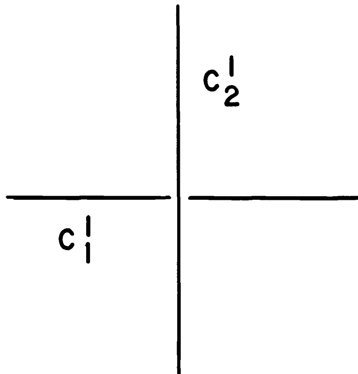


Fig. 1

Assume $F(x, y)$ is a defining polynomial for this cad. By definition, $c_1^1 \subset V(F)$. Hence it follows that $F(x, 0) = 0$ for all x, so $< 0, 0 > \in V(F)$. But $< 0, 0 >$ does not belong to any section of the cad, a contradiction.

Figure 2 shows a proper $\{xy\}$-invariant cad with defining polynomial y:

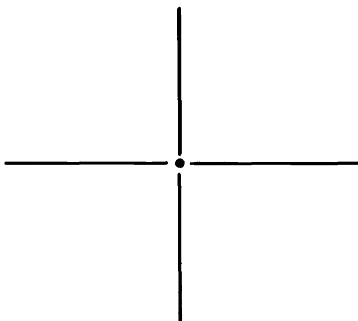


Fig. 2

In dealing with interstack adjacencies, it will be convenient not to have to treat points at infinity specially. We therefore introduce the following notation and terminology. We write E^* for $E \cup \{-\infty, +\infty\}$, the usual two-point compactification of the real line. For a subset X of any E^i, we write $Z^*(X)$ to denote $X \times E^*$, which we call the *extended cylinder* over X. If R is a region in E^i, a *section of* $Z^*(R)$ is either a section of $Z(R)$ or $R \times \{-\infty\}$ or $R \times \{+\infty\}$. $R \times \{-\infty\}$ and $R \times \{+\infty\}$ are respectively the *$-\infty$-section* and the *$+\infty$-section*, or collectively the *infinite sections*, of $Z^*(R)$. If S is a stack over R, then S^* is S plus the infinite sections of $Z^*(R)$. S^* is the *extension* of S, and we say also that it is an *extended stack* over R.

Given a cad $D = \bigcup_{c \in D'} S(c)$, for any $c \in D'$ we write $S^*(c)$ to denote the extension of $S(c)$. If c has cell index (i), and $S(c)$ has j sections, then the cell indices of the $-\infty$ and $+\infty$-sections of $S^*(c)$ are defined to be $(i, 0)$ and $(i, 2j + 2)$, respectively.

THEOREM 2.2. *Let $c^0 = \alpha$ be a 0-cell in E^1, let R be a region in E^1 which is*

adjacent to c^0, and let $F(x, y) \in I_2$ be such that $F(\alpha, y) \neq 0$ (thus F is delineable on c^0), and F is delineable on R. If s is a section of $S^*(F, R)$, then s has a unique limit point p in $Z^*(c^0)$, and p is a section of $S^*(F, c^0)$.

Proof. If s is an infinite section of $S^*(F, c^0)$, then the assertion is obvious. Suppose s is the graph of a continuous function $f : R \to E$. Then $F(x, f(x)) = 0$ for all $x \in R$, hence f is an algebraic function. Hence sufficiently close to c^0, f is monotone as $x \in R$ approaches c^0. Then where $\{a_1, a_2, \ldots\}$ is a sequence of points in R converging to c^0, the sequence $\{f(a_1), f(a_2), \ldots,\}$ converges to a limit γ in E^*, and $p = < \alpha, \gamma >$ is the unique limit point of s in $Z^*(c^0)$.

If $p = < \alpha, -\infty >$ or $p = < \alpha, +\infty >$ we are done. Suppose p is neither of these. It is a standard fact that the variety of a polynomial is a closed set. Hence $p \in V(F)$. By hypotheses, $F(\alpha, y) \neq 0$, hence it has finitely many real roots, hence since $S(F, c^0)$ is F-invariant, $F \neq 0$ at every point of every sector of $S(F, c^0)$. Hence p is a section of $S(c^0)$. ∎

We call p the *boundary section* of s in $S^*(F, c^0)$.

Our next theorem requires a general notion of boundary. For a subset X of a topological space T, the *boundary of* X, written ∂X, is $\bar{X} - X$ ($\bar{X}$ denotes the closure of X). One can easily show that ∂X is the set of all limit points of X which do not belong to X. We introduce some notation: for a region R in E^i, suppose that sections s_1 and s_2 of $Z^*(R)$ are respectively the f_1-section and f_2-section of $Z^*(R)$, and that $f_1 < f_2$. We write (s_1, s_2) to denote the (f_1, f_2)-sector of $Z(R)$, and $[s_1, s_2]$ to denote $s_1 \cup s_2 \cup (s_1, s_2)$ (see Part I, Section 2, for the notation f_1-section, f_2-section, and $(f_1 f_2)$-sector).

THEOREM 2.3. *Let $c^0 = \alpha$ be a 0-cell in E^1, let R be a region in E^1 which is adjacent to c^0, and let $F(x, y) \in I_2$ be such that $F(\alpha, y) \neq 0$, and F is delineable on R. Let $s = (s_1^1, s_2^1)$ be a sector of $S(F, R)$; let t_1^0 and t_2^0 be the respective boundary sections of s_1^1 and s_2^1 in $S^*(F, c^0)$. Then the portion of ∂s contained in $Z^*(c^0)$ is $[t_1^0, t_2^0]$.*

Proof. Suppose s_1^1 is the f_1-section, and s_2^1 the f_2-section, of $Z^*(R)$. The following argument will assume that both t_1^0 and t_2^0 are finite, but can easily be modified for the cases where either or both is infinite. Let p be a point of $[t_1^0, t_2^0]$. p can be written in the form

$$\gamma\, t_1^0 + (1 - \gamma) t_2^0, \quad 0 \leq \gamma \leq 1.$$

Let $\{a_1, a_2, \ldots,\}$ be a sequence of points in R converging to c^0. Then the sequence

$$\{< a_1, \gamma f_1(a_1) + (1 - \gamma) f_2(a_1) >, \ldots\}$$

in s converges to p, hence p is a limit point of s. Suppose p is a limit point of s in $Z^*(c^0)$. Then there is some sequence of points $\{< x_i, y_i >\}$ in s converging to p. We have $f_1(x_i) < y_i < f_2(x_i)$, hence $p = \lim\{< x_i, y_i >\}$ is an element of $[t_1^0, t_2^0]$. Hence $\partial s \cap Z^*(c^0) = [t_1^0, t_2^0]$. ∎

Given two disjoint regions, it is not difficult to see that they are adjacent if and only if one contains a limit point of the other. Using this fact, we now show that if c^0, R, and $F(x, y)$ are as in the hypotheses of Theorems 2.2 and 2.3, then all interstack adjacencies between $S(F, c^0)$ and $S(F, R)$ can be determined from their section-section interstack adjacencies. Let s^1 be a section of $S^*(F, R)$. By Theorem 2.2 and the fact that regions are adjacent if and only if one contains a limit point of the other, if s^1 is adjacent to a cell of $S^*(F, c^0)$, then that cell is a section of $S^*(F, c^0)$. Hence any interstack adjacency involving a section of $S^*(F, R)$ is a section-section adjacency. Let $s^2 = (s_1^1, s_2^1)$ be a sector of $S^*(F, R)$. Where t_1^0 and t_2^0 are the respective boundary sections of s_1^1 and s_2^1 in $S^*(F, c^0)$, by Theorem 2.3, s^2 is adjacent to all cells of $S^*(F, c^0)$

between t_1^0 and t_2^0 inclusive, and to only those cells of $S^*(F, c^0)$. Hence knowledge of the section-section adjacencies between $S(F, c^0)$ and $S(F, R)$ suffices for determining all interstack adjacencies between them.

We now relate our general development to proper cad's. We first establish that we can make use of Theorems 2.2 and 2.3.

THEOREM 2.4. *Let D be a proper cad of E^2, let $c^0 = \alpha$ and c^1 be adjacent cells of D', and let $F(x, y) \in I_2$ be a defining polynomial for D. Then $F(\alpha, y) \neq 0$, F is delineable on c^1, $S(F, c^0) = S(c^0)$, and $S(F, c^1) = S(c^1)$.*

Proof. By Theorem 2.1, F is delineable on every $c \in D'$, and $D = \bigcup_{c \in D'} S(F, c)$. The conclusions of the theorem now follow directly. ■

Theorems 2.2 - 2.4 give us:

COROLLARY 2.5. *Let D be a proper cad of E^2, and let $c^0 = \alpha$ and c^1 be adjacent cells of D'. If s is a section of $S^*(c^1)$, then s has a unique limit point p in $Z^*(c^0)$, and p is a section of $S^*(c^0)$. If $s = (s_1^1, s_2^1)$ is a sector of $S(c^1)$, and t_1^0 and t_2^0 are the respective boundary sections of s_1^1 and s_2^1 in $S^*(c^0)$, then the portion of ∂s contained in $Z^*(c^0)$ is $[t_1^0, t_2^0]$.*

The same argument we used in the more general setting can be applied to show that if D is a proper cad of E^2, and if c^0 and c^1 are adjacent cells of D', then all interstack adjacencies between $S(c^0)$ and $S(c^1)$ can be inferred from their section-section interstack adjacencies. It follows that all interstack adjacencies of D can be inferred from knowledge of its section-section interstack adjacencies.

3. Determination of section-section adjacencies. To find the section-section interstack adjacencies of a proper cad D of E^2, we use the following strategy: for each 0-cell c^0 of D', and for the two 1-cells c_1^1 and c_2^1 of D' adjacent to c^0, we find all adjacencies between sections of $S^*(c^0)$ and sections of $S^*(c_1^1)$, and all adjacencies between sections of $S^*(c^0)$ and sections of $S^*(c_2^1)$. Algorithm SSADJ2, which we develop in this section, handles a particular triple c^0, c_1^1, c_2^1. The application of SSADJ2 to each triple c^0, c_1^1, c_2^1 of D' is done by algorithm CADA2 of Section 4.

Our results in this section apply to a setting similar to that which we had for Theorems 2.2 and 2.3. That is, we have a 0-cell $c^0 = \alpha$ in E^1, a region R in E^1 which is adjacent to c^0, and an $F(x, y) \in I_2$ such that $F(\alpha, y) \neq 0$, and F is delineable on R. The application of these general results to the context of proper cad's is straightforward.

Suppose e^0 is a section of $S(F, c^0)$. Theorems 3.1 and 3.2 below provide a method of determining how many sections of $S(F, R)$ are adjacent to e^0. This method can be described as: draw a suitable "box" centered at e^0; then the number of sections of $S(F, R)$ adjacent to e^0 is equal to the number of intersections of $V(F)$ with the left (right) side of the box (R is to the left (right) of c^0).

But knowing how many sections of $S(F, R)$ are adjacent to e^0 is not enough; we must determine *which* sections are adjacent to it. This we accomplish by processing the sections of $S(F, c^0)$ in consecutive order from bottom to top, after an initial step suggested by Theorem 3.3: determine how many sections of $S(F, R)$ are adjacent to the $-\infty$-section of $S^*(F, c^0)$, i.e. how many sections of $S(F, R)$ tend to $-\infty$ as $x \to \alpha$ in R. If there are $n \geq 0$ such sections, then by Theorem 2.2 they are the bottom n sections of $S(F, R)$. Next, we apply the "box" method of Theorems 3.1 and 3.2 to the bottommost section e^0 of $S(F, c^0)$. If we determine that there are $m \geq 0$ sections of $S(F, R)$ adjacent to e^0, then by Theorem 2.2 these are the $(n+1)^{st}, \ldots, (n+m)^{th}$ sections (counting upwards) of $S(F, R)$. Next, let e^0 be the

second (from the bottom) section of $S(c^0)$, and apply the box method to find out how many sections of $S(F,R)$ are adjacent to it. If there are $q \geq 0$ such sections, then these are the $(n+m+1)^{st}, \ldots, (n+m+q)^{th}$ sections of $S(F,R)$. We continue in this fashion until we have processed all sections of $S(F,c^0)$. If there remain sections "at the top" of $S(F,R)$ which are not adjacent to the top section of $S(F,c^0)$, then they are adjacent to the $+\infty$-section of $S^*(F,c^0)$ i.e., they tend to $+\infty$ as $x \to \alpha$ in R.

The above steps can be described as analyzing (either the left or the right) "sides" of a collection of "boxes" stacked on top of each other. This is indeed what SSADJ2 does, but with a refinement: a complete set of suitable boxes is determined at the beginning of the algorithm, all of which have the same width (they may have different heights). Thus one might picture a single "ladder", whose "compartments" are the "boxes" for the different sections of $S(F,c^0)$ (cf. Figure 8 in Section 6).

We now give Theorems 3.1 - 3.3, followed by SSADJ2.

THEOREM 3.1. *Let $c^0 = \alpha$ be a 0-cell in E^1, suppose for some $b \in E$, $\alpha < b$, that $F(x,y) \in I_2$ is delineable on $R = (\alpha, b]$, and suppose for $s,t,u \in E$, $s < t < u$, that t is the unique real root of $F(\alpha,y)$ in $[s,u]$, and neither $F(x,s)$ nor $F(x,u)$ has real roots in $[\alpha,b]$. Then the number of sections of $S(F,R)$ which are adjacent to the section $e^0 = <\alpha,t>$ of $S(F,c^0)$, is equal to the number of real roots of $F(b,y)$ in (s,u).*

Proof. Let σ be a section of $S(F,R)$ which is adjacent to e^0. Then e^0 is a limit point of σ, so clearly there is a point of σ in $(\alpha,b) \times (s,u)$. For some $y \in E$, $<b,y> \in \sigma$. If y is not in (s,u), then by the Intermediate Value Theorem, either $F(x,s)$ or $F(x,u)$ has a real root in $[\alpha,b]$, contrary to hypothesis. Hence $y \in (s,u)$, and σ gives rise to a real root of $F(x,b)$ in (s,u).

Consider any real root $y \in (s,u)$ of $F(x,b)$. By definition of $S(F,R)$, $<b,y>$ lies in a section σ of $S(F,R)$. If σ contains any point outside of $R \times (s,u)$, then by the Intermediate Value Theorem, we violate our hypotheses. By Theorem 2.2, σ has a limit point $<\alpha,z>$ in $Z^*(c^0)$. We must have $z \in [s,u]$, since $\sigma \subset R \times (s,u)$. Since $V(F)$ is closed, $<\alpha,z> \in V(F)$, hence by hypothesis, $z = t$, hence σ is adjacent to $<\alpha,t>$. ■

Let us see an example of how we will use Theorem 3.1. Let $F = y^2 - x^3$, $\alpha = 0$, $b = 1$, $s = -2$, $t = 0$, and $u = 2$. Figure 3 shows the curve defined by $F = 0$, and Figure 4 illustrates our complete situation:

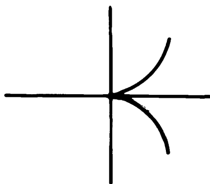


Fig. 3

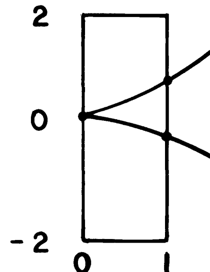


Fig. 4

Clearly we satisfy the hypotheses of Theorem 3.1. From the fact that $F(1,y)$ has

two real roots in $(-2, 2)$, we learn that there are two sections of $S(F, R)$ adjacent to the section $< 0, 0 >$ of $S(F, c^0)$.

There is a companion to Theorem 3.1 which differs from it only in that $b < \alpha$ rather than $\alpha < b$.

We must deal with the following problem before we can apply Theorem 3.1 to proper cad's. Suppose, for a proper cad D of E^2 with defining polynomial $F(x, y)$, that we have constructed the induced cad D' of E^1. Consider a 0-cell $c^0 = \alpha$ of D' and the adjacent 1-cell d^1 of D' immediately to its right. Let t be a particular real root of $F(\alpha, y)$. By isolating the real roots of $F(\alpha, y)$ we can obtain s and u such that $s < t < u$ and t is the unique real root of $F(\alpha, y)$ in $[s, u]$. Let b be the sample point for d^1; F is delineable on $(\alpha, b]$. Thus we have found α, b, s, t, and u which nearly satisfy the hypotheses of Theorem 3.1, except that $F(x, s)$ or $F(x, u)$ might have a real root in $[\alpha, b]$. This can indeed occur, as we show in Figure 5 for $F = y^2 - x^3$, $\alpha = 0$, $b = 1$, $s = -\frac{1}{2}$, $t = 0$, and $u = \frac{1}{2}$.

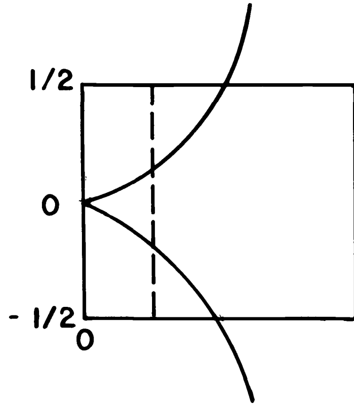


Fig. 5

Theorem 3.2 establishes that we can "shrink" the width of the box, i.e., move b closer to α, so that neither $F(x, s)$ nor $F(x, u)$ has a real root in $[\alpha, b]$.

THEOREM 3.2. *Suppose that the hypotheses of Theorem 3.1 are fulfilled, except that possibly either $F(x, s)$ or $F(x, u)$ has a real root in $[\alpha, b]$. Then there exists b^* in E, $\alpha < b^* \leq b$, such that neither $F(x, s)$ nor $F(x, u)$ has a real root in $[\alpha, b^*]$.*

Proof. By hypothesis, $F(x, s) \neq 0$ and $F(x, u) \neq 0$. Hence each has finitely many real roots, hence since $F(\alpha, s) \neq 0$ and $F(\alpha, u) \neq 0$, there exists b^*, $\alpha < b^* \leq b$, such that neither $F(x, s)$ nor $F(x, u)$ has real roots in $[\alpha, b^*]$. ∎

The proof of the next theorem is similar to the proof of Theorem 3.1.

THEOREM 3.3. *Let $c^0 = \alpha$ be a 0-cell in E^1, suppose for some $b \in E$, $\alpha < b$, that $F(x, y) \in I_2$ is delineable on $R = (\alpha, b]$, and suppose for $s \in E$ that $F(\alpha, y)$ has no real roots in $(-\infty, s]$, and $F(x, s)$ has no real roots in $[\alpha, b]$. Then the number of sections of $S(F, R)$ which are adjacent to the $-\infty$-section of $S^*(F, c^0)$ is equal to the number of real roots of $F(b, y)$ in $(-\infty, s)$.*

Consider an example. Let $F = xy + 1$, $\alpha = 0$, $b = 1$, and $s = 0$. Figure 6 shows the curve $F = 0$, and Figure 7 illustrates the overall situation:

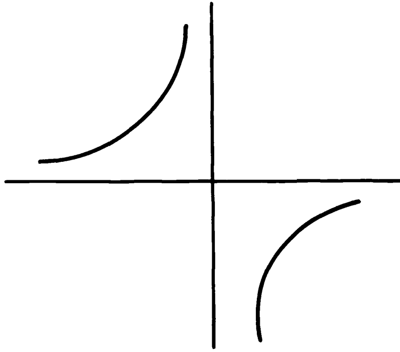


Fig. 6

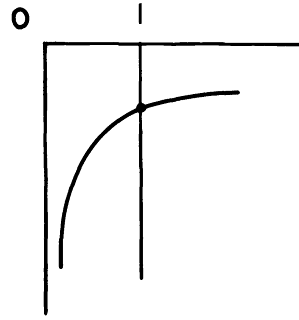


Fig. 7

From the fact that $F(1, y)$ has one real root in $(-\infty, 0)$, we learn that there is one section of $S(F, R)$ adjacent to the $-\infty$-section of $S^*(F, c^0)$.

There are companions to Theorem 3.3 with $b < a$ rather than $a < b$, and with $+\infty$ in place of $-\infty$. Also, a result similar to Theorem 3.2, but tailored for 3.3 instead of 3.1, exists.

We now give algorithm SSADJ2. We adopt the convention that the sections of a stack are numbered consecutively from bottom to top, starting with section 1 (the lowest finite section), $2, \ldots, n$ (the highest finite section). The sections of the stack's extension are numbered starting with section 0 (the $-\infty$-section), 1 (the lowest finite section), $2, \ldots, n$ (the highest finite section), $n + 1$ (the $+\infty$-section).

SSADJ2($F(x, y)$,α,b_1,b_2;L_1,L_2)

Inputs: $F(x, y)$ is an element of I_2. α is a real algebraic number such that $F(\alpha, y) \neq 0$ (we view α as also being the 0-cell c^0 in the real line). b_1 and b_2 are rational numbers such that $b_1 < \alpha < b_2$, F is delineable on $R_1 = [b_1, \alpha)$, and F is delineable on $R_2 = (\alpha, b_2]$.

Outputs: L_1 is a list of all section-section interstack adjacencies between $S^*(F, c^0)$ and $S^*(F, R_1)$. L_2 is a list of all section-section interstack adjacencies between $S^*(F, c^0)$ and $S^*(F, R_2)$.]

(1) [Construct tops and bottoms of "ladder compartments".] Set $L_1 \leftarrow$ the empty list and $L_2 \leftarrow$ the empty list. Isolate the real roots $t_1, \ldots, t_m$, $(m \geq 0)$, of $F(\alpha, y)$, obtaining rational $s_0, \ldots, s_m$, such that $s_0 < t_1 < s_1 < \cdots < t_m < s_m$. (If $m = 0$, set $s_0 \leftarrow$ an arbitrary rational number).

(2) [Construct left and right "sides" of "ladder".] Set $u \leftarrow b_1$ and $v \leftarrow b_2$. While there is an s_j, $0 \geq j \geq m$, such that $F(x, s_j)$ has a real root in $[u, v]$, set b^* to a rational approximate midpoint of (u, v) different from α. Set u, v to whichever of b_1,b^* and b^*,b_2 yields the property that (u, v) contains α.

(3) [Adjacencies of the $-\infty$-section of $S^*(F, c^0)$ and sections of $S^*(F, R_1)$.] Set $n \leftarrow$ the number of real roots of $F(u, y)$ in $(-\infty, s_0)$. Record in L_1 that section 0 of $S^*(F, c^0)$ is adjacent to sections $0, 1, \ldots, n$ of $S^*(F, R_1)$.

(4) [Adjacencies of finite sections of $S^*(F, c^0)$ and finite sections of $S^*(F, R_1)$.] For $j = 1, \ldots, m$ do the following three things: First, set $n_j \leftarrow$ the number of real roots of $F(u, y)$ in (s_{j-1}, s_j). Second, record in L_1 that section j of $S^*(F, c^0)$ is adjacent to sections $n + 1, \ldots, n + n_j$ of $S^*(F, R_1)$. Third, set $n \leftarrow n + n_j$.

(5) [Adjacencies of the $+\infty$-section of $S^*(F, c^0)$ and sections of $S^*(F, R_1)$.] Set $n_{m+1} \leftarrow$ the number of real roots of $F(u, y)$ in $(s_m, +\infty)$. Record in L_1 that section $m+1$ of $S^*(F, c^0)$ is adjacent to sections $n+1, \ldots, n+n_{m+1}, n+n_{m+1}+1$ of $S^*(F, R_1)$.

(6) [Adjacencies of sections of $S^*(F, c^0)$ and sections of $S^*(F, R_2)$.] Repeat steps (3), (4), and (5) with v in place of u, and L_2 in place of L_1. Exit. ■

4. Construction of proper cylindrical algebraic decompositions. Theorem 4.1 tells us how, given $A \subset I_2$, to construct a proper A-invariant cad of E^2. For nonzero $F \in I_r = I_{r-1}[x_r]$, the *content* of F is the greatest common divisor of its coefficients. If $F = 0$, its content is 1. F is *primitive* if its content is 1. The *primitive part* of F, written $pp(F)$, is F divided its content. The reader may wish to refer back to Section 2 of Part I for other terms and notation used in the theorem.

THEOREM 4.1. *Let $A \subset I_2$, let A^* be the primitive part of the product of the nonzero elements of A, and let D' be the proper cad of E^1 for which $P(x) = \prod PROJ(A)$ is a defining polynomial. Then*
(1) A^* is delineable on every $c \in D'$.
(2) $\bigcup_{c \in D'} S(A^*, c)$ is a proper A-invariant cad of E^2, and A^* is a defining polynomial for it.

Proof. (1) For any nonzero $F \in I_2$, and any $\alpha \in E$, one can easily see that $F(\alpha, y) = 0$ if and only if α is a root of *content* (F). Hence since A^* is primitive, $A^*(\alpha, y) \neq 0$ for all $\alpha \in E$, and so A^* is delineable on every 0-cell of D'. Let c be a 1-cell of D'. Let A_i be any nonzero element of A. $content(A_i)$ divides $ldcf(A_i)$, hence since $ldcf(A_i) \in PROJ(A)$, and D' is $PROJ(A)$-invariant, $content(A_i)$ is nonvanishing at all points of c, hence $A_i(\alpha, y) \neq 0$ for all $\alpha \in c$, hence by Theorem 3.4 of Part I, A_i is delineable on c. By another application of the same theorem, it is clear that the product F of the nonzero elements of A is delineable on c. Since the content of F is the product of the contents of the nonzero A_i's, $V(F) \cap Z(c) = V(pp(F)) \cap Z(c)$, and so since $A^* = pp(F)$, A^* is delineable on c.
(2) Since A^* is delineable on every $c \in D'$, $\bigcup_{c \in D'} S(A^*, c)$ is a cad of E^2. By Theorem 2.1 it is proper, and clearly A^* is a defining polynomial for it. Let A_i be any nonzero element of A. If c is a 1-cell of D', then as just shown, A_i is delineable on c, and every A_i-section of $Z(c)$ is an A^*-section of $Z(c)$, hence $S(A^*, c)$ is A_i-invariant. If c is a 0-cell with sample point α, then either $content(A_i)$ vanishes at α and A_i is identically zero on c, or $content(A_i)$ does not vanish at α, A_i is delineable on c, and every A_i-section of $Z(c)$ is an A^*-section of $Z(c)$. In either case, $S(A^*, c)$ is A_i-invariant. Hence $S(A^*, c)$ is A-invariant. ■

Here now is our algorithm for construction of a proper A-invariant cad of E^2, and its adjacencies. It is not difficult to see that the arguments we pass to SSADJ2 each time we call it in step (3) satisfy the hypotheses on its inputs. Theorem 2.4 implies that the adjacencies we get back from SSADJ2 are indeed adjacencies of our cad of E^2.

CADA2(A;I,L,S)

Input: A is a subset of I_2.
Outputs: I is a list of the indices of the cells of a proper A-invariant cad D of E^2. L is a list of all adjacencies of D, plus the adjacencies involving infinite sections. S is a list of sample points for D.

(1) [Construct sample points for induced cad D' of E^1.] Set $P \leftarrow PROJ(A)$. Isolate the real roots of the irreducible factors of the nonzero elements of P to determine the 0-cells of D'. Construct a sample point for each cell of D'. Set $A^* \leftarrow$ primitive part of the product of the nonzero elements of A.
(2) [Construct cell indices, determine intrastack adjacencies for D.] Let $a_1 < a_2 < \cdots < a_{2n} < a_{2n+1}$, $n \geq 0$, be the sample points for D' (Each a_{2i+1} is a rational sample point for a 1-cell; each a_{2i} is an algebraic sample point for a 0-cell). Set

$I \leftarrow$ the empty list. Set $L \leftarrow$ the empty list. For $i = 1, \ldots, 2n+1$ do the following three things: first, isolate the real roots of $A^*(a_i, y)$ to determine the sections of a stack T in E^2, second, construct cell indices for the cells of T and add to I, third, record the intrastack adjacencies for T in L.

(3) [Section-section adjacency determination.] For $i = 1, \ldots, n$, call SSADJ2 with inputs A^*, a_{2i}, a_{2i-1}, and a_{2i+1}, and add the contents of its outputs L_1 and L_2 to L. (The section numbers which occur in the adjacencies returned by SSADJ2 must first be converted into the indices of the corresponding cells of D. For example, if the list L_1 returned by the ith call to SSADJ2 contains the adjacency $\{3, 2\}$, it must be converted to $\{(2i, 6), (2i-1, 4)\}$ before being added to L).

(4) [Inference of remaining interstack adjacencies.] Use the current contents of L to infer the remaining interstack adjacencies of D, as described at the end of Section 2. Add them to L.

(5) [Construct sample points.] Set $S \leftarrow$ the empty list. Use the sample points for D' and the isolating intervals constructed in step (2) to construct sample points for the cells of D, adding them to S. Exit. ■

5. An example. We now trace CADA2 for the sample cad of E^2 discussed in Section 6 of Part I. The input polynomials are:

$$A_1(x, y) = 144y^2 + 96x^2y + 9x^4 + 105x^2 + 70x - 98,$$
$$A_2(x, y) = xy^2 + 6xy + x^3 + 9x,$$

As in Part I, we may take $PROJ(A)$ to consist of the two elements $p_1(x) = x^4 - 15x^2 - 10x + 14$ and $p_2(x) = x$. The four real roots of p_1, which are -3.26, -1.51, 0.7, 4.08 (the presence of a decimal point in a number will indicate that its value is approximate), and the unique root $x = 0$ of p_2, become the 0-cells of D'. We take the 1-cell sample points to be -4, -2, -1, $\frac{1}{2}$, 4, and 9. We set A^* to be the primitive part of A_1A_2. Since A_1 has content 1 and A_2 has content x, $A^* = \frac{(A_1A_2)}{x}$, which is

$$144y^4 + 96x^2y^3 + 864y^3 + 9x^4y^2 + 825x^2y^2 + 70xy^2 + 1198y^2 + 150x^4y$$
$$+1494x^2y + 420xy - 588y + 9x^6 + 186x^4 + 70x^3 + 847x^2 + 630x - 882.$$

After constructing cell indices and determining intrastack adjacencies, CADA2 makes a total of five calls to SSADJ2, moving from left to right along the cells of the induced cad of 1-space. Let us look at the third of these calls. The inputs to SSADJ2 are $F = A^*$, $\alpha = 0$, $a_1 = -1$, and $a_2 = \frac{1}{2}$. Let c^0, c_1^1, and c_2^1 denote the cells of D' whose respective sample points are α, a_1, and a_2. SSADJ2 begins by isolating the real roots of $A^*(0, y)$, which are -3.0, -0.82, and 0.82. We obtain $s_0, \ldots, s_3 = -4, -1, 0, 1$. At the start of step 2, we set $u \leftarrow -1$ and $v \leftarrow \frac{1}{2}$. We check to see if $F(x, s_0) = F(x, -4)$ has a root in $[u, v]$. The real roots of $F(x, -4)$ are -4.58 and -3.30, so this does not happen. However, the real roots of $F(x, s_1) = F(x, -1)$ are -1.39 and -0.79, so since $u < -0.79 < v$, we shrink our "ladder". We set $u \leftarrow -\frac{1}{4}$, and continue checking for the new u and v. We find that $F(x, 0)$ has real roots of -1.25 and 0.68, and $F(x, 1)$ has no real roots, so we have completed step 2.

Figure 8 illustrates what occurred in step 2. We see that with the wide "ladder" we started with, the curve crosses one of the horizontal "rungs" (i.e. $y = s_1 = -1$), but when we shrink the "ladder", this no longer occurs. Thus for the narrower "ladder", the intersections of the curve with the vertical sides of a "compartment" are in 1-1 correspondence with the adjacencies in that compartment.

In step 3, we find that $F(-\frac{1}{4}, y)$ has no real roots in $(-\infty, -4)$, and so we record only that section 0 of $S^*(c^0)$ is adjacent to section 0 of $S^*(c_1^1)$. In step 4, we have

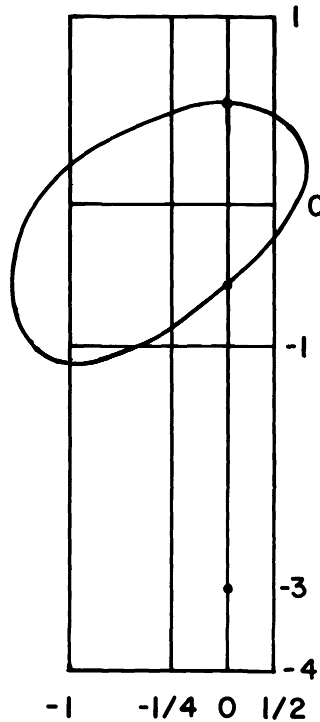


Fig. 8

$m = 3$, so the loop will be executed three times. The real roots of $F(-\frac{1}{4}, y)$ are -0.89 and 0.85; comparing these with $s_0, \ldots, s_3 = -4, -1, 0, 1$, we can see what will happen as we do the loop. The first time through we record no adjacencies (section 1 of $S^*(c^0)$ is an isolated point). The second time we record that section 2 of $S^*(c^0)$ is adjacent to section 1 of $S^*(c_1^1)$. The third time we record that section 3 of $S^*(c^0)$ is adjacent to section 2 of $S^*(c_1^1)$. In Step 5 we record only that section 4 of $S^*(c^0)$ (its $+\infty$-section) is adjacent to section 3 of $S^*(c_1^1)$ (its $+\infty$-section). The events of step 6 are similar.

We saw above that $A^*(0, y)$ has three real roots, and thus the cad D constructed by CADA2 has three sections over the 0-cell $c^0 = 0$ of D'. Yet the cad constructed by algorithm CAD of Part I for the same input, shown in Figure 5 of Part I, has only two sections over $c^0 = 0$. We have another example of how CAD and CADA2 may construct different cad's for the same input. Since $content(A_2) = x$, $A_2(0, y) = 0$, and CAD ignores the fact that the point $< 0, -3 >$ is in $V(pp(A_2))$, whereas CADA2 makes that point a section of $S(c^0)$.

REFERENCES

[ACM82] D. S. ARNON, G. E. COLLINS AND, S. MCCALLUM, *Cylindrical algebraic decomposition I: The basic algorithm*, Tech. Rept. CSD-427, Computer Science Dept., Purdue University, 1982.

[ARM84] D. S. ARNON AND S. MCCALLUM, *A polynomial-time algorithm for the topological type of a real algebraic curve – extended abstract*, Rocky Mountain J. Math., 14 (1984), pp. 849-852.

[ARN81] D. S. ARNON, *Algorithms for the geometry of semi-algebraic sets*, Ph.D. Dissertation, Technical Report No. 436, Computer Sciences Department, University of Wisconsin - Madison, 1981.

[ARN83] ——, *Topologically reliable display of algebraic curves*, Proceedings of SIGGRAPH '83, July 25-29, 1983, Detroit, Michigan, Assoc. Comp. Mach., pp. 219-227.

[BRT71] W. S. BROWN, J. F. TRAUB, *On Euclid's algorithm and the theory of subresultants*, J. Assoc. Comp. Mach., 18, 4 (1971), pp. 505-514.

[CLO82] G. E. COLLINS AND R. G. K. LOOS, *Real zeros of polynomials*, in Computing, Supplementum 4: Computer Algebra-Symbolic and Algebraic Computation, Springer-Verlag, Vienna and New York, 1982.

[COL75] G. E. COLLINS, *Quantifier elimination for real closed fields by cylindrical algebraic decomposition*, in Second GI Conference on Automata Theory and Formal Languages, vol. 33 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1975, pp 134-183.

[COL76] ——, *Quantifier elimination for real closed fields by cylindrical algebraic decomposition - a synopsis*, SIGSAM Bulletin of the ACM 10, 1 (1976), pp 10-12.

[DER79] N. DERSHOWITZ, *A note on simplification orderings*, Info. Proc. Letters, 9, 5 (1979), pp. 212-215.

[FER79] J. FERRANTE, C. RACKOFF, *The Complexity of Decision Procedures for Logical Theories*, Lecture Notes in Mathematics No. 718, Springer-Verlag, New York, 1979.

[HIL32] H. HILTON, *Plane Algebraic Curves*, Clarendon Press, Oxford, 1932 (2nd edition).

[HUE80] G. HUET AND D. C. OPPEN, *Equations and rewrite rules: a survey*, in Book, RV (ed.), Formal Language Theory: Perspectives and Open Problems, Academic Press, New York, 1980, pp. 349-405.

[KAH78] P. J. KAHN, Private communication to G. E. Collins, May 1978.

[KAH79] ——, *Counting types of rigid frameworks*, Inventiones math., 55, (1979), pp. 297-308.

[KAL82] E. KALTOFEN, *Polynomial factorization*, in Computing, Supplementum 4: Computer Algebra-Symbolic and Algebraic Computation, Springer-Verlag, Vienna and New York, 1982.

[KRE67] G. KREISEL AND J. L. KRIVINE, *Elements of Mathematical Logic (Model Theory)*, North-Holland, Amsterdam, 1967.

[LAN78] D. LANKFORD, Private communication to G.E. Collins, June, 1978.

[LOO82a] R. G. K. LOOS, *Computing in algebraic extensions*, in Computing, Supplementum 4: Computer Algebra-Symbolic and Algebraic Computation, Springer-Verlag, Vienna and New York, 1982.

[LOO82b] ——, *Generalized polynomial remainder sequences*, in Computing, Supplementum 4: Computer Algebra-Symbolic and Algebraic Computation, Springer-Verlag, Vienna and New York, 1982.

[MAS78] W. S. MASSEY, *Homology and Cohomology Theory*, Marcel Dekker, New York, 1978.

[MCC79] S. MCCALLUM, *Constructive triangulation of real curves and surfaces*, M.Sc. thesis, University of Sydney, 1979.

[MUE77] F. MÜLLER, *Ein exakter Algorithmus zur nichtlinearen Optimierung für beliebige Polynome mit mehreren Veranderlichen*, Verlag Anton Hain, Meisenheim am Glan, 1978.

[SCH83a] J. SCHWARTZ AND M. SHARIR, *Mathematical problems and training in robotics*, Notices Amer. Math. Soc., 30, 5, (1983), pp. 475-477.

[SCH83b] ——, *On the 'piano movers' problem II. General techniques for computing topological properties of real algebraic manifolds*, Advances in Applied Mathematics, 4 (1983) pp. 298-351.

[TAR48] A. TARSKI, *A Decision Method for Elementary Algebra and Geometry*, University of California Press, 1948; second edn., rev. 1951.

[WAE29] B. L. VAN DER WAERDEN, *Topologische Begründung des Kalküls der abzählenden Geometrie*, Math. Ann. 102 (1929), pp 337-362.